Redbooks

ibm.com/redbooks

SQL Procedures, Triggers, and Functions on IBM DB2 for i

Jim Bainbridge

Hernando Bedoya

Rob Bestgen

Mike Cain

Dan Cruikshank

Jim Denton

Doug Mack

Tom Mckinley

Simona Pacchiarini

Power Systems

IBM

Redbooks

International Technical Support Organization

SQL Procedures, Triggers, and Functions on IBM DB2 for i

April 2016

SG24-8326-00

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

First Edition (April 2016)

This edition applies to Version 7, Release 2, of IBM i (product number 5770-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®	Power Systems™	System i®
DRDA®	Rational®	z/OS®
i5/OS™	Redbooks®	
IBM®	Redbooks (logo)  ®	

The following terms are trademarks of other companies:

is a trademark or registered trademark of Ustream, Inc., an IBM Company.

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

Windows, 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 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.

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

Preface

Structured Query Language (SQL) procedures, triggers, and functions, which are also known as user-defined functions (UDFs), are the key database features for developing robust and distributed applications. IBM® DB2® for i supported these features for many years, and they are enhanced in IBM i versions 6.1, 7.1, and 7.2. *DB2 for i* refers to the IBM DB2 family member and relational database management system that is integrated within the IBM Power operating system that is known as IBM i.

This IBM Redbooks® publication includes several of the announced features for SQL procedures, triggers, and functions in IBM i versions 6.1, 7.1, and 7.2. This book includes suggestions, guidelines, and practical examples to develop DB2 for i SQL procedures, triggers, and functions effectively.

This book covers the following topics:

- Introduction to the SQL/Persistent Stored Modules (PSM) language, which is used in SQL procedures, triggers, and functions
- SQL procedures
- SQL triggers
- SQL functions

This book is for IBM i database engineers and data-centric developers who strive to provide flexible, extensible, agile, and scalable database solutions that meet business requirements in a timely manner.

Before you read this book, you need to know about relational database technology and the application development environment on the IBM Power Systems™ with the IBM i operating system.

Authors

This book was produced by the IBM DB2 for i Center of Excellence team in partnership with the International Technical Support Organization (ITSO), Rochester, Minnesota, US.

Jim Bainbridge is a senior DB2 consultant on the DB2 for i Center of Excellence team in the IBM Lab Services and Training organization. His primary role is training and implementation services for IBM DB2 Web Query for i and business analytics. Jim began his career with IBM 30 years ago in the IBM Rochester Development Lab, where he developed cooperative processing products that paired IBM personal computers with IBM System/36 and AS/400 systems. Since then, Jim has held numerous technical roles, including independent software vendor (ISV) technical support on a broad range of IBM technologies and products, and client support in the IBM Executive Briefing Center and IBM Project Office.

Hernando Bedoya is a Senior IT Specialist at STG Lab Services and Training in Rochester, Minnesota. He writes extensively and teaches IBM classes worldwide in all areas of DB2 for i. Before he joined STG Lab Services, Hernando worked in the ITSO for nine years, writing multiple IBM Redbooks publications. He also worked for IBM Colombia as an IBM AS/400 IT Specialist for presales support for the Andean countries. He has 28 years of experience in the computing field. Hernando taught database classes in Colombian universities. He holds a Master's degree in Computer Science from EAFIT, Colombia. His areas of expertise are database technology, performance, and data warehousing. Hernando can be contacted at hbedoya@us.ibm.com.

Rob Bestgen is a member of the DB2 for i Center of Excellence team, helping clients to use the capabilities of DB2 for i. In addition, Rob is the Chief Architect of the DB2 SQL Query Engine (SQE) for DB2 for i, and he is the Product Development Manager for DB2 Web Query for i.

Mike Cain is a Senior Technical Staff Member within the IBM Systems and Technology Group. He is also the founder and team leader of the DB2 for i Center of Excellence in Rochester, Minnesota, US. Before his current position, he worked as an IBM AS/400 Systems Engineer and Technical Consultant. Before he joined IBM in 1988, Mike worked as a System/38 Programmer and Data Processing Manager for a property and casualty insurance company. Mike has 26 years of experience with IBM, engaging clients and IBM Business Partners around the world. In addition to assisting clients, he uses his knowledge and experience to influence the IBM solution, development, and support processes.

Dan Cruikshank is a member of the DB2 for i Center of Excellence team with IBM Rochester Lab Services. Dan has worked as an IT Professional since 1972. Since he joined IBM Rochester in 1988, Dan has consulted on many project areas. Since 1993, Dan focused primarily on the resolution of IBM System i® application and database performance issues at several IBM client accounts. Since 1998, Dan has been one of the primary instructors for the Database Optimization Workshop.

Jim Denton is a Senior Consultant at the IBM DB2 for i Center of Excellence, where his responsibilities include both teaching courses and hands-on consulting. Jim specializes in SQL performance, data-centric programming, and database modernization. Jim started his IBM career in 1981 as a System/38 Operating System Programmer. Before he joined the consulting team, his key assignments included 10 years as a Systems Performance Specialist, five years as the Lead “JDE on i” Analyst, three years as a Consultant at the IBM Benchmark and Briefing Center in Montpellier, France, and a total of 11 years as an Operating System Developer, including five years designing and implementing enhancements to DB2 for i.

Doug Mack is a DB2 for i and Business Intelligence Consultant in the IBM Power Systems Lab Services organization. Doug’s 30+ year career with IBM spans many roles, including product development, technical sales support, Business Intelligence Sales Specialist, and DB2 for i Product Marketing Manager. Doug is a featured speaker at User Group conferences and meetings, IBM Technical Conferences, and Executive Briefings.

Tom Mckinley is an IBM Lab Services Consultant who works on IBM DB2 for i in Rochester, MN. His main focus is complex query performance that is associated with Business Intelligence that runs on extremely large databases. He worked as a Developer or Performance Analyst in the DB area 1986 - 2006. Several of his major works include the Symmetric Multiple Processing capabilities of DB2 for i and Large Object (LOB) data types. In addition, he was on the original team that designed and built the SQL Query Engine. Before his database work, he worked on Licensed Internal Code for System/34 and System/36.

Simona Pacchiarini works for IBM Italy. Since 1989, she has participated in IBM Redbooks projects about client connectivity, Windows cooperation, and database, first on the AS/400 and now on IBM i. She currently works in STG Lab Services in Italy, assisting clients on DB2 Web Query for i projects, DB2 for i performance, and IBM i system performance studies.

She can be contacted at simona_pacchiarini@it.ibm.com.

Debra Landon
International Technical Support Organization, Poughkeepsie Center

Scott Hanson, Sue Romano, Mark Anderson, Scott Forstie
IBM Rochester Development

Kent Milligan
IBM Watson

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

1

Introduction to data-centric programming

This book is for the IBM i database engineer and data-centric developer who strives to provide flexible, extensible, agile, and scalable database solutions that meet business requirements in a timely manner. The authors assume that the reader has an understanding of Structured Query Language (SQL) and a general knowledge of relational database concepts.

DB2 for i refers to the IBM DB2 family member and relational database management system that is integrated within the IBM Power operating system that is known as *IBM i*. The other DB2 family members are IBM DB2 for z/OS® and DB2 for Linux, UNIX, Windows. Each of these database software products has its own code base, which is highly optimized for its own hardware and software stacks. Database development and SQL support are fairly consistent across the family members, including support for procedures, functions, and triggers that are written in the SQL programming language.

SQL procedures, functions, and triggers are typified by their routine logic, which is contained within the CREATE statement. Because of this characteristic, we tend to characterize SQL procedures, functions, and triggers as SQL routines. In contrast, the routine logic of external procedures, functions, and triggers is implemented in high-level language (HLL) source code. SQL routines can contain and execute fewer SQL statements than external routines. However, they can offer equal power and high performance when they are implemented according to preferred practices. This book focuses on SQL routines.

This chapter includes the following topics:

- Data-centric programming
- Database engineering

1.1 Data-centric programming

Data-centric programming is both a concept and a methodology. It involves designing and developing logic and processes that reside in or near the database. Data-centric programming offers more efficient, more effective, and faster processing within the database server when compared to data processing in the application server, web server, or hand-held device. With data-centric programming, DB2 performs more of the work, which means less work for you.

SQL is the language of database, representing an extremely robust set of capabilities and techniques that enable data-centric programming. It is important to remember SQL, by itself, does not provide presentation or reporting capabilities. SQL is used to communicate with the database management system and facilitate operations on data, data that hopefully is modeled and stored according to sound relational principles and the correct normalization form.

Note: To fully appreciate SQL functionality, rules, and syntax, see the following resources:

- DB2 for i SQL reference, which is at this website:

  https://ibm.biz/Bd42dh
- SQL programming, which is at this website:

  https://ibm.biz/Bd42dA

SQL routines are a major contributor to data-centric programming success. They offer the greater manageability of components, separation of development roles between application developer and database engineer, and increased performance and scalability of data-centric applications.

1.2 Database engineering

Database engineering involves the architecture, design, and implementation of database systems and components in support of information management and analytical analysis. It includes understanding and applying the science and art of data modeling that is practiced through modern methods, tools, and interfaces. The database engineer (DBE) is involved in gathering business requirements, logical modeling, physical modeling, implementing the model, and maintaining the model.

The DBE, in partnership with the data owner, is responsible for the accessibility, availability, and integrity of the database and data. The DBE is responsible for defining and implementing the correct strategy and methods to maintain control and governance of the database and the data, as defined and mandated by the owner. This responsibility includes understanding and accounting for all of the potential data access interfaces and techniques, and ensuring that the correct security and auditing measures are in place.

The DBE is responsible for ensuring that correct, adequate data-centric techniques are used in the design, development, and implementation of applications. This responsibility includes taking advantage of built-in database management system features and functions, such as constraints, functions, and triggers. The DBE is responsible for understanding and applying the science and art of coding and implementing SQL routines through modern methods, tools, and interfaces. Leading, guiding, and reviewing data-centric application development is a fundamental task of the database engineer. The DBE is also responsible for identifying, understanding, reconciling, and meeting the data-serving performance and scalability requirements, which include “thinking in sets” and using set-at-a-time SQL requests.

Ultimately, this book serves as a practical guide and real-world reference for the database engineer when the DBE designs, codes, and implements DB2 for i SQL routines in support of a robust data-centric application.

2

Introduction to SQL Persistent Stored Module

This chapter describes the Structured Query Language (SQL) that is used to define and maintain persistent database language modules. It also describes how to invoke them by calling procedures, invoking functions, or attaching triggers.

You can use the support for Persistent Stored Modules (PSM) in IBM DB2 for i to write routines that use the SQL language that is based on the American National Standards Institute (ANSI)/International Organization for Standardization (ISO) PSM specification. The ANSI SQL implementation of DB2 for i, language elements, error handling, and transaction management in SQL PSM are described.

SQL PSM offers the following advantages:

- Common language. SQL is used throughout the IT industry to work with relational database management systems (DBMS).
- Portable. DB2 for i follows the SQL standard for PSM. Therefore, it is easier to port routines between different relational database management systems (RDBMS).
- Powerful. SQL PSM allows a natural mix of procedural logic with database access. It is simple to process data.
- Ongoing enhancements. SQL support in DB2 for i continues to be actively enhanced with each release, both in features and performance improvements.

This chapter includes the following topics:

- Introduction
- System requirements and planning
- Structure of an SQL PSM program
- SQL control statements
- Error handling
- Transaction management in procedures

2.1 Introduction

Implementation of the SQL Persistent Stored Module (PSM) in DB2 for i is based on industry standards. PSM supports constructors that are common to most programming languages. It supports the declaration of variables, control flows, the assignment of expression results to variables, receiving and returning parameters, returning result sets, and error handling.

How SQL PSM objects are invoked depends on how they are used. SQL PSM can be used in the following ways:

- *Procedures*: These procedures are invoked by using the SQL CALL statement, as described in Chapter 4, “Procedures” on page 73.
- *Triggers*: After the triggers are added and activated on a database table, these triggers are automatically called by DB2 for i when the trigger event occurs on the table, as described in Chapter 5, “Triggers” on page 107.
- *Functions*: These functions are referenced within SQL statements and they are automatically called by DB2 for i as necessary when the SQL statement is invoked. A *user-defined function* (UDF) can return either a scalar (single) value or an entire “logical” table result set, as described in Chapter 6, “Functions” on page 151.

2.2 System requirements and planning

DB2 for i, including its SQL capabilities, is integrated into the IBM i Operating System (OS). You can use SQL if you have an active IBM i system. No additional feature or product is necessary to run SQL PSM objects.

SQL scripts can be readily run by using the **RUNSQL** or **RUNSQLSTM** control language (CL) commands. IBM also provides specific IBM i SQL interfaces through its IBM i Access Client Solutions and Navigator for i interfaces.

For an additional charge, the IBM DB2 Query Manager and SQL Development Kit for i product (5770-ST1) can be purchased to provide “green screen” interfaces, such as CL commands **STRSQL** and **STRQMQRY**. IBM also provides SQL scripting and development tools, such as IBM Data Studio. Several third-party solutions are also available. In addition, because of the DB2 for i adherence to the SQL standard, most SQL scripting tools that use the Java Database Connectivity (JDBC), Open Database Connectivity (ODBC) or command-line interface (CLI) database interfaces can be used to connect to and run SQL against an IBM i system.

When DB2 for i executes the CREATE PROCEDURE, CREATE TRIGGER, or CREATE FUNCTION statement for an SQL PSM procedure, trigger, or function, DB2 for i uses a multiple phase process to create an Integrated Language Environment (ILE) C program or service program object (*PGM or *SRVPGM). During this process, DB2 for i generates intermediary ILE C code with embedded SQL statements. This ILE C code is then precompiled, compiled, and linked automatically.

In addition to its use on the system where it is created, the ILE C object can be saved and restored onto any system with the same or a newer OS release. The object can also be restored to an older OS release if all of the following conditions are met:

- The target release is specified on the save.
- The SQL that is involved in PSM is limited to SQL that is supported on that older release.
- And, SET OPTION TGTRLS was used in the SQL routine or trigger or on the **RUNSQLSTM** command to indicate that this body of SQL code body is built for a previous or specific release.

2.3 Structure of an SQL PSM program

An SQL PSM consists of the following parts:

- The CREATE statement. The CREATE statement indicates the beginning of the routine or trigger and distinguishes its type. The possible values are CREATE PROCEDURE, CREATE FUNCTION, and CREATE TRIGGER.
- The routine or trigger name.
- Parameter declarations for procedures and functions. SQL triggers do not have parameter declarations.
- Properties. These characteristics vary based on usage, as explained in the related chapters of this book.
- Options. These options control the way that the PSM object is created.
- Body. The body is the main portion of the routine. The body is a single or compound SQL statement. A compound statement is a set of statements that are enclosed by BEGIN and END clauses. A compound statement can also contain (embed) other compound statements.

Note: In PSM, a statement is delineated or “ended” by the semicolon (;) character. However, in certain situations, a semicolon might be considered needed but it is not. Most significantly, the BEGIN and DO keywords are not followed by a semicolon because they are part of the statement that follows them and they are not a statement on their own.

Example 2-1 is a procedure that is described later in this book that illustrates the main aspects of a PSM.

Example 2-1 A sample procedure

```
CREATE OR REPLACE PROCEDURE Add_New_Employee (
  -- Required parameters
     IN P_EMPNO CHAR(6),
     IN P_FIRSTNME VARCHAR(12),
   IN P_MIDINIT CHAR(1),
   IN P_LASTNAME VARCHAR(15),
      IN P_WORKDEPT CHAR(3),
   IN P_EDLEVEL SMALLINT,
   IN P_SALARY DECIMAL(9,2),
  -- Default parameters
   IN p_PHONENO CHAR(4) DEFAULT NULL,
   IN p_HIREDATE DATE DEFAULT CURRENT DATE,
   IN p_Job CHAR(8) DEFAULT NULL,
   IN p_Gender CHAR(1) DEFAULT NULL,
   IN p_Birthdate DATE DEFAULT NULL,
```

```
   IN p_Bonus DECIMAL(9,2) DEFAULT NULL,
   IN p_Comm DECIMAL(9,2) DEFAULT NULL)

   SPECIFIC ADDNEWEMP

  P1 : BEGIN ATOMIC

     DECLARE EXIT HANDLER FOR SQLSTATE '23505'
       RESIGNAL SQLSTATE '70001'
          SET MESSAGE_TEXT='Duplicate employee number';

   INSERT INTO employee (empno, firstnme, midinit, lastname, workdept,
             edlevel, salary, phoneno, hiredate, job, gender,
             birthdate, bonus, comm)
      VALUES (P_EMPNO, P_FIRSTNME, P_MIDINIT, P_LASTNAME, P_WORKDEPT,
              P_EDLEVEL, P_SALARY, p_phoneno, p_hiredate, p_job, p_gender,
              p_birthdate, p_bonus, p_comm) ;
END P1
```

Note: In PSM, and in SQL in general, comments can be indicated in one of two ways:

- The forward slash and asterisk (/*) and the asterisk and forward slash (*/) bracket a comment and indicate that everything between them is a comment.
- The double hyphen (--) indicates that everything after it to the end of the line is a comment.

A more in-depth analysis of SQL procedures, triggers, and functions is covered in each of their chapters.

For a complete SQL syntax, see DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

2.4 SQL control statements

The *control statements* within PSM are the logic that controls the flow through the code when PSM is invoked. A control statement is one of the main statements that can be placed in the routine *body* of an SQL PSM. The different constructs are listed:

- Assignment statements
- Conditional control statements
- Iterative control statements
- Calling statement
- Compound statements

2.4.1 Assignment statement

In SQL PSM, the *assignment* statement is used to assign a value to a variable or to an INOUT or output parameter of a procedure. The assignment statement can be accomplished in two ways:

- SET statement
- INTO clause

SET

The SET statement is the more recognizable assignment form because it most closely matches how other languages assign variables. In this form, the statement starts with the **SET** keyword and the assignment follows.

Example 2-2 illustrates simple assignment statements.

Example 2-2 SET examples

```
SET num_records = num_records + 1;
SET credit_limit = credit_limit * 1.20;
SET num_orders = NULL;
SET max_sales = (SELECT max(sales) FROM sales);
```

Multiple assignment operations can be specified on a single SET statement. This capability is called *repeated assignment* and it can be useful to produce less verbose code.

Note: In a repeated assignment, the statements are processed independently and they are not aware of calculations in a preceding assignment in the same SET.

In Example 2-3, which shows repeated assignments, the value of the variable *v2* is `3` after the last SET statement is run, not `4`.

Example 2-3 Repeated assignment SETs

```
DECLARE v1, v2 INT;
SET v1=0, v2=0;        -- Repeated assignment
SET v1=1, v2=v1+3;     -- Repeated assignment
```

Example 2-3 differs from Example 2-4 where the value of the variable *v22* is `4` after the last SET statement is run.

Example 2-4 Separate SETs

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

In a repeated assignment statement, all operations in the statement are treated as a single statement. This effect is most notably seen by realizing that any check on the success or failure of the statement, which is commonly performed by GET DIAGNOSTICS, is a check on all operations in the SET statement. If it is important to distinguish between assignments, separate each assignment into its own SET statement.

If the expression that is assigned is a subselect (SELECT), the expression must be wrapped in parentheses.

Note: For more complex assignments, sometimes a performance advantage is possible by combining multiple assignments into a single SET statement. Complex assignments often require external calls to the database. By combining multiple complex assignments into a single SET, fewer external calls are made.

However, this advantage is not true for simple assignments where a variable is assigned a simple value. In fact, the combination of multiple simple assignments into a single SET statement can degrade performance. SQL converts simple, single operation SETs, for example, SET var = 0, into low-level, inline operations. However, if multiple simple assignment operations are combined into a single SET statement, an external call is made.

When in doubt, use a separate SET statement for each simple variable assignment.

Another assignment example

In Example 2-5, an SQL SELECT statement is constructed by using parameters that are passed to the SQL procedure. In this case, the CONCAT operator “glues” the strings together.

Example 2-5 Dynamic statement that uses concatenation

```
CREATE PROCEDURE MYMAX
       (IN fld_name VARCHAR(30),
        IN file_name VARCHAR(128))
  LANGUAGE SQL
  BEGIN ATOMIC
      DECLARE sql_stmt CHAR(500);
...
      SET sql_stmt = ' SELECT ' CONCAT fld_name CONCAT ' FROM '
                     CONCAT file_name CONCAT ' ORDER BY 1';
...
END
```

INTO

Although INTO is not strictly a PSM-only construct, the INTO clause comes from the SQL language because it is embedded in other high-level languages (HLLs). Therefore, INTO is a natural addition to PSM programming. The INTO clause is paired with either the SELECT (including WITH) or VALUES statement. The use of VALUES with INTO is powerful with dynamic SQL, which is described in the dynamic SQL section. The SELECT or VALUES indicate the values, and the INTO clause identifies the variables. Example 2-6 illustrates simple INTO assignment statements.

Example 2-6 INTO samples

```
VALUES(num_records + 1) INTO num_records;
VALUES(credit_limit * 1.20) INTO credit_limit;
VALUES(NULL) INTO num_orders;
SELECT max(sales) INTO max_sales FROM sales;

/* dynamic statement using VALUES INTO */
BEGIN
 DECLARE d_sql VARCHAR(100);
 DECLARE table_name VARCHAR(20) DEFAULT ‘MYTABLE’;
 DECLARE result INT;
 SET d_sql = ‘VALUES(SELECT COUNT(*) FROM ‘ CONCAT table_name CONCAT ‘) INTO ?’;
 PREPARE getcount FROM d_sql;
 EXECUTE getcount USING result;
END;
```

Note: When the INTO is used with the SELECT statement, the INTO is “embedded” in the statement and placed between the SELECT and FROM clauses.

As with the SET statement, multiple assignment operations can be specified on a single INTO clause. When multiple values are involved, all values are computed first, then the assignment to the variables is performed.

In Example 2-7, the value of the variable *v2* is `3` (not `4`) after the last VALUES INTO statement is run because the values are calculated first and will be (`1`,`3`).

Example 2-7 INTO, repeated assignments

```
DECLARE v1, v2 INT;
/* Examples of repeated assignments */
VALUES(0, 0) INTO v1, v2;
VALUES(1, V1+3) INTO v1, v2;
SELECT max(sales), sum(sales) INTO max_sales, total_sales FROM sales;
```

When INTO is used, the statement must produce at most one row. Otherwise, a runtime error is signaled. If no row is returned from the SELECT, the null value is assigned to the variables.

Assignment rules

The following rules apply when you use the assignment statement:

- The assignment statements in SQL PSM must conform to the SQL assignment rules and to the SQL arithmetic operators. The data type of the target and source must be compatible.
- An error is raised when a string is assigned to a variable that uses the `DEFAULT` value, a column, a parameter of a function or procedure, or to a transition variable, and the string is longer than the length attribute of the target.
- If truncation of the whole part of a number occurs on an assignment, an error is raised.
- If a `NULL` value is assigned to a variable that is not null capable, an error is raised. A NULL result can be generated as the result of a failed operation or if no value is produced.
- If the assignment is to a parameter, the parameter must be an output or INOUT type in the procedure for the caller to “see” the assignment. Although input parameters can be assigned and used within a routine, any changes to input parameters are not seen by the caller. Function parameters are always defined as input parameters, and triggers do not have parameters.

2.4.2 Conditional control

Conditional statements control the flow through the code. SQL PSM has four conditional control structures:

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

The three IF-THEN structures are of a similar type and all end with the END IF statement. However, they have flow differences.

IF-THEN

The IF-THEN conditional control structure tests a condition. If the condition evaluates to TRUE, the lines of code up to the associated END IF are executed. If the condition evaluates to FALSE, the control of the program is passed to the next statement after the associated END IF. Example 2-8 illustrates this case.

Example 2-8 IF THEN

```
IF ref_error = 1
   THEN SET o_error = 'NOT FOUND';
END IF;
```

IF-THEN-ELSE

The IF-THEN-ELSE conditional control structure tests a condition. If the condition evaluates to TRUE, the lines of code up to the associated ELSE are executed. If the condition evaluates to FALSE, the lines of code that start after the associated ELSE and up to the associated END IF are executed. Example 2-9 illustrates this case.

Example 2-9 IF THEN ELSE

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

IF-THEN-ELSEIF

The IF-THEN-ELSEIF conditional control structure is an alternative to the use of the nested IF-THEN-ELSE structure. For example, the nested IF-THEN-ELSE in Example 2-9 can be rewritten to a simpler form, as shown in Example 2-10.

Example 2-10 IF THEN ELSEIF

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

No matching END IF is available with each ELSEIF, which is the main advantage over the nested form.

CASE

You can use the CASE conditional control structure to select different execution paths based on multiple different conditions.

A CASE consists of a set of WHEN/THEN clauses where the THEN action is executed if the corresponding WHEN condition is TRUE.

Note: Although the CASE statement is also in SQL statements, it is different from this PSM control CASE. The CASE in an SQL statement is an expression, and it returns a value. This CASE is strictly for code flow control.

Two ways exist to code the CASE statement: as a simple-when-clause or as a searched-when-clause.

SIMPLE-WHEN-CLAUSE

The simple case form is ideal to compare a value to a series of possible values, choosing a different execution path for each compare value. A certain value can be anything from a simple constant to a complex scalar expression, as shown in Example 2-11.

Example 2-11 CASE, SIMPLE WHEN

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

SEARCHED-WHEN-CLAUSE

The searched when case form is necessary when the comparisons are more complex than a series of simple equal comparisons. Unlike the simple when clause, a searched when clause can be an arbitrarily complex logical expression that involves ANDs, ORs, and other operators. As with the first form of CASE, a certain value can be anything from a simple constant to a complex scalar expression as shown in Example 2-12.

Example 2-12 CASE, SEARCHED WHEN

```
CASE
 WHEN evaluation BETWEEN 91 AND 100 THEN UPDATE employee SET salary = salary * 1.3;
 WHEN evaluation BETWEEN 81 AND 90  THEN UPDATE employee SET salary = salary * 1.2;
 WHEN evaluation BETWEEN 71 AND 80  THEN UPDATE employee SET salary = salary * 1.1;
 ELSE  UPDATE employee SET salary = salary * 1.05;
END CASE;
```

CASE RULES

When multiple WHEN/THEN clauses are specified in a specific CASE, the order of processing is left to right and top to bottom until a WHEN clause is TRUE. If multiple WHENs evaluate to TRUE, the THEN of the first WHEN that is TRUE is executed and the rest of the WHEN/THENs are skipped. See Example 2-13. The highlighted line is explained after the example.

Example 2-13 CASE, overlapping WHENs

```
CASE
 WHEN evaluation > 90 THEN UPDATE employee SET salary = salary * 1.3;   1
 WHEN evaluation > 80 THEN UPDATE employee SET salary = salary * 1.2;
 WHEN evaluation > 70 THEN UPDATE employee SET salary = salary * 1.1;
 ELSE  UPDATE employee SET salary = salary * 1.05;
END CASE;
```

Notes about Example 2-13 on page 13: If the value of variable evaluation is greater than 90, it satisfies the first WHEN (1) clause condition. However, it also satisfies the other WHEN clause conditions. Due to the order of processing, the first WHEN is found to be TRUE, salary is multiplied by 1.3, and the rest of the CASE statement is ignored.

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

Case expressions can be nested.

Note: If the order of the WHENs is not important, performance can be improved by ordering the WHEN clauses so that the WHEN clauses that most often evaluate TRUE are put before the WHEN clauses that are rarely true. This approach assumes that the complexity of the WHENs is similar.

When you decide whether to use the simple versus searched form of CASE, if you can use the simple form, it is normally the better approach because less processing is involved. An example can illustrate this point.

Compare equivalent statements in Example 2-14 and Example 2-15.

Example 2-14 CASE, simple form

```
CASE (base+change)
  WHEN 1 THEN ...;
  WHEN 2 THEN ...;
  WHEN 3 THEN ...;
  WHEN 4 THEN ...;
END CASE;
```

Example 2-15 CASE, searched WHENs

```
CASE
  WHEN (base+change) = 1 THEN ...;
  WHEN (base+change) = 2 THEN ...;
  WHEN (base+change) = 3 THEN ...;
  WHEN (base+change) = 4 THEN ...;
END CASE;
```

In the simple form of Example 2-14, the database evaluates the expression (base+change) one time and uses the resulting value in the comparison of each WHEN clause. In the searched form of Example 2-15, the database evaluates the expression (base+change) for each WHEN, resulting in a repeated process.

The performance difference becomes even more obvious if the expression contains a function or scalar subselect.

In addition to a performance difference, a functional difference can exist between the simple and searched when forms. Because the expression is reevaluated in the searched when case, if the expression performs an external action, such as invoking a function that affects a table's contents or that is affected by a table's changes, the function's actions and return value can change from WHEN to WHEN.

2.4.3 Iterative control

Iterative control statements provide a way to repeat a series of instructions for a certain number of times or until a condition is satisfied. Iterative control statements are also known as *looping structures*.

Iterative control statements are powerful statements because they repeat or “loop”, and they can drive significant processing in less code. It is important to code iterative control statements with a clear definition of the condition that ends the repetition and when the condition is satisfied.

Four iterative control statements are available:

- LOOP
- WHILE
- REPEAT
- FOR

In addition, two statements can directly affect iterative statements:

- LEAVE
- ITERATE

LOOP

A LOOP iterative control statement repeats a series of instructions indefinitely. A LOOP repetition ends only when another action occurs, usually with the LEAVE statement. Therefore, the LOOP and LEAVE statements are often paired together. Example 2-16 shows a LOOP statement.

Example 2-16 A LOOP statement

```
the_loop:                                  1
LOOP
  CALL work_to_do(all_done);
  IF all_done = 1 THEN
   LEAVE the_loop;                         2
  END IF;
END LOOP the_loop;                         3
```

Notes about Example 2-16:

1 A label can be defined for a LOOP statement.
2 A LEAVE statement is used to avoid an endless execution.
3 Every LOOP statement has an associated END LOOP clause.

Although LEAVE is a common way to get out of a loop, other ways are possible. A handler can be used to complete the process. Alternatively, a RETURN statement can be used to completely leave PSM.

WHILE

The WHILE programming structure repeats the statements between the WHILE and END WHILE clauses while the specified condition is true. It is important to note that the exit condition is checked in the WHILE condition before the start of an iteration. Therefore, the values that are checked in the WHILE condition must be set before the WHILE. For this reason, it is possible that the WHILE “loop” might never occur if the WHILE condition is false on the first check.

After the code is in the iteration, the WHILE condition is checked at the end of the iteration before the next iteration begins.

As with the other iterative statements, the LEAVE statement can be used with the WHILE statement. See Example 2-17.

Example 2-17 WHILE and END WHILE

```
SET all_done = 0;           1
the_loop:                   2
WHILE all_done = 0 DO       3
  CALL work_to_do(all_done, hit_error);
  IF hit_error = 1 THEN
   LEAVE the_loop;          4
  END IF;
END WHILE the_loop;         5
```

Notes about Example 2-17:

1 In advance, initialize the value that is checked in the WHILE.
2 Label the WHILE to help identify the structure, particularly if multiple loops exist.
3 The WHILE condition is checked at the beginning of each iteration.
4 Optional: A LEAVE can be used with a WHILE.
5 Every WHILE statement ends with an END WHILE.

REPEAT

With this programming structure, the statements that are between the REPEAT and END REPEAT are executed until the specified condition is true. In contrast to the WHILE statement, on the REPEAT statement, the exit condition is tested at the end of each iteration. Therefore, at least one iteration is executed.

As with the other iterative statements, the LEAVE statement can be used with the REPEAT statement, as shown in Example 2-18.

Example 2-18 REPEAT and END REPEAT

```
the_loop:
REPEAT                                  1
  CALL work_to_do(all_done, hit_error);
  IF hit_error = 1 THEN
   LEAVE the_loop;
  END IF;
UNTIL all_done=1                        2
END REPEAT the_loop;                    3
```

Notes about Example 2-18:

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

FOR

The FOR statement is the last type of iterative structure. It executes a statement for each row that is returned from a SELECT statement. The FOR statement is unique from other iterative structures because it is used only for processing rows from a SELECT statement. It can be considered a concise way to perform a cursor open/fetch/close loop. In fact, the **CURSOR** keyword can be specified within the FOR clause.

See Example 2-19.

Note: An SQL procedure statement cannot include an OPEN, FETCH, or CLOSE that specifies the same cursor name as a FOR statement. Also, the cursor name in a FOR must not be the same as the name of another cursor that is declared in the SQL procedure. If a cursor name is not specified on the FOR clause, a unique name is generated.

Example 2-19 FOR statement

```
for_loop: FOR each_birthday AS                                    1
   birthday_cursor CURSOR FOR                                     2
      SELECT empno FROM employee WHERE birthdate = current date
         DO
            CALL send_email(each_birthday.empno);                 3 4
            UPDATE employee                                       4
               SET bonus = bonus + 500
               WHERE CURRENT OF birthday_cursor;                  5
END FOR;                                                          6
```

Notes about Example 2-19:

1 The *each_birthday* is known as the *variable name* of the FOR. The for_loop is known as the *label* of the FOR. The variable name (*each_birthday*) can be used within the FOR structure to qualify variables that represent the values of the columns in the FOR associated cursor.

2 The cursor name is birthday_cursor.

3 A value from the FOR select can be qualified by the variable name.

4 This statement is executed for each row that is returned from the cursor1 cursor.

5 The FOR cursor can be referenced in the WHERE CURRENT OF clause.

6 Every FOR statement ends with an END FOR statement.

Important: Be careful not to perform operations within the FOR loop that can close cursors. In particular, if a procedure is called within the FOR loop, the procedure must *not* perform a COMMIT or the FOR loop's underlying cursor will be closed and the FOR loop will end.

LEAVE

You can use the LEAVE statement to break out of a loop or go out of a block. The execution continues with the first statement that follows the loop or block statement.

ITERATE

The ITERATE statement forces the next iteration in a loop structure. It works for any of the iterative control statements. See Example 2-20.

Example 2-20 Iterate

```
for_loop: FOR each_birthday AS
   cursor1 CURSOR FOR
      SELECT empno, job FROM employee WHERE birthdate = current date
         DO
```

```
            IF each_birthday.job = 'MANAGER' THEN
              CALL send_email(each_birthday.empno);
              ITERATE each_birthday;                          1
            END IF;
            CALL send_email_bonus(each_birthday.empno, 500);
            UPDATE employee
               SET bonus = bonus + 500
               WHERE CURRENT OF cursor1;
END FOR;
```

Note about Example 2-20:

1 The ITERATE causes everything after it to the END FOR to be skipped, and the loop moves to the next row of the cursor's answer set.

2.4.4 Calling procedures

You invoke a PSM procedure by using the CALL control statement.

Example 2-21 shows the invocation of a procedure that has default parameters.

Example 2-21 CALL procedure

```
for_loop: FOR each_birthday AS
   cursor1 CURSOR FOR
      SELECT empno, job FROM employee WHERE birthdate = current date
         DO
            IF each_birthday.job = 'MANAGER' THEN
              CALL send_email(each_birthday.empno);             1
              ITERATE each_birthday;
            END IF;
            CALL send_email(each_birthday.empno, bonus=>500);   2
            UPDATE employee
               SET bonus = bonus + 500
               WHERE CURRENT OF cursor1;
END FOR;
CALL QSYS2.QCMDEXC('DSPJOBLOG OUTPUT(*PRINT)');                 3
```

Notes about Example 2-21:

1 This call to the send_email procedure (procedure not shown) uses the default for all parameters, except empno.

2 In this call to send_email, a value is passed for the bonus parameter.

3 This line is an example of the use of the DB2 for i supplied procedure QCMDEXC to perform a system command.

Example 2-21 used QCMDEXC, which is one of several procedures that are provided automatically by DB2 for i. Other supplied procedures can be viewed by using one of the database interfaces, such as Navigator for i, or by accessing the procedures catalog by using the following SQL statement:

```
SELECT ROUTINE_NAME FROM qsys2.sysprocs WHERE routine_schema = 'QSYS2'
```

2.4.5 Compound SQL statement

According to the SQL PSM specification, an SQL routine (that is, a stored procedure or a function) consists of a single SQL statement. The specification also introduces the concept of a compound statement.

A *compound statement* consists of a BEGIN and END block and any number of SQL statements that are contained within the block. The body of a non-trivial SQL routine is, in fact, a compound statement. Example 2-22 shows the general structure of a compound statement. *The various constructs must be ordered as listed.*

Example 2-22 Structure of a compound SQL statement

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

In most cases, the body of an SQL PSM needs more than one statement. Therefore, a compound statement is essential to PSM, as shown in Example 2-23.

Example 2-23 Compound statement

```
BEGIN NOT ATOMIC                                                          3 4
   DECLARE totsales DECIMAL(11,2);                                        1
   DECLARE countsales INTEGER;
   SET (totsales, countsales) = (select sum(sales), count(*) from sales); 2
   CALL PROC1 (totsales);                                                 2
END                                                                       3
```

Notes about Example 2-23:

1 The SQL code in Example 2-23 includes variable declarations.

2 The SQL code includes two control statements: SET and CALL.

3 The BEGIN and END wrap the control statements into a single compound statement.

4 The compound statement is defined as NOT ATOMIC. Because NOT ATOMIC is the default, this definition was not strictly necessary. However, the definition helps to point out that the statements within the compound statement are not required to apply together as a single unit. This example has no data-altering statements, so NOT ATOMIC is correct. If multiple data change statements exist, such as an INSERT and an UPDATE, the developer needs to determine whether the compound statement will treat them as ATOMIC (either both statements succeed or neither statement occurs) or NOT ATOMIC (one statement can occur without the other statement). Chapter 3, “SQL fundamentals” on page 47 covers ATOMIC in more detail.

Nested compound statements

Compound statements can be placed inside other compound statements. Or, a BEGIN/END block can be placed inside another BEGIN/END block. This coding is common in complex PSM to scope constructs, such as variables, cursor declarations, and condition handlers. Only constructs that were defined within the same or enclosing (higher) compound statements are visible. Otherwise, statements within a compound statement cannot refer to constructs in other compound statements in the same PSM.

Nested compound statements are useful to improve flexibility in programming and readability in the source code, for example, in declaring error handlers.

Variable declaration

Within a compound statement, variables can be declared that are locally scoped to that compound statement. Those variables can be used for many purposes:

- Calculations
- Counters
- A holder for values that are eventually assigned to output parameters
- A variable in a data access statement, such as a table update
- An input parameter that is passed to a procedure that is called
- Error handling

Any valid SQL data type can be used for variables, as shown in Example 2-24.

Example 2-24 Defining a valid SQL data type

```
BEGIN
   DECLARE total_sales     DECIMAL(11,2) NOT NULL DEFAULT 0;
   DECLARE number_customer DECIMAL(5);
   DECLARE err_msg         CHARACTER(10) DEFAULT 'ALL CLEAR';
   DECLARE timestamp_order TIMESTAMP;
   DECLARE date_order      DATE;
   DECLARE picture         BLOB(10M) DEFAULT NULL;
   DECLARE id_ary          smallint_array; 1
   .....
END
```

Note about Example 2-24:

1 This statement assumes that the array type smallint_array was created earlier, for example:

```
CREATE TYPE smallint_array AS SMALLINT ARRAY[99];
```

A variable is always assigned a default value. If the **DEFAULT** keyword is specified, the corresponding value is assigned as the variable's initial value. If no default is specified, the database assigns the NULL value to the variable. If a variable is defined as NOT NULL capable, as *total_sales* was in Example 2-24, a default value must be provided.

Multiple variables can be declared in a statement. This form is more compact than separate declarations, and it is useful when multiple variables are set to the same initial value:

```
DECLARE i, j, k INT DEFAULT 0;
```

Note: Unlike the SET statement, no performance penalty occurs for assigning multiple variables in a single DECLARE clause. Default values are always generated as inline code in PSM.

Data types

Generally, all of the data types that are supported on the CREATE TABLE statement are also supported by the SQL stored procedures.

For more information, see DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

In addition, array types, such as the array types that are shown in the *id_ary* variable in Example 2-24 on page 20, are supported.

Using cursors

When SQL runs a SELECT statement, the resulting rows comprise a result set. A *cursor* provides a way to access that result set. SQL uses a cursor to work with the rows in the result set and to make them available to PSM. The cursor also maintains the position in the result set.

For PSM, cursors can be declared by using the DECLARE statement or cursors can be defined in the FOR loop.

Example 2-25 uses a cursor to process through a query result set. For each row in the result set, it calls a procedure, which is named send_email, to send email with the generated text. The example uses dynamic SQL, which will be described in the next section. The query uses more advanced SQL techniques, such as a common table expression and the ROW_NUMBER online analytical processing (OLAP) function.

Example 2-25 Using a cursor

```
CREATE OR REPLACE PROCEDURE send_thanks(                    1
 IN i_date DATE,
 OUT o_date DATE,
 IN i_topn INTEGER DEFAULT -1)
 LANGUAGE SQL
 BEGIN
   DECLARE at_end SMALLINT DEFAULT 0;
   DECLARE d_sales_rank INT DEFAULT 0;
   DECLARE d_sales_date DATE;
   DECLARE d_sales_person VARCHAR(100);
   DECLARE d_daily_sales DECIMAL(9,2);
   DECLARE d_award_text VARCHAR(200);
   DECLARE d_sql VARCHAR(3000) DEFAULT                      2
      'WITH thesales AS (
       SELECT ROW_NUMBER()
                OVER(PARTITION BY sales_date ORDER BY SUM(sales) DESC) sales_rank,
              sales_date, sales_person, SUM(SALES) daily_sales
       FROM SALES
       WHERE sales_date >= ?                                 11
       GROUP BY sales_date, sales_person)
       SELECT * from thesales';
   DECLARE sales_cursor CURSOR FOR sales_list;              3
   DECLARE CONTINUE HANDLER FOR NOT FOUND                   4
      SET at_end = 1;

   IF i_topn > 0 THEN /* limit per day thanks */            5
      SET d_sql = d_sql CONCAT ' WHERE sales_rank <= ' CONCAT CHAR(i_topn);
    END IF;
    SET d_sql = d_sql CONCAT ' ORDER BY sales_date';

   PREPARE sales_list FROM d_sql;                           6
   OPEN sales_cursor USING i_date;                          7
    FETCH sales_cursor INTO d_sales_rank, d_sales_date, d_sales_person, d_daily_sales;
   WHILE at_end = 0 DO                                      8
     SET d_award_text = 'Congratulations on your sales of $' CONCAT
TRIM(CHAR(d_daily_sales))
                     CONCAT ' on ' CONCAT CHAR(d_sales_date);
     CALL send_email(d_sales_person, d_award_text);
      FETCH sales_cursor INTO d_sales_rank, d_sales_date, d_sales_person, d_daily_sales;
```

```
  END WHILE;
  CLOSE sales_cursor;                                    9
  IF d_sales_date IS NOT NULL THEN
    SET o_date = d_sales_date;                           10
  ELSE
    SET o_date = i_date;                                 10
  END IF;
END
```

Notes about Example 2-25:

1 The procedure's name is send_thanks.

2 The DEFAULT clause is used to set the initial value for the variable.

3 The DECLARE cursor follows the local variable declarations.

4 The exception handler follows cursor and variable declarations.

5 The conditional addition of a WHERE clause to the cursor statement depends on whether a top N value is specified.

6 PREPARE the dynamic statement (held in variable *d_sql*) to use with cursor sales_cursor (3) by using prepare statement name sales_list to tie the prepared statement to the cursor.

7 The cursor sales_cursor (of the dynamic statement) is opened. The input date i_date is used for the parameter marker's (11) value.

8 While loop through all of the result set of the cursor.

9 Close the cursor when all rows are read.

10 Conditionally set output variable *o_date* before the procedure ends.

2.5 Dynamic SQL in PSM

Two ways are available to use SQL statements: static SQL and dynamic SQL.

Dynamic SQL is characterized by statements that are known only at run time, such as when a statement is built up from the program logic and then the statement is executed.

Dynamic SQL offers a high degree of application flexibility. The SQL statement can be assembled based on various factors, including parameter values that are received from the application interface or through global variables. The use of global variables is described in Chapter 3, "SQL fundamentals" on page 47.

Many types of SQL statements can be processed as dynamic statements. Several examples are shown:

- SELECT: SELECTs are used for cursors and for INTO statements. They are the most common way, other than parameters, to get information into a PSM application. Dynamic SQL is a good way to generate these SELECT statements, particularly when the environment is unknown until run time.
- UPDATE/DELETE/INSERT: These set-based statements are easily generated into a dynamic statement because they are stand-alone statements.
- CREATE: The creation of almost any SQL object is possible by using dynamic SQL.
- BEGIN/END compound statements: It is possible to generate an entire BEGIN/END block of statements and process it as dynamic SQL.

Example 2-26 shows the dynamic SELECT.

Example 2-26 Dynamic SELECT

```
SET v_sqlstring = 'SELECT empno, salary, bonus, comm FROM EMPLOYEE WHERE edlevel = 17';
```

In addition, dynamic SQL offers variable substitution in the way of *parameter markers*.

Parameter markers are markers in the SQL statement where values can be substituted at run time. A parameter marker is indicated as a question mark (?) within the SQL statement. When the statement is executed, a value is passed at the time of the execution and substituted in place of the marker. See Example 2-27.

Example 2-27 Usage of parameter marker in the WHERE clause

```
SET v_sqlstring = 'SELECT empno, salary, bonus, comm FROM EMPLOYEE WHERE edlevel =
?';
```

Note: If multiple parameter markers are specified in the SQL statement, the values (that are separated by commas) that are provided during execution are substituted in order as the parameter markers are identified by reading through the SQL statement left to right, top to bottom, without regard to the statement's organization or definition.

The parameter markers provide an efficient way to perform a "build once/run many" statement. Of more importance, the parameter markers protect against potential errors and accidental mishaps that can happen when you construct a dynamic SQL statement from a set of variables. In particular, the parameter markers can combat potential security hacking, such as "SQL injection".

Several limitations exist where parameter markers cannot be used. For example, parameter markers cannot be used as a substitution for a table reference. Parameter markers are most commonly used in selection, that is, the WHERE clause of SQL statements, although they can be used in many other places, too. For more information about parameter markers, see the DB2 for i SQL reference at this website:

https://ibm.biz/Bd42dh

The following statements are used in PSM to process dynamic SQL statements:

- DECLARE CURSOR, PREPARE, and OPEN
- PREPARE then EXECUTE
- EXECUTE IMMEDIATE (

2.5.1 DECLARE CURSOR, PREPARE, and OPEN

These statements are used together when you work with dynamic select statements that are declared as cursors. Although these statements are not required, the SQL select statement is typically constructed as a text string and stored in a variable.

Because the select statement is a dynamic statement, the use of parameter markers is allowed in the select statement. If parameter markers are used, the values for the parameter markers are provided on the OPEN statement.

Note: No certain order is required for the DECLARE and PREPARE statements. Either statement can precede the other statement. Because the DECLARE CURSOR must be defined in the DECLARE section of a compound statement, the DECLARE CURSOR normally comes before the PREPARE statement in the same compound statement. If the PREPARE comes first, the DECLARE CURSOR must be specified in a subsequent nested compound statement.

The OPEN statement must appear after both the DECLARE CURSOR and PREPARE statements.

DECLARE CURSOR

The DECLARE CURSOR statement associates a select statement to a cursor, which can be used to read the resulting rows from the select statement. The general form of the statement is shown:

```
DECLARE <cursor name> CURSOR FOR <prepared statement name>
```

The *cursor name* identifies the cursor that is used later for the OPEN, FETCH, and CLOSE statements. The *prepared statement name* must match the name that is used on the PREPARE statement.

For more information about the syntax of the DECLARE CURSOR statement, see the DB2 for i SQL reference at the following website:

https://ibm.biz/Bd42dh

PREPARE

The PREPARE statement prepares an SQL select statement for execution. The most common, simple form is shown:

```
PREPARE <statement name> FROM <variable>
```

or

```
PREPARE <statement name> FROM <expression>
```

The *statement name* provides a name that is used to associate the statement to a cursor. The prepared statement name must match the DECLARE CURSOR prepared statement name to successfully pair the two together.

The PREPARE statement creates an executable form of an SQL statement from a character-string form of the statement. The character-string form is called a *statement string*, and the executable form is called a *prepared statement*.

During the PREPARE, the statement is validated and all referenced objects, such as tables, are resolved. Any syntax problems with the statement text or unresolved objects causes an error.

For more details about the syntax of the PREPARE statement, see the DB2 for i SQL reference at the following website:

https://ibm.biz/Bd42dh

OPEN

The OPEN statement makes the resulting rows of the statement available for consumption. The syntax is shown:

```
OPEN <cursor name>
```

If parameter markers were specified in the select statement, the syntax includes the USING clause to provide the values for the parameter markers. That syntax is shown:

```
OPEN <cursor name> USING <var1>, <var2>, and so on
```

WORKING TOGETHER

A procedure that contains the process of declaring, preparing, and opening a dynamic cursor is shown in Example 2-28. In this example, the cursor is used to return a result set of rows to the caller of the procedure.

Example 2-28 DECLARE_and_PREPARE procedure

```
CREATE OR REPLACE PROCEDURE Declare_And_Prepare()
RESULT SETS 1

P1: BEGIN

DECLARE v_sqlString CLOB (2M);                                                    1
DECLARE employee_cursor CURSOR FOR Employee_statement;                            2
SET v_sqlString = 'SELECT empno, lastname, firstnme, midinit FROM employee';      3
PREPARE Employee_statement FROM v_sqlString;                                      4
OPEN employee_cursor;                                                             5

END P1
```

Notes about Example 2-28:

1 An SQL statement can be a maximum of 2 megabytes. A character large object (CLOB) can be used to contain large statement strings.

2 The DECLARE CURSOR statement name Employee_statement is used to associate the prepared statement name that is assigned by the PREPARE statement (4).

3 The statement string variable is populated with the SQL statement text.

4 The PREPARE statement assigns a statement name to the SQL statement string and validates the string syntax. The statement name must match the name in the cursor (2). The PREPARE must precede the OPEN CURSOR statement.

5 The OPEN CURSOR statement makes the result set available for consumption.

In Example 2-29, a parameter marker is used and the result set from the cursor is consumed directly within the procedure.

Example 2-29 Using a parameter marker in a dynamic cursor

```
CREATE OR REPLACE PROCEDURE send_thanks(IN i_date DATE)
 LANGUAGE SQL
 BEGIN
   DECLARE at_end SMALLINT DEFAULT 0;
   DECLARE d_sales_date DATE;
   DECLARE d_sales_person VARCHAR(100);
   DECLARE d_sales_amount DECIMAL(9,2);
   DECLARE d_award_text VARCHAR(200);
   DECLARE d_sql VARCHAR(3000) DEFAULT
      'SELECT sales_date, sales_person, sales
       FROM SALES WHERE sales_date >= ?';                                          1
   DECLARE sales_cursor CURSOR FOR sales_list;                                    2
   DECLARE CONTINUE HANDLER FOR NOT FOUND
      SET at_end = 1;
```

```
   PREPARE sales_list FROM d_sql;                                              3
   OPEN sales_cursor USING i_date;                                             4
   FETCH sales_cursor INTO d_sales_date, d_sales_person, d_sales_amount;       5
   WHILE at_end = 0 DO
     SET d_award_text = 'Congratulations on your sales of $' CONCAT
                        TRIM(CHAR(d_sales_amount))
                        CONCAT ' on ' CONCAT CHAR(d_sales_date);
     CALL send_email(d_sales_person, d_award_text);
     FETCH sales_cursor INTO d_sales_date, d_sales_person, d_sales_amount;     5
   END WHILE;
   CLOSE sales_cursor;                                                         6
END
```

Notes about Example 2-29:

1 The SQL statement contains a parameter marker (question mark (?)).

2 The DECLARE CURSOR associates the prepared statement name sales_list to the cursor name.

3 The PREPARE statement assigns statement name sales_list to the SQL statement string and validates the string syntax. It must precede the OPEN statement.

4 The OPEN statement provides the value (i_date, the input parameter) for the parameter marker and makes the result set available for the subsequent FETCH statements.

5 The FETCH statements retrieve the rows from the result set and make them available to the procedure.

6 The CLOSE statement deactivates the cursor.

2.5.2 PREPARE then EXECUTE

The PREPARE and EXECUTE statements are used in tandem when you work with non-select SQL statements, for example, INSERTs, UPDATEs, DELETEs, CREATEs, and CALLs. A non-select statement does not have an explicit cursor. The non-select SQL statement is typically constructed as a text string and stored in a variable. After the statement is prepared, the EXECUTE statement is used to execute the SQL. This two-step process has the added benefit of allowing parameter markers to be used in the constructed SQL statement. After a statement is prepared, it can be executed multiple times.

PREPARE

The PREPARE statement prepares an SQL select statement for execution. The most common, simple form is shown:

```
PREPARE <statement name> FROM <variable>
```

or

```
PREPARE <statement name> FROM <expression>
```

The *statement name* provides the name that is used to associate the statement to a subsequent EXECUTE. The prepared statement name must match the name that is used on the EXECUTE statement to successfully pair the two together.

During the PREPARE, the statement is validated and all referenced objects, such as tables, are resolved. Any syntax problems with the statement text or unresolved objects cause an error.

For more information about the syntax of the PREPARE statement, see the DB2 for i SQL reference at this website:

https://ibm.biz/Bd42dh

EXECUTE

This statement “runs” the SQL statement that was previously prepared. The syntax is shown:

```
EXECUTE <statement name>
```

If parameter markers were specified in the select statement, the syntax includes the USING clause to provide the values for the parameter markers. That syntax is shown:

```
EXECUTE <statement name> USING <var1>, <var2>, and so on
```

Also, the EXECUTE statement supports the use of DESCRIPTORS. For an explanation and usage examples of descriptors, see Appendix A, “Allocating, describing, and manipulating descriptors” on page 285.

Example 2-30 shows the dynamic nature of building up an SQL statement, which is based on input parameters. It also uses multiple parameter markers.

Example 2-30 PREPARE_THEN_EXECUTE procedure

```
CREATE OR REPLACE PROCEDURE Prepare_Then_Execute (
-- Required input parameters
  IN p_empno CHAR(6),
-- Optional parameters                                                    1
  IN p_salary DECIMAL(9,2) DEFAULT NULL,
  IN p_bonus DECIMAL(9,2) DEFAULT NULL,
  IN p_comm DECIMAL (9,2) DEFAULT NULL)

P1 : BEGIN ATOMIC                                                        2

IF p_salary IS NULL AND
       p_bonus IS NULL AND
       p_comm IS NULL THEN
      RETURN; /* Nothing to do */
END IF ;

PREPARE update_compensation FROM                                         3
   'UPDATE employee SET (salary, bonus, comm) =
     (COALESCE(?,salary),COALESCE(?,bonus),COALESCE(?,comm))             4
    WHERE empno = ?';
EXECUTE update_compensation USING p_salary, p_bonus, p_comm, p_empno;    5

END P1
```

Notes about Example 2-30 on page 27:

1 This procedure takes advantage of default parameters.

2 The block is named P1. It is also defined as ATOMIC. ATOMIC means that all change operations within the block must succeed. If any change operation fails, all changes in the block are rolled back. ATOMIC requires that any changed tables within the block, which is employee in this example, must be journaled. Journaling is automatic when both the library and the table are created with the SQL CREATE SCHEMA and CREATE TABLE statements.

3 The statement string is prepared and assigned the name update_compensation.

4 The COALESCE function (4) within the statement is used to handle potential incoming null values. If a provided parameter marker value (5) is null, the COALESCE uses the second value in its list, which means that the column is set to itself.

5 The prepared statement is executed by using the parameters. If any of the p_salary, p_bonus, or p_comm parameters are null, the COALESCE (4) in the statement effectively ignores them and sets the column to itself.

Example 2-31 shows various ways that the Example 2-30 on page 27 procedure can be called.

Example 2-31 CALL PREPARE_THEN_EXECUTE

```
CALL Prepare_Then_Execute ('650302', p_comm => 10000);   1
CALL Prepare_Then_Execute ('650303', 66000);             2
CALL Prepare_Then_Execute ('650305', p_bonus => 0);      3
```

Notes about Example 2-31:

1 This call updates the commission amount for employee 650302.
2 The call updates the salary amount for employee 650303.
3 This call updates the bonus amount for employee 650305.

2.5.3 EXECUTE IMMEDIATE statement

The EXECUTE IMMEDIATE statement combines the functions that are associated with PREPARE and EXECUTE statements into a single step. It can be used only with non-select SQL statements that do not contain parameter markers. The SQL statement is typically constructed as a text string and stored in a variable or built up as an expression. The syntax is shown:

```
EXECUTE IMMEDIATE <variable>
```

or (IBM i 7.2 or later)

```
EXECUTE IMMEDIATE <expression>
```

EXECUTE IMMEDIATE is a terse way to execute a stand-alone dynamic statement. Any statement that can be run with EXECUTE IMMEDIATE can also be run with a PREPARE and EXECUTE. See Example 2-32.

Example 2-32 EXECUTE IMMEDIATE procedure

```
CREATE OR REPLACE PROCEDURE Execute_Immediate (
IN p_schema VARCHAR(128) DEFAULT CURRENT SCHEMA )
RESULT SETS 1                                                1
```

```
P1: BEGIN
    -- Declare variables
    DECLARE v_sqlstring CLOB (2M);
    -- Declare cursor
    DECLARE employee_cursor CURSOR FOR
      SELECT empno, firstnme CONCAT SPACE(1) CONCAT midinit AS name, hiredate,
          YEAR(CURRENT_DATE) - YEAR(hiredate) AS years_of_service
      FROM qtemp.employee                                                    2
      ORDER BY Years_of_Service;

    -- Build statement with schema explicitly provided
    SET v_sqlstring = 'CREATE OR REPLACE ALIAS qtemp.employee FOR ' CONCAT
                       RTRIM(p_schema) CONCAT '.employee';                   3
    EXECUTE IMMEDIATE v_sqlstring;                                           4

    OPEN employee_cursor;                                                    5
END P1
```

Notes about Example 2-32:

1 This procedure returns a result set of data from the employee table. An ALIAS (3) is used to access data from a table with the same name but in a specified schema (parameter p_schema).

2 The FROM clause of the select refers to the ALIAS name in library QTEMP. The ALIAS is dropped automatically when the session ends.

3 The CREATE ALIAS statement is a good candidate for EXECUTE IMMEDIATE.

4 The local variable that contains the ALIAS statement is prepared and executed with a single EXECUTE IMMEDIATE statement.

5 The OPEN statement activates the cursor employee_cursor to make the result set available to the procedure caller.

2.6 Error handling

Almost every production level piece of code has a type of error handling, or soon will after a painful lesson or two. Why? Because the real world is full of unexpected situations and code must be able to handle problems and react correctly. This situation is the world of error handling.

Error handling is, sadly, one of those overlooked aspects of software development probably because it is normally painful, or at least thought to be painful.

Almost all PSM objects contain SQL statements that can be tested separately, for example, INSERT, UPDATE, DELETE, or CREATE. It is fairly easy to develop the individual pieces and put them together into an executable PSM, which can give the developer a false sense of perfection. Like all programs though, SQL PSM objects can, and do, encounter errors. It is the developer's responsibility to ensure that errors are handled correctly.

Writing and deploying SQL PSM is relatively easy and so is the error handling. PSM employs conditions, handlers, and GET DIAGNOSTICS for processing errors, and SIGNAL, RESIGNAL, and RETURN for indicating errors to the caller.

2.6.1 The basic database error indicators

Before delving into error handling in detail, it is important to know how feedback is handled in the database. Two variables are used by the database management system (DBMS) to return feedback: *SQLCODE* and *SQLSTATE*. These values are implicitly set after the execution of each SQL statement in an SQL PSM object (each statement, that is, except those statements that are used to retrieve the values). These two values indicate the success or failure of the SQL statement.

SQLCODE and *SQLSTATE* provide the same information, but they provide it in different forms. It is not necessary to know all of the possible *SQLSTATE* and *SQLCODE* values. However, it is important to understand their general format and which codes represent the most common set of errors.

SQLSTATE

SQLSTATE is a CHAR(5) value. Each value consists of a two-character class code value, which is followed by a three-character subclass code value, of the form 'ccsss'. *SQLSTATE* is designed so that application programs can test for specific errors or classes of errors.

The *SQLSTATE* definitions are the same for all databases and they are based on the ISO/ANSI standards, providing portability. It is also the basis for the error handlers in PSM and is of most interest here.

Within the *SQLSTATE* definition is a set of reserved areas of class codes (the first two characters) that can be used by applications, allowing an application to communicate within itself and to customize error handling. The definitions are explained:

- *SQLSTATE* classes that begin (first character) with the characters 7 - 9 or I - Z can be defined for use in the application. Any subclass (the last three characters) for these classes can be defined. For example, SQLSTATEs '70000' and '91567' can be defined for use by the application itself.
- *SQLSTATE* classes that begin with the characters 0 - 6 or A - H are *reserved for the database manager*. Within these classes, subclasses that begin with the characters 0 - H are reserved for the database manager. Subclasses that begin with the characters I - Z can be defined.

In general terms, an *SQLSTATE* that starts with (a class of) '00' is successful, '01' is a warning, and '02' is no data.

A complete listing for SQLSTATEs is provided in the SQL Messages and Codes topic at this website:

https://ibm.biz/Bd42Cx

SQLCODE

SQLCODE is an INTEGER value where a certain warning or error is indicated with a certain value. A value of zero (0) means that the statement was successful. A positive value means that a warning was issued when the statement was run. A negative value means that an error occurred.

Important: A common error of many programmers is to code for *SQLCODE* not equal 0, for example:

```
IF (SQLCODE <> 0) THEN <error>
```

This code is problematic because it also causes the <*error*> code to be run for warnings. It is a preferred practice to be more discerning on the *SQLCODE* values and at least to check for a negative return code for errors.

A complete listing for SQLSTATEs and SQLCODEs is provided in the SQL Messages and Codes topic at this website:

https://ibm.biz/Bd42Cx

2.6.2 Conditions and handlers

Various situations arise where you want to handle the situation in PSM and continue, for example:

- Handling duplicate key value errors that might occur during the execution of an INSERT statement
- Handling constraint violations (for example, referential integrity, constraint, primary, or unique) that might occur during the execution of INSERT, UPDATE, or DELETE statements
- Handling “row not found” or “end of the file” conditions when you process a FETCH statement

These situations, and many other situations, can be controlled by the use of two constructs:

- Condition declarations
- Condition handlers

A *condition declaration* provides a way to declare a meaningful condition name for a corresponding *SQLSTATE* value.

The syntax of a condition declaration is shown:

```
DECLARE <name> CONDITION FOR ‘<sqlstate>’
```

Where <*name*> is a meaningful name that is provided to the condition and <*sqlstate*> is the 5-character *SQLSTATE* value.

A *condition handler* is an SQL statement that is executed when an exception or warning condition occurs within the scope of the handler. The actions that are specified in a handler can be any SQL statement, including a compound statement.

The syntax of a condition handler is shown:

```
DECLARE <type> HANDLER FOR <condition>  <statement>
```

The scope of a handler is limited to the compound statement in which it is defined.

Condition handlers are important to a fully functioning PSM object, such as a procedure. In fact, whenever an error occurs, the running SQL PSM object terminates and control is passed to the calling application, unless a *condition handler* is defined.

Three types of condition handlers are available:

- CONTINUE: When this condition is specified, after the SQL statement in the handler is successfully executed, control is returned to the SQL statement that follows the statement that raised the exception.
- EXIT: If EXIT is specified, after the SQL statement in the handler is successfully executed, control is returned to the end of the compound statement that defines the handler.
- UNDO: When UNDO is specified, a rollback operation is performed within the compound statement and 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.

Note: More than one condition can be specified in a handler by separating the conditions with commas. An example is `DECLARE CONTINUE HANDLER FOR CONDITION1, CONDITION2`, and so on.

Three general condition values are provided by the database:

- SQLEXCEPTION: This condition value can be used as a general catchall for any errors, which are *SQLSTATE* values that start with characters other than `00`, `01`, or `02`, that are not otherwise caught by previous handlers in the block.
- SQLWARNING: This condition value is used to catch any warnings, which are *SQLSTATE* values that start with `01`, that are not otherwise caught by previous handlers in the block.
- NOT FOUND: This condition value is used to catch “not found” conditions, which are *SQLSTATE* values that start with `02`, that are not otherwise caught by previous handlers in the block.

Example 2-33 shows a condition and handler example.

Example 2-33 Condition and handler example

```
DECLARE ALREADY_EXISTS INT DEFAULT 0;
DECLARE DUPEKEY CONDITION FOR '23505';                    1
DECLARE CONTINUE HANDLER FOR DUPEKEY SET ALREADY_EXISTS=1  2
```

Notes about Example 2-33:

1 A condition declaration that is called DUPEKEY is defined for a duplicate key error (*SQLSTATE* `23505`).

2 The condition is processed by a condition handler that sets a local variable, *ALREADY_EXISTS*, to 1. The handler is a CONTINUE handler, so processing continues on the next SQL statement after the SQL statement that caused the error.

Example 2-34 shows an EXIT handler example.

Example 2-34 An EXIT handler

```
DECLARE MSGTEXT VARCHAR(120);
DECLARE EXIT HANDLER FOR SQLEXCEPTION                     1
 BEGIN
  GET DIAGNOSTICS CONDITION 1 MSGTEXT = MESSAGE_TEXT;     2
  CALL LOG_IT('ERROR : ‘ CONCAT MSGTEXT);                 3
 END
```

Notes about Example 2-34 on page 32:

1 A general-purpose catchall error handler is defined by using the database-provided general condition SQLEXCEPTION.

2 The handler retrieves the error's message text by using the GET DIAGNOSTICS statement (which is described in the next section) and copies the text into the local variable *MSGTEXT*.

3 The handler then calls a procedure LOG_IT (not shown), which logs the error in a log.

The handler is an EXIT handler (1) so processing continues after the end of the compound statement in which the handler was defined.

Example 2-35 shows a CONTINUE handler.

Example 2-35 A CONTINUE handler

```
DECLARE ALREADY_EXISTS INT;
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505' SET ALREADY_EXISTS=1;     1
```

Notes about Example 2-35:

1 A condition handler captures errors for *SQLSTATE* 23505 (duplicate key error) and sets a local variable, *ALREADY_EXISTS*, to 1. The handler is a CONTINUE handler so processing continues on the next SQL statement after the SQL statement that caused the error.

Although this example codes the *SQLSTATE* directly in the handler, it shows how useful a CONDITION is to document what the handler "catches". Compare this example to Example 2-33 on page 32, which performs the same function but uses a condition that is named DUPEKEY, which helps you understand what is happening.

2.6.3 GET DIAGNOSTICS

The GET DIAGNOSTICS statement is used to get information about an SQL statement after it executes. GET DIAGNOSTICS can be used to gather extensive information about the SQL statement. When GET DIAGNOSTICS is used with handlers, it can be used as the first statement in the handler to determine what happened.

For details about the rich set of information that is available from the GET DIAGNOSTICS statement, see the DB2 for i SQL reference at the following website:

https://ibm.biz/Bd42dh

Two of the most common uses of GET DIAGNOSTICS are described: CONDITION and ROW COUNT.

CONDITION

The GET DIAGNOSTICS CONDITION statement is used to access information that is associated with an error or warning. 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 2-36 writes *SQLSTATE* and the error message text to an errorlog table.

Example 2-36 Using GET DIAGNOSTICS

```
CREATE PROCEDURE GetDiag()
LANGUAGE SQL
```

```
BEGIN
   DECLARE msgtxt CHAR(70);
   DECLARE PrevSQLState CHAR(5);
   DECLARE SQLSTATE CHAR(5);                                          1
   DECLARE CONTINUE HANDLER FOR SQLEXCEPTION                          5
      BEGIN
         SET PrevSQLState=SQLSTATE;                                   2
         GET DIAGNOSTICS EXCEPTION 1 msgtxt=MESSAGE_TEXT;
         INSERT INTO errorlog VALUES(PrevSQLState,msgtxt);            3
      END;

   INSERT INTO unknown values('TEST');                                4
END;
```

Notes about Example 2-36:

1 Every SQL statement implicitly sets the *SQLSTATE* variable. If it is declared, *SQLSTATE* can be referenced in the code.

2 In reality, these two statements are better if they are combined into one GET DIAGNOSTICS, such as in this example:

```
GET DIAGNOSTICS EXCEPTION 1 msgtxt=MESSAGE_TEXT,
PrevSQLState=RETURNED_SQLSTATE;
```

However, for this example, the *SQLSTATE* variable was of interest. Therefore, it was used instead.

3 When the error handler is invoked, it writes an entry to a table that is called errorlog (not shown). In this example, an errorlog row likely reads like the following example:

```
'42704','UNKNOWN in QGPL type *FILE not found.'
```

4 The INSERT statement tries to insert a row into a non-existent table (unknown). This action causes an SQL error, which is handled by the SQLEXCEPTION error handler(5).

Important: GET DIAGNOSTICS does not reset the SQL state. Therefore, GET DIAGNOSTICS needs to generally be the first statement in a condition handler.

ROW_COUNT

ROW_COUNT retrieves the number of rows that are affected by an INSERT, an UPDATE, a DELETE, a REFRESH, or a MERGE statement. Also, it returns the number of rows if the statement was a multi-row fetch.

Example 2-37 shows code that is used to increase the credit of a customer in the city of Rochester by 5%. GET DIAGNOSTICS is used to retrieve the number of updated records in a variable that is called *num_records*.

Example 2-37 Using ROW_COUNT

```
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
```

Notes: The DB2 for i implementation of ROW_COUNT differs slightly from other database implementations because it does not capture the number of rows that are processed by a SELECT statement except in certain multi-row fetches.

2.6.4 SIGNAL and RESIGNAL

Often, a PSM needs its own error messages and error handling for more application-specific error messages about the errors that might occur. PSM also provides the flexibility for an application to have its own set of errors (SQLSTATEs) that it can signal and handle to provide a communication mechanism between modules.

PSM supports two programming constructs to use to handle user-defined errors: SIGNAL and RESIGNAL.

SIGNAL

The SIGNAL statement signals an error or warning condition explicitly. It causes an error or warning to be signaled with the specified *SQLSTATE* with the specified message text.

When a SIGNAL statement is issued, the *SQLCODE* is set based on the *SQLSTATE* value in the following manner:

- If the specified *SQLSTATE* class is either '01' or '02', a warning or not found is signaled and the *SQLCODE* is set to +438.
- Otherwise, an exception is signaled and the *SQLCODE* is set to -438.

If a handler is defined to handle the exception, it handles the SIGNAL statement in the same way as a handler for database-signaled errors.

If no handler is defined to catch the *SQLSTATE* in the SIGNAL statement, the exception is propagated to the caller.

Example 2-38 shows the use of the SIGNAL statement.

Example 2-38 Raising an error by using a SIGNAL statement

```
...
IF COUNTER > 0 OR COUNTER IS NULL OR NOT_FOUND = 1 THEN                    1
   SIGNAL SQLSTATE '75002' SET MESSAGE_TEXT = 'COULD NOT FIND ALERT FILE'; 2
END IF;
```

Notes about Example 2-38 on page 35:

1 Conditionally check to see whether the situation requires signaling an error.
2 SIGNAL error (*SQLSTATE*) 75002 with message text 'COULD NOT FIND ALERT FILE'.

RESIGNAL

The RESIGNAL statement is only allowed inside a condition handler. It is used to resignal the error or warning condition for which the handler was invoked. 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 resignaled to the next higher-level scope. A RESIGNAL statement *with* an operand causes the original condition to be replaced with the new specified condition.

Example 2-39 contains a simple example of SIGNAL and RESIGNAL.

Example 2-39 Raising an error by using SIGNAL and RESIGNAL statements

```
CREATE PROCEDURE G8()
LANGUAGE SQL
BEGIN
   DECLARE not_found_text CHAR(70);
   DECLARE CONTINUE HANDLER FOR SQLSTATE '38TNF'                          1
        BEGIN
           INSERT INTO result(proc,res) values('exec of G8',
             'ErrHandler Fired');
           RESIGNAL SQLSTATE '38TNF' SET MESSAGE_TEXT=not_found_text;  2
        END;

   SET not_found_text = 'Part number not found!';
   INSERT INTO result(proc,res) values('exec of G8','Start');
   SIGNAL SQLSTATE '38TNF';
END;
```

Notes about Example 2-39:

1 A continue handler is defined to handle errors where the *SQLSTATE* is `38TNF`.
2 The RESIGNAL statement is inside the handler. By running inside the handler that caught the original message that was signaled, it resignals the message after first overriding the message text with more specific text, which is '`Part number not found!`'.

Example 2-40 is a more complex, realistic procedure example. The procedure MODSAL 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 rows in EMPLOYEE and DEPARTMENT are related by department number.

The MODSAL SQL procedure implements a business rule that the total compensation of an employee must not exceed the compensation of that employee's manager. The logic of the routine checks that the rule is not compromised. If the rule is compromised, the logic 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(6) and salary change of type DECIMAL(9,2).

Example 2-40 Stored procedure that uses SIGNAL and RESIGNAL

```
create procedure 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 e1 where e1.empno = d.mgrno)
   as mgrcomp
   from employee e, department d
   where e.empno = i_empno and e.workdept = d.deptno;
-- Declare handlers for user-defined error sql states
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;

-- check, if the new compensation within the limit
if  (i_empno <> v_mgrno) and ((i_salary + v_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
```

Notes about Example 2-40 on page 36:

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. Or, the SIGNAL might include the message text and be signaled directly to the invoker.

2 This error handler, which is defined for the '38S01' *sqlstate*, signals the user-defined error condition. The RESIGNAL statement is used to resignal the return *sqlstate* '38S01', but it is more specific about the error message text. After the RESIGNAL is fired, the stored procedure immediately returns the specified error to the caller.

3 The *sqlstate* '02000' is returned by the database if no data exists for the employee number that 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.

2.6.5 RETURN statement

For certain interfaces, such as ODBC and JDBC, the caller of a procedure can code the call to get back a return code. A procedure invocation looks similar to this example:

```
{? = CALL ERROR_HANDLING_BASICS(?,?,?,?,?,?,?)}
```

In these cases, a return code is expected back from the call. PSM supports this type of return code processing by using the RETURN statement. The RETURN statement causes the procedure to end (return) to the caller and pass back the specified integer value.

Example 2-41 calls a generic SQLEXCEPTION handler by using the RETURN statement. The example uses an exit handler to set the return value.

Example 2-41 Stored procedure with RETURN

```
CREATE OR REPLACE PROCEDURE Error_Handling_Basics (
   IN P_EMPNO CHAR(6),
```

```
   IN P_FIRSTNME VARCHAR(12),
   IN P_MIDINIT CHAR(1),
   IN P_LASTNAME VARCHAR(15),
   IN P_WORKDEPT CHAR(3),
   IN P_EDLEVEL SMALLINT,
   IN P_SALARY DECIMAL(9,2)
   )

P1 : BEGIN ATOMIC
   --Error handling variables
   DECLARE EXIT HANDLER FOR SQLEXCEPTION  1
     RETURN -1;  2

   INSERT INTO employee (empno, firstnme, midinit, lastname, workdept, edlevel, salary)
    VALUES (P_EMPNO, P_FIRSTNME, P_MIDINIT, P_LASTNAME, P_WORKDEPT, P_EDLEVEL, P_SALARY);

END P1
```

Notes about Example 2-41:

1 The exit handler is declared to handle any exception.
2 When the exception occurs, a -1 is returned to the calling program and this procedure is terminated.

Although RETURN code processing is supported, RETURN code processing is more likely to miss (not catch) errors when it is compared to handlers because it is the calling application's responsibility to check the return code each time after it calls the procedure.

2.6.6 Direct SQLSTATE and SQLCODE usage

Sometimes, it is handy to use the *SQLSTATE* or *SQLCODE* directly. Although error handling is provided by the conditions, handlers and GET DIAGNOSTICS in PSM mostly preclude the need for direct usage. However, direct usage can facilitate efficiencies sometimes in writing code.

To access these values in an SQL procedure, you must first define them as variables with the exact names, as shown in Example 2-42.

Example 2-42 Declaring SQLCODE and SQLSTATE

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

When an SQL procedure encounters an error, the returned *SQLCODE* is negative, and the first two digits of the *SQLSTATE* differ from '00', '01', and '02'. If SQL encounters a warning while it is processing the SQL statement, but the warning is a valid condition, 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 returned *SQLCODE* is 0, and the *SQLSTATE* is '00000'.

Example 2-43 shows the code for a simple procedure that inserts new employee data in the employee table.

Example 2-43 Error_Handling_Basics procedure

```
CREATE OR REPLACE PROCEDURE Error_Handling_Basics (
   IN P_EMPNO CHAR(6),
```

```
   IN P_FIRSTNME VARCHAR(12),
   IN P_MIDINIT CHAR(1),
   IN P_LASTNAME VARCHAR(15),
   IN P_WORKDEPT CHAR(3),
   IN P_EDLEVEL SMALLINT,
   IN P_SALARY DECIMAL(9,2)
   )

P1 : BEGIN ATOMIC
   --Error handling variables
   DECLARE SQLCODE INTEGER DEFAULT 0;                                              1
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000';                                       1
   INSERT INTO employee (empno, firstnme, midinit, lastname, workdept, edlevel, salary)
    VALUES (P_EMPNO, P_FIRSTNME, P_MIDINIT, P_LASTNAME, P_WORKDEPT, P_EDLEVEL, P_SALARY) ;

   If SQLCODE < 0 THEN                                                             2
      RETURN -1;
   END IF;


END P1
```

Notes about Example 2-43 on page 38:

1 *SQLCODE* and *SQLSTATE* are explicitly declared in the block so that they can be referenced in the code.

2 *SQLCODE* is checked for an error (negative value).

2.6.7 Error handling in nested compound statements

When nested compound statements are used, each compound statement has its own scope for variable definitions, its condition definitions, and its error handlers. When an error is signaled, the handler at the closest or innermost block where the error occurs catches the error.

Example 2-44 illustrates how handlers can be scoped at different levels.

Example 2-44 Error handlers in nested compound statements

```
CREATE PROCEDURE SHOW_ERROR_HANDLERS()
LANGUAGE SQL
BEGIN -- outer compound statement
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
   DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H02'                    1
      BEGIN
         DECLARE TEXT VARCHAR(70);
         SET TEXT = '38H02 MANAGED BY OUTER ERROR HANDLER' ;
         RESIGNAL SQLSTATE VALUE '38HE0'
         SET MESSAGE_TEXT = TEXT;
      END;

   BEGIN -- inner compound statement
      DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H02'                 2
         RESIGNAL SQLSTATE VALUE '38HI3'
         SET MESSAGE_TEXT = '38H02 MANAGED BY INNER ERROR HANDLER';

      SIGNAL SQLSTATE VALUE '38H02'                                   3
         SET MESSAGE_TEXT = 'ERROR SIGNALED IS CAUGHT BY INNER COMPOUND HANDLER';
```

```
   END; -- end inner compound statement

   SIGNAL SQLSTATE VALUE '38H02'                                          4
      SET MESSAGE_TEXT = 'ERROR SIGNALED IS CAUGHT BY OUTER COMPOUND HANDLER';
END;
```

Notes about Example 2-44:

1 The outer compound statement's handler for 38H02.

2 The inner compound statement's handler for 38H02.

3 This error will be handled by the inner compound's handler 2 because it is the closest handler in the compound nesting.

4 This error will be handled by the outer compound's handler 1 because it is the closest (only) handler in the compound nesting.

2.7 Transaction management in procedures

Transaction management is introduced. For a more involved description, see 3.4, "Transactions" on page 64 about isolation levels.

When you run a moderately complex procedure, it can be common to update several rows in one table or even update a number of rows in different tables. What happens when a failure occurs and rows that were supposed to be updated were not? Does it matter?

Transaction management is the concept of treating one or more data change as one unit, which is known as a *transaction*. The most common example of a transaction is a money transfer between two accounts. The process that is involved in a transfer is the subtraction of money from the source account and the addition of that same amount of money to the target account. Failing the process in the middle, when the source account is debited but the target account is not credited, is an extremely bad situation. On this type of failure, the correct answer is to undo or "roll back" the source account debit so that the accounts are back where they started. Transaction management plays a critical role in ensuring atomicity on these types of multiple step operations.

In more involved cases, an overall transaction is the collection of smaller subtransactions that can themselves complete or roll back. Therefore, the final result is that the larger transaction completes or rolls back. Making travel plans is a good example.

Consider a trip that involves a plane trip, a car, and a hotel. The overall trip is the major transaction. However, several smaller transactions are involved: booking a flight, reserving a car, and reserving a hotel room. Booking the flight is the first part of the transaction. Next, a hotel reservation is made. However, part of the way through the hotel reservation, a failure occurs so that the hotel room was chosen but then it was determined that it was the wrong room or even the wrong hotel. So, the hotel reservation needs to be rolled back, but only the hotel, the flight needs to stay reserved. Now, the correct hotel and room are reserved, and the car needs to be reserved. Again, various mishaps occur where the car reservation needs to be rolled back without losing either the plane reservation or the hotel reservation. Finally, all aspects of the trip are complete and the trip can be completed (committed).

Transaction management is an important but too frequently overlooked aspect of data management processing. It is important to consider transaction management when you design and write PSM. With a little planning, SQL and PSM make it fairly straightforward to accomplish correct transaction management.

As a definition, operations within a transaction are participating in a *logical unit of work* (LUW).

Notes: In the remainder of the description of transaction management, we assume that any necessary database files are journaled, which is a prerequisite to use transaction management (commitment control).

As a principle of design, procedures, triggers, and functions must be built independently from the caller or firing process order. Designers and programmers must carefully plan for an error-handling strategy that does not impede the caller or dictate a certain order of process work.

ATOMIC

The simplest form of transaction management is to define a compound statement as ATOMIC. By starting a compound statement with BEGIN ATOMIC, every data change statement within that compound statement is treated as one transaction. If any SQL statement in the compound statement fails, all operations are rolled back.

SAVEPOINT

The SAVEPOINT statement is used to establish a milestone in an LUW. A name for the savepoint is important because the name uniquely identifies the savepoint.

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 2-1.

Table 2-1 Events that initiate and terminate savepoint levels

Savepoint level is initiated when this event occurs	Savepoint level terminates when this event occurs
A new unit of work is started.	A COMMIT or ROLLBACK is issued.
A trigger is invoked.	The trigger completes.
A user-defined function (UDF) is invoked.	The UDF 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 included in an ATOMIC compound SQL statement.	An END is included in an ATOMIC compound SQL statement.

When a savepoint level ends, all active savepoints that were 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 they are subject to any savepoint-related statements that are issued within the parent savepoint level.

COMMIT

The COMMIT statement ends a unit of work and commits the database changes that were made by that unit of work.

ROLLBACK and ROLLBACK TO SAVEPOINT

The ROLLBACK statement is used to back out database changes to the beginning of the current transaction or the specified savepoint. When the ROLLBACK TO SAVEPOINT is used, the database is backed out to the last savepoint. Example 2-45 shows a specific example.

Example 2-45 SAVEPOINTs

```
INSERT INTO MYLIB.TRACE_TABLE VALUES ('FIRST INSERTED ROW');          1
SAVEPOINT savepoint_A;
INSERT INTO MYLIB.TRACE_TABLE VALUES ('SECOND INSERTED ROW');         2
SAVEPOINT savepoint_B;                                                3
INSERT INTO MYLIB.TRACE_TABLE VALUES ('THIRD INSERTED ROW');          4
SAVEPOINT savepoint_C;
INSERT INTO MYLIB.TRACE_TABLE VALUES ('FOURTH INSERTED ROW');         5
ROLLBACK TO SAVEPOINT savepoint_B;                                    6
```

Notes about Example 2-45:

- The first (1), second (2), third (4), and fourth (5) rows are inserted in table TRACE_TABLE.
- However, when the ROLLBACK TO SAVEPOINT savepoint_B (6) occurs, the third (4) and fourth (5) rows will be rolled back.
- Savepoint_B (6) will not be released because the ROLLBACK targeted it only. The savepoint 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.

The *isolation level* that is used during the execution of SQL statements determines the degree to which the session is isolated from other, concurrently executing sessions. The isolation level is specified as an attribute of an SQL procedure, and it applies to the sessions that use the SQL procedure.

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. For the SELECT, SELECT INTO, INSERT, UPDATE, DELETE, and DECLARE CURSOR statements, a certain isolation level can be specified at the statement level. The isolation level is in effect only for the execution of the statement that contains the isolation clause. Example 2-46 show an example of the SET TRANSACTION statement.

Example 2-46 SET TRANSACTION ISOLATION LEVEL

```
SET TRANSACTION ISOLATION LEVEL UR
```

2.7.1 Transaction management example

Example 2-47 and Example 2-48 on page 44 perform the same work from a transaction management perspective. Example 2-47 uses explicit ROLLBACK and COMMIT to accomplish the action. Example 2-48 on page 44 uses the ATOMIC compound statement option.

Important: If ATOMIC is specified, the COMMIT and ROLLBACK statements must *not* be specified in the compound statement.

If UNDO is specified in the declaration of a handler in a compound statement, ATOMIC *must* be specified in the BEGIN clause.

Example 2-47 illustrates the use of commitment control statements within a compound SQL.

Example 2-47 Commitment control statements within a compound SQL

```
CREATE PROCEDURE CREDITP
     (IN    i_percentage_increase DECIMAL(3,2),
      INOUT o_number_records_processed INTEGER)
      LANGUAGE SQL
 BEGIN     1
    DECLARE proc_cusnbr CHAR(5);
    DECLARE proc_cuscrd DECIMAL(11,2);
    DECLARE numrec      INTEGER;
    DECLARE at_end      INTEGER 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_percentage_increase);
          UPDATE customer
              SET cuscrd = proc_cuscrd
                WHERE CURRENT OF c1;
          SET numrec = numrec + 1;
       FETCH c1 INTO proc_cusnbr, proc_cuscrd;
    END WHILE;
    SET o_number_records_processed = numrec;
    CLOSE c1;
    COMMIT;  3
 END
```

Notes about Example 2-47 on page 43:

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 procedure closes the cursor, the procedure COMMITs all of the changes to the file.

Example 2-48 presents an equivalent procedure to Example 2-47 on page 43, but it uses BEGIN ATOMIC instead. The sqlexception error handler that causes a ROLLBACK and the commit statement in Example 2-47 on page 43 are not present in Example 2-48 because ATOMIC indicates that the database handles this situation automatically. If an unhandled error occurs within the block, the database will automatically ROLLBACK any changes that occurred in the block. If the block is successful, the database will automatically COMMIT all changes that occurred in the block.

Example 2-48 shows commitment control statements that use BEGIN ATOMIC.

Example 2-48 Commitment control statements that use BEGIN ATOMIC

```
CREATE PROCEDURE CREDITP_ATOMIC
     (IN    i_percentage_increase DECIMAL(3,2),
      INOUT o_number_records_processed INTEGER)
      LANGUAGE SQL
 BEGIN ATOMIC    1
    DECLARE proc_cusnbr CHAR(5);
    DECLARE proc_cuscrd DECIMAL(11,2);
    DECLARE numrec      INTEGER;
    DECLARE at_end      INTEGER DEFAULT 0;
    DECLARE not_found
           CONDITION FOR '02000';
    DECLARE c1 CURSOR FOR
          SELECT cusnbr, cuscrd
            FROM  ordapplib.customer;
    DECLARE CONTINUE HANDLER FOR not_found           2
           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_percentage_increase);
          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_number_records_processed = numrec;
    CLOSE c1;
 END
```

Notes about Example 2-48 on page 44:

1 The BEGIN clause uses the **ATOMIC** keyword to make the compound statement a transaction.

2 Important: If the not_found condition is true, it will not cause a rollback.

3

SQL fundamentals

This chapter describes Structured Query Language (SQL) fundamentals that are referenced throughout this book.

This chapter includes the following topics:

- SQL concepts
- Common information for SQL routines and triggers
- DB2 sample database
- Transactions
- DB2 for i catalog views

3.1 SQL concepts

This section covers the SQL concepts are used and referenced throughout this book:

- Schemas and libraries
- Unqualified object names
- SQL PATH
- Global variables

For more information, see the DB2 for i SQL reference at this website:

https://ibm.biz/Bd42dh

3.1.1 Schemas and libraries

Objects in a relational database are organized into sets, which are called *schemas*. On IBM DB2 for i, the schema concept is based on the traditional IBM i approach of using libraries to organize objects. The terms schema and library are almost interchangeable because only a few differences exist. When a schema or collection is created by using CREATE SCHEMA, the following objects are created:

- An IBM i library with the same name as the schema. When the name is longer than 10 characters, the schema uses the 10-character name that is specified on the FOR clause or a name that is generated by system.
- The new library contains a journal that is named QSQJRN and a journal receiver that is named QSQJRN0001. By default, DB2 for i uses these journals to record any changes to the tables that were created in the schema.
- The new library contains a set of catalog views whose names start with the three letters SYS.

This book might refer to either schemas or libraries and it will highlight any situation in which their behaviors differ.

3.1.2 Unqualified object names

DB2 for i supports two methods for resolving unqualified references to tables and views in SQL statements. Example 3-1 shows a simple example.

Example 3-1 Simple SELECT statement

```
SELECT * FROM table1;
```

To run this statement, DB2 for i must know the schemas to use to locate table1. This schema is called the *default schema*. The following rules apply:

- For static SQL (SQL statements that are known at compile time), the default schema is specified:
  - The **DFTRDBCOL** parameter on the SET OPTION statement can be used to specify the name of the default schema. (The high-level language (HLL) precompilers also support this usage on their command interfaces.)
  - In all other cases, the default schema is based on the naming convention:
    - In SQL naming, an unqualified table reference is resolved by looking in a schema that has the same name as the current user. If JIMD is signed on, DB2 for i looks for table1 in a schema that is named JIMD. If table1is not in a schema that is named JIMD, the query fails, which is consistent with SQL standards and more portable across database implementations.
    - In system naming, an unqualified table reference is resolved by using the library list (*LIBL). This approach is not standard SQL but it is convenient for IBM i developers who are familiar with and often depend on this behavior.
- For dynamic SQL statements (statement that is constructed at run time), the default schema is determined:
  - If the default schema was set explicitly, the setting takes precedence. The default schema can be specified in many ways, depending on the interface. However, for SQL routines and triggers, only two methods apply:
    - The SET OPTION statement can specify the **DFTRDBCOL** keyword, which specifies the default schema for static SQL. Also, the specification of DYNDFTCOL(*YES) results in the use of the same default schema for dynamic SQL.
    - The SQL statement SET SCHEMA can be used to specify the default schema. This statement is covered next.
  - If a default schema is not explicitly specified, the following rules apply:
    - For SQL naming, the default schema is a schema with the same name as the current user (as described previously for static SQL).
    - For system naming, unqualified references are resolved by using the job library list (*LIBL).

The default schema is used to qualify table and view objects. It is also used for unqualified references to the following objects:

- Alias
- Constraint
- External program
- Index
- Mask
- Nodegroup
- Package
- Permission
- Sequence
- Trigger
- XML schema repository (XSR) objects

Figure 3-1 shows the syntax for the SET SCHEMA SQL statement.

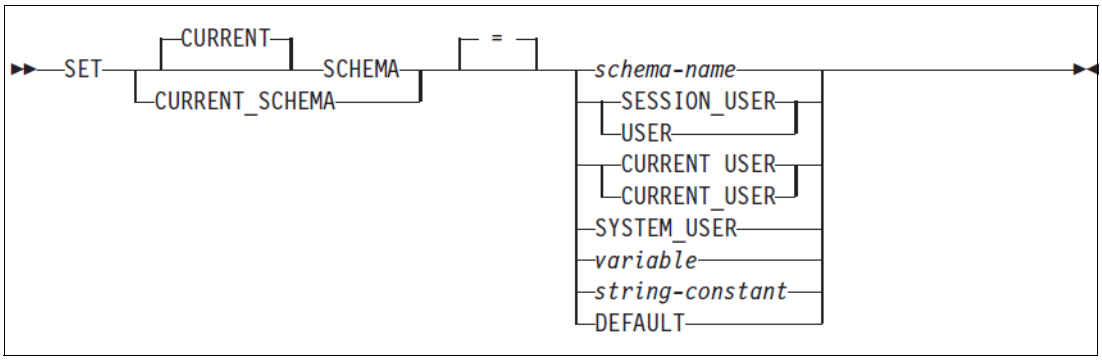

Figure 3-1 SET SCHEMA syntax

The SET SCHEMA SQL statement is an executable statement that can be embedded in programs or issued interactively. It can also be prepared. The options are described:

- The schema name explicitly identifies a schema.
- USER or SESSION_USER is a special register that contains the runtime user profile that determines object authorities for the current session.
- CURRENT USER or CURRENT_USER is similar to USER but has an important difference. It also reflects adopted authority in programs or SQL routines. It was added in IBM i 7.2.
- SYSTEM_USER is the user profile that initiated the session. Many jobs, including the QZDASOINIT pre-started jobs, initially connect to the server with a default user profile and then change to use another user profile. SYSTEM_USER reports the default user, which is typically QUSER for a QZDASOINIT job. It has a data type of VARCHAR(128).
- A variable can be specified by using a program variable in static SQL or as a parameter marker string constant in dynamic SQL.
- A *string constant* is a literal value that must contain at least one schema name. Schema names are case-sensitive. They must be specified in the correct uppercase or lowercase.
- DEFAULT resets the environment to its default behavior, which is USER for SQL naming and *LIBL for system naming.

Note: SET SCHEMA does not check whether the schema exists. Whether the schema exists will be determined and reported later. Also, SET SCHEMA affects dynamic SQL only. It does not affect static SQL. The resolution of unqualified references to procedures, functions, global variables, and user-defined types is managed by the SQL PATH, which is covered next.

3.1.3 SQL PATH

The *SQL path* is an ordered list of schema names and it is used by the database to resolve the schema for unqualified references to types, functions, variables, and procedure names. For IBM i developers, it works in a similar manner to a library list for those objects.

The default path differs based on the naming convention that is used:

- In SQL naming, the path defaults to a list of schemas that are defined by the database and followed by the schema that has the same name as the current user. For example, if JIMD is signed on, the database searches for an unqualified reference in this list of schemas:

  `"QSYS","QSYS2","SYSPROC","SYSIBMADM","JIMD"`

  This approach is consistent with SQL standards and more portable across database implementations. The values appear with double quotation marks (" "). These double quotation marks are optional, and they are only required when the name of the schema contains mixed case or special characters.
- In system naming, the path defaults to *LIBL, which means that the library list is used to resolve unqualified references. *LIBL is nonstandard, and it is not recognized by other SQL implementations.

The path can be changed by using the SET PATH statement that is shown in Figure 3-2.

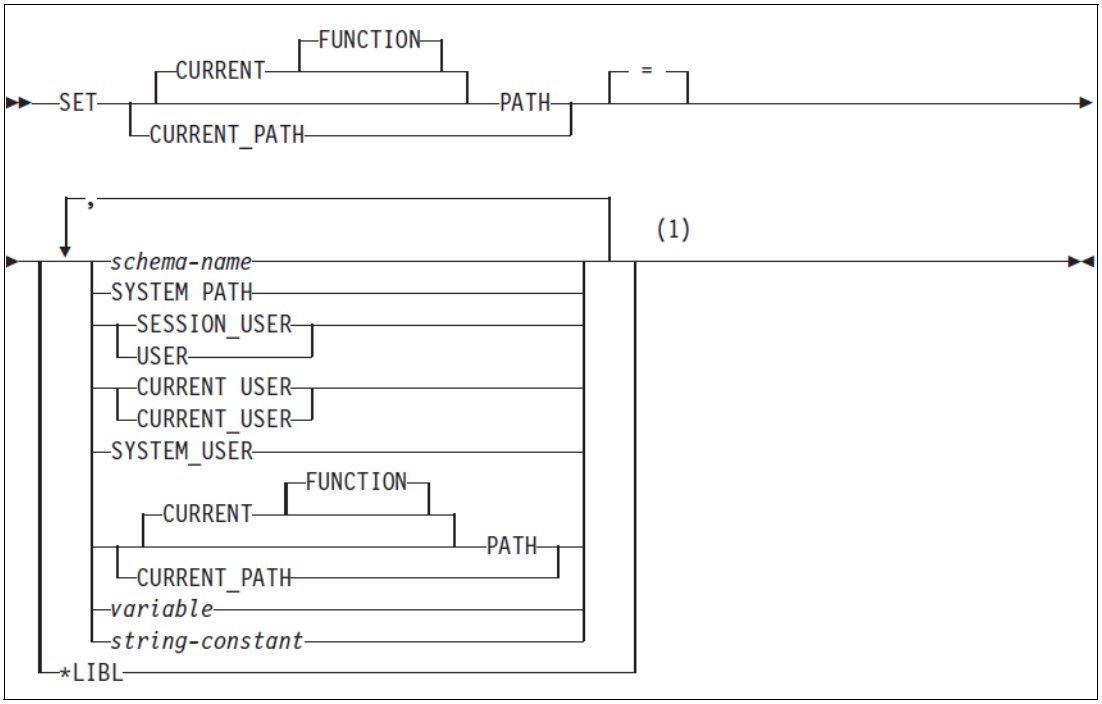

Figure 3-2 SET PATH syntax

Many of these same options, such as USER, are defined the same way for SET SCHEMA, so only those options that are new or different are covered in this list:

- SYSTEM PATH indicates the list of default schemas that are defined by the database. For DB2 for i, this option is the same as specifying "QSYS","QSYS2","SYSPROC","SYSIBMADM".
- CURRENT PATH represents the value of the CURRENT PATH special register before you run this statement. You cannot reference CURRENT PATH if the current path is *LIBL.
- A string or variable must contain a list of one or more schema names that are separated by commas. These names are case-sensitive.

Example 3-2 shows examples of the SET PATH statement.

Example 3-2 SET PATH examples

```
SET PATH JIMD;                                    1

SET PATH CURRENT PATH, JIMD;                      2

SET PATH SYSTEM PATH, JIMD;                       3

SET PATH DANCRUIK, SIMONA, JIMD;                  4
```

Notes about Example 3-2:

1 The path contains only one schema, JIMD.

2 This path contains the schemas that were in this path before and appends the schema JIMD to the end of the list.

3 The path contains the standard system list of QSYS, QSYS2, SYSPROC, and SYSIBMADM, followed by the schema JIMD.

4 The path contains the schemas DANCRUIK, SIMONA, and JIMD. They are followed by QSYS, QSYS2, SYSPROC, and SYSIBMADM, which are implicitly added to the end of the path when they are not explicitly specified.

The SQL path is used to resolve the schema name for unqualified type names (built-in types, distinct types, and array types), function names, global variable names, and procedure names. Based on their signatures, procedures and functions require additional considerations, which will be described later.

Note: The path is not used when the SQL object is the main object of an ALTER, CREATE, DROP, COMMENT, LABEL, GRANT, or REVOKE statement. In these cases, the unqualified object names follow the same rules for unqualified tables that were described previously. When the object is not schema-qualified on a CREATE statement, the schema where it will be created depends on whether it is a procedure, function, or trigger.

3.1.4 Global variables

Global variables provide a powerful technique for sharing values within a session. On DB2 for i, the session corresponds to a connection or a job. Although the definition of a variable is shared across sessions, the values that the variables contain are always scoped to the job/session. They can be useful in SQL routines and triggers for effectively passing values without defining them as parameters.

Figure 3-3 shows the CREATE VARIABLE syntax.

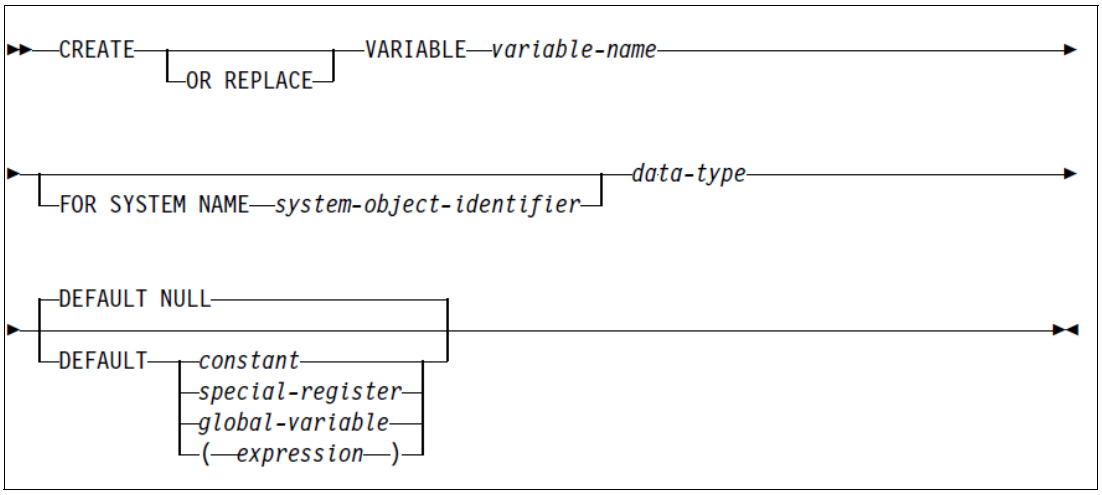

Figure 3-3 CREATE VARIABLE syntax

Figure 3-3 shows the syntax to create a global variable:

- The name of the global variable can be up to 128 characters. We recommend that you use a naming convention to differentiate them from column names or variable names. DB2 for i will generate a service program to manage the behavior of the value. FOR SYSTEM NAME allows the specification of a 10-character value rather than allowing DB2 for i to generate the service program name when the SQL name is longer than 10 characters.
- The data type can be any of the data types that are supported in an SQL table, including INT, DECIMAL, VARCHAR, and so on. It can also be a user-defined type, which was previously created by using the CREATE TYPE statement.
- The default value is not instantiated until the first time that the variable is referenced within the session, and it must be explicitly changed later. That is, they are not dynamically maintained and they will not reflect any changes in the underlying value that is used for their initialization. The default value can be set (and defaults to) the NULL value. It can also be any of these values:
  - A literal value, such as 'ABC' or 123.
  - A special register, such as USER or CURRENT_TIMESTAMP.
  - The value that is contained in another global variable.
  - An expression. The expression can be a simple combination of literals, such as (1+2) but it can also be a select statement.

Example 3-3 shows sample CREATE statements for global variables.

Example 3-3 CREATE VARIABLE examples

```
CREATE VARIABLE gv_status_flag CHAR(1) DEFAULT NULL;                    1

CREATE VARIABLE gv_starttime TIMESTAMP DEFAULT(CURRENT_TIMESTAMP);      2

CREATE VARIABLE gv_employee_count INT
   DEFAULT(SELECT COUNT(*) FROM employee);                              3
```

Notes about Example 3-3 on page 53:

1 This example creates a variable that is named *gv_status_flag*, which consists of a single character. When it is first referenced in the session, it has a NULL value.

2 This example creates a variable that is named *gv_starttime* with a data type of TIMESTAMP. When it is first referenced in the session, it contains the value of the CURRENT_TIMESTAMP special register. It will retain this value until it is explicitly changed. References to gv_starttime in other sessions have their own and potentially different values.

3 This example creates a variable that is named *gv_employee_count* with the data type of INT. When it is first referenced in a session, DB2 for i runs the SELECT statement and stores the resulting count of employees in this variable. Because they are not maintained, the variable will not change if the number of rows in the employee table changes.

Global variables can be modified by using SET, VALUES INTO, or SELECT INTO statements, as shown in Example 3-4.

Example 3-4 Changing global variable values

```
SET gv_status_flag = 'A';                                                    1

VALUES CURRENT_TIMESTAMP-2 HOURS INTO gv_starttime;                          2

SELECT COUNT(*) INTO gv_employee_count FROM employee WHERE workdept='D11';3

SET gv_employee_count = DEFAULT;                                             4
```

Notes about Example 3-4:

1 The *gv_status_flag* is assigned a value of 'A'.

2 The *gv_starttime* is assigned a value that is equal to the CURRENT_TIMESTAMP minus 2 hours.

3 The *gv_employee_count* is assigned a value that is based on a SELECT that returns the count of employees that are assigned to the department 'D11'. The schema for the employee table is determined when the variable is created and not when the variable is referenced.

4 The *gv_employee_count* is assigned the DEFAULT value that was specified when the global variable was created. Again, the schema for the table is determined when the variable is created.

Global variables can be used in many places that allow literal values. They can be referenced in many places:

- In the SELECT list of a query
- In the WHERE clause of a query
- In the SELECT list or the WHERE clause of a view definition
- As variables in the SQL programming language, which will be covered later in this book

Example 3-5 shows an example of a “filtered view” in which a global variable is used to define the subset of the underlying table that is returned by a view.

Example 3-5 The use of a global variable to filter data in a view

```
CREATE VARIABLE gv_workdept CHAR(3) DEFAULT NULL;                    1

SET gv_workdept = 'E21';                                             2

CREATE VIEW filtered_employee AS
   SELECT * FROM employee WHERE workdept=gv_workdept;                3
```

Notes about Example 3-5:

1 The *gv_workdept* is created as a character(3) with a default value of `NULL`.

2 The *gv_workdept* is assigned a value of '`E21`'.

3 The filtered_view shows only those employees that work in the department that is identified by *gv_workdept*. If *gv_workdept* is not set, the view does not show any employees.

3.2 Common information for SQL routines and triggers

Common information across procedures, triggers, and functions is described. For more information, see the DB2 for i SQL reference:

`https://ibm.biz/Bd42dh`

A *Persistent Stored Module (PSM)* is the architectural term for an SQL object that contains programs that are written in SQL. This book uses the terms *PSM* or *SQL programming language*.

An *SQL routine* is the general term for an executable SQL object that includes both procedures and functions. Triggers are not included under the definition of routines because they are activated by the database rather than directly called or invoked. This book is focused on routines and triggers that are implemented in the SQL programming language. It does not cover external routines or triggers that are written in other HLLs, such as RPG.

3.2.1 Routine and trigger creation process

When you execute an SQL CREATE statement for a routine or trigger, DB2 for i takes your SQL programming language statements and rewrites them as Integrated Language Environment (ILE) C. Many control statements, such as the declaration of variables or simple assignment statements, can be converted directly to ILE C syntax. SQL statements that are contained within your routine or trigger body are converted to embedded SQL.

Note: This process is largely transparent to you, and no dependency exists on the installation of a C compiler. At most, you might notice that the database creates a temporary source file that is named QSQLSRC with a few other objects in the QTEMP library.

The object that is generated depends on whether it is a procedure, function, or trigger in addition to the options that are specified on the corresponding CREATE statement:

- Procedures are created either as *PGM or *SRVPGM objects, depending on the PROGRAM TYPE that is specified on the CREATE statement. The specification of PROGRAM TYPE MAIN results in the creation of a *PGM object, which is the default behavior. The specification of PROGRAM TYPE SUB results in the creation of a *SRVPGM object, which can provide a modest performance improvement because calls to service programs are faster than calls to programs.
- Triggers are always created as *PGM objects.
- Functions are always created as *SRVPGM objects.

These programs and service programs inherit the behaviors of embedded SQL. Statements that are contained in the routine or trigger body effectively become static SQL. Any statements that are constructed at execution time become dynamic SQL.

The DB2 Query Manager and SQL Development Kit for IBM i licensed product is not required to create SQL routines or triggers.

3.2.2 IBM i names for generated SQL objects

The names of SQL procedures, triggers, and functions can be up to 128 characters. By default, DB2 for i generates a 10-character short name that is recognized by IBM i but it is not a useful name. It takes the first five characters of the long name and then adds a 5-digit number to ensure that the short name is unique within the schema.

We recommend that you control the short name of your generated objects rather than letting the system generate them:

- The CREATE statements for procedures and functions allow the specification of a SPECIFIC name, which can be up to 128 characters and can also be schema-qualified. This name is often used to provide a name, which is 10 or fewer characters, that is used for the generated program or service program. If the name is longer than 10 characters, it has a generated specific name that follows the rules that were described previously.
- The CREATE TRIGGER statement has a PROGRAM NAME clause that can be used to control the name of the generated program.

3.2.3 CREATE OR REPLACE

Procedures, triggers, and functions all support the optional OR REPLACE clause, which gives DB2 for i permission to delete an existing routine or trigger of the same name. For developers who are familiar with the HLL compilers today, this function is equivalent to specifying REPLACE(*YES). In SQL, this function corresponds to first using DROP to delete the existing routine or trigger followed by the CREATE statement. Procedures, triggers, and functions all behave in the following manner:

- Any existing comment or label on the object is discarded.
- The authorized users will be maintained but the object owner might change, depending on the environment and the attributes that are specified when it is re-created.
- Current journal auditing is preserved.

Procedures, triggers, and functions have specific behaviors that will be explained in the corresponding sections of this book.

3.2.4 Shared attributes

This section describes syntax and attributes that are shared across routines and with triggers:

- SET OPTION
- MODIFIES or READS SQL DATA
- CONCURRENT ACCESS RESOLUTION
- SECURED
- WRAPPED

Table 3-1 shows the attributes that apply to SQL routines and triggers.

Table 3-1 SQL routine and trigger attributes

Function	Procedures	Triggers	Functions
SET OPTION	Yes	Yes	Yes
MODIFIES or READS SQL DATA	Yes	No	Yes
CONCURRENT ACCESS RESOLUTION	Yes	Yes	Yes
SECURED[a]	No	Yes	Yes
WRAPPED	Yes	Yes[b]	Yes

a. SECURED support was added in IBM i 7.2.
b. WRAPPED support for triggers was added in IBM i 7.2.

SET OPTION

Both routine types and triggers support the SET OPTION clause, which provides an interface to pass options to the C precompiler during the process of generating the underlying program or service program.

Figure 3-4 shows the SET OPTION clause syntax.

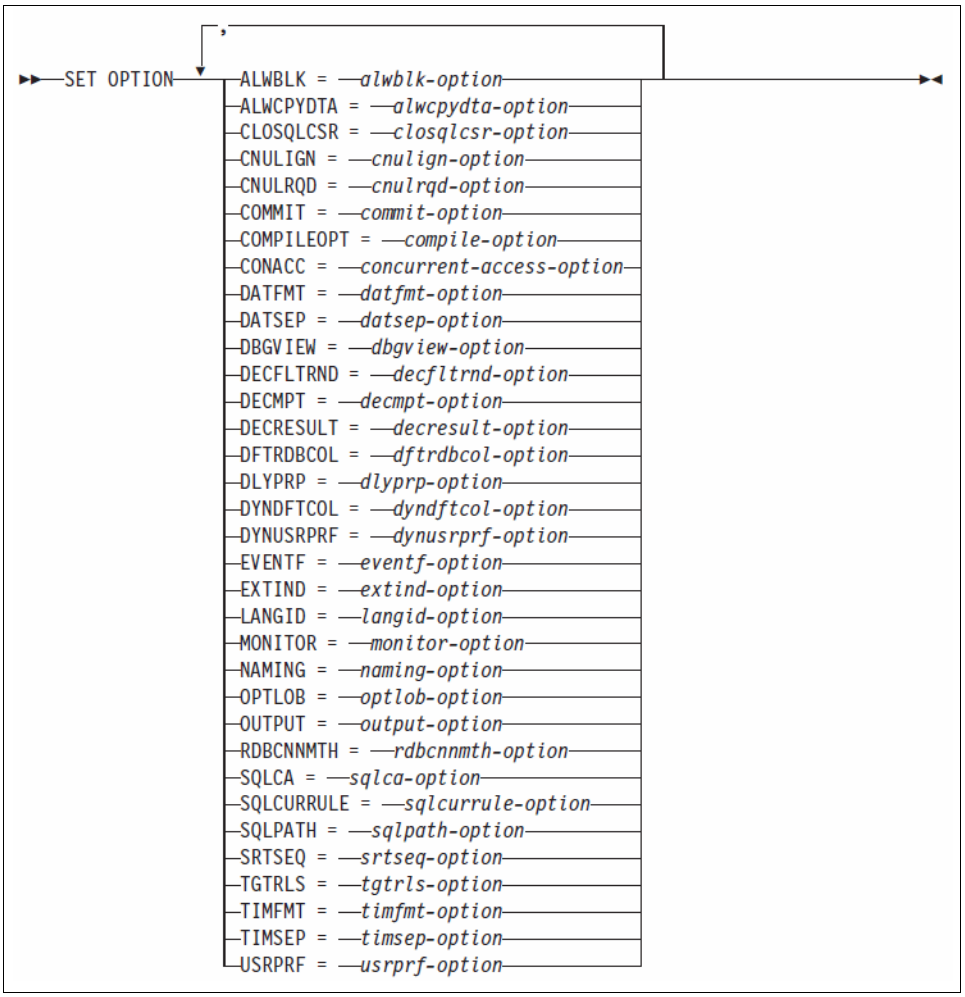

Figure 3-4 SET OPTION clause syntax

Figure 3-4 shows the syntax for the SET OPTION clause. These options are all documented in the DB2 for i SQL reference, which is available at this website:

https://ibm.biz/Bd42dh

The DBGVIEW and USRPRF options are most typically used for SQL routines and triggers.

Figure 3-5 shows the SET OPTION DBGVIEW syntax.

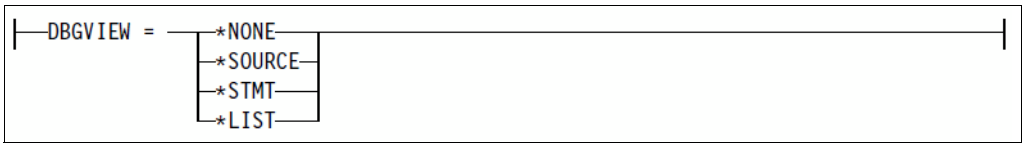

Figure 3-5 SET OPTION DBGVIEW syntax

The DBGVIEW clause is specific to procedures, triggers, and functions. It specifies whether the generated program or service program can be debugged and the type of debug information that will be provided by the compiler.

The options are defined:

- *NONE specifies that a debug view is not generated.
- *SOURCE specifies that the resulting program or service program can be debugged by using the SQL statement source. If *SOURCE is specified, the modified source is stored in the source file QSQDSRC in the same schema as the created procedure, trigger, or function.
- *STMT specifies that the resulting program or service program can be debugged by using program statement numbers and symbolic identifiers.
- *LIST specifies that the listing view for the program or service program needs to be generated.

Note: If DEBUG MODE is specified in a CREATE PROCEDURE or ALTER PROCEDURE STATEMENT, a DBGVIEW option in the SET OPTION statement must not be specified.

If DEBUG MODE is not specified, but a DBGVIEW option in the SET OPTION statement is specified, the procedure cannot be debugged by the Unified Debugger, but it can be debugged by the system debug facilities. If you do not specify DEBUG MODE or a DBGVIEW option, the debug mode that is used is from the CURRENT DEBUG MODE special register.

Additional information is provided in the 7.2, “Debug SQL routines and triggers” on page 203.

Figure 3-6 shows the SET OPTION USRPRF syntax.

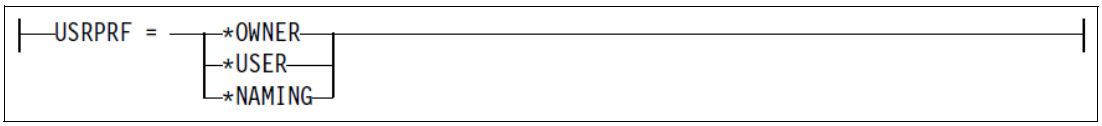

Figure 3-6 SET OPTION USRPRF syntax

Figure 3-6 shows the syntax for the USRPRF clause and specifies the user profile that will be used when the program or service program is run, including the authority that the program object has for each object in static SQL statements. The profile of either the owner or the user is used to control the objects that can be used by the program or service program. The options are defined:

- *NAMING indicates that the user profile is determined by the naming convention. If the naming convention is *SQL, USRPRF(*OWNER) is used. If the naming convention is *SYS, USRPRF(*USER) is used.
- *USER indicates that the profile of the user that runs the program or service program is used.
- *OWNER indicates that the user profiles of both the owner and the user are used when the program or service program is run.

Figure 3-7 shows the SET OPTION DYNUSRPRF syntax.

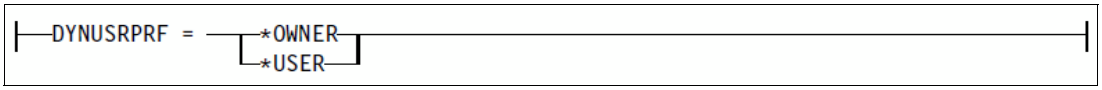

Figure 3-7 SET OPTION DYNUSRPRF syntax

Figure 3-7 on page 59 shows the syntax for the DYNUSRPRF clause, which specifies the user profile that will be used when the program or service program executes dynamic SQL statements:

- *USER indicates that dynamic SQL statements are run under the user profile of the job.
- *OWNER indicates that dynamic SQL statements are run under the user profile of the program's owner.

Other clauses serve the same purpose that they serve in any HLL precompiler. You can use clauses, such as DFTRDBCOL, DYNDFTCOL, and SQLPATH, to control how unresolved object references are resolved.

SET OPTION has certain behaviors that are specific to procedures, triggers, and functions, which are covered later.

MODIFIES or READS SQL DATA

The MODIFIES or READS SQL DATA clause is supported for procedures and functions.

This clause is used to verify the types of SQL statements that can be executed within the routine. This option is ignored for parameter default expressions. The syntax is not included here because it differs from statement to statement. It can have the following values:

- NO SQL: This value specifies that the routine can execute only SQL statements that are classified as NO SQL. This value is only valid for external routines.
- CONTAINS SQL: The function cannot execute any SQL statements that read or modify data. This value corresponds to using SQL only as a programming language without any actual data access.
- READS SQL DATA: The routine cannot execute SQL statements that modify data. In simpler terms, this value allows read-only operations.
- MODIFIES SQL DATA. This value specifies that the function can execute any SQL statements that are supported within the routine type.

The default value depends on the type of routine. Procedures default to MODIFIES SQL DATA, and functions default to READS SQL DATA.

CONCURRENT ACCESS RESOLUTION

The CONCURRENT ACCESS RESOLUTION clause is supported for both SQL triggers and routines. It provides options to work with locked records.

Figure 3-8 shows the syntax of the CONCURRENT ACCESS RESOLUTION clause.

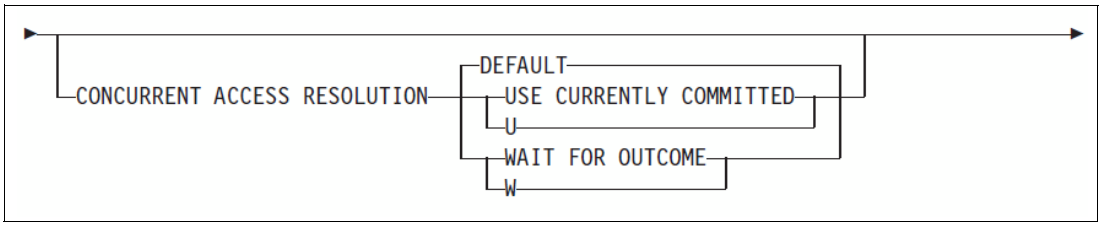

Figure 3-8 CONCURRENT ACCESS RESOLUTION clause syntax

When INSERT and UPDATE operations are run under commitment control, DB2 for i holds row-level locks on affected rows until the transaction is completed by a COMMIT or ROLLBACK operation. These locks can block other processes that are trying to read data in the same table, which can affect scalability and performance. The CONCURRENT ACCESS RESOLUTION clause that is shown in Figure 3-8 on page 60 provides triggers and functions with additional options to manage this situation.

The options are defined:

- DEFAULT specifies that the concurrent access resolution is not explicitly set for this trigger or function. The value that is in effect when the trigger is activated or when the function is invoked will be used. (The concurrent access resolution behavior can be configured as a QAQQINI option, and at the program level, as a connection property for interfaces, such as Java Database Connectivity (JDBC) and Object Database Connectivity (ODBC). It can also be specified at the SQL statement level.)
- WAIT FOR OUTCOME specifies that DB2 for i will wait for the lock, which will not be released until the transaction is completed by a COMMIT or ROLLBACK operation.
- USE CURRENTLY COMMITTED specifies that DB2 for i can read the currently committed version of the data when it encounters conflicting locks from pending changes. This option is implemented by reading data from the relevant journal. When the lock contention is between a read transaction and a delete or update transaction, the clause is applicable to scans with isolation level cursor stability (CS) (but not for CS KEEP LOCKS).

NOT SECURED and SECURED

The NOT SECURED or SECURED attribute is supported for both triggers and functions.

SECURED or NOT SECURED specifies whether the trigger or function is considered secure for row and column access control (RCAC). RCAC is supported on IBM i 7.2 and uses new SQL objects to control the users that can see data in the tables. An SQL PERMISSION can be used to manage the users that are allowed to see rows in a table. An SQL MASK can be used to manage the users that are allowed to see column values in a table.

They complement each other and provide more robust methods to manage data access for both SQL and record-level access applications. NOT SECURED is the default. NOT SECURED specifies that the trigger or function is not considered secure for use by RCAC.

The following requirements apply to NOT SECURED:

- NOT SECURED must not be specified explicitly or implicitly for a trigger whose subject table uses row access control or column access control.
- Also, NOT SECURED must not be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition uses row access control or column access control.
- NOT SECURED must not be specified on a function that references a column with an enabled column mask.

SECURED specifies that the trigger or function is considered secure for row and column access control.

The following requirements, which are the opposite of the NOT SECURED requirements, apply to SECURED:

- SECURED must be specified for a trigger whose subject table uses row access control or column access control.
- SECURED must be specified for a trigger that is created for a view and one or more of the underlying tables in the view definition uses row access control or column access control.
- SECURE must be specified when a function is referenced in a row permission or a column mask.

Implementation of RCAC can be complex. Do not implement RCAC without the correct guidance. We recommend that you read *Row and Column Access Support in IBM DB2 for i*, REDP-5110.

WRAPPED

The WRAPPED clause is supported for procedures, triggers, and functions to allow the use of an obfuscated version of the routine body. Support for triggers was added in IBM i 7.2. Figure 3-9 shows the WRAPPED clause syntax.

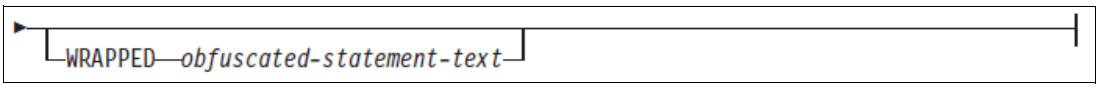

Figure 3-9 WRAPPED clause syntax

Figure 3-9 shows the syntax of the WRAPPED clause, which can be used on any of these SQL CREATE statements:

- CREATE FUNCTION (SQL scalar)
- CREATE FUNCTION (SQL table)
- CREATE PROCEDURE (SQL)
- CREATE TRIGGER

In an obfuscated statement, the procedural logic and embedded SQL are scrambled so that is harder to extract any intellectual property in its logic. Although, this scrambled procedural logic and embedded SQL is not a form of strong encryption, it can still be a valuable way to protect intellectual property.

The consequences of obfuscating the create statements for SQL routines and triggers are listed:

- The resulting routine or trigger does not reflect any debug options that were specified in its syntax or the use of the SET OPTION clause. This restriction is necessary to avoid the exposure of the logic of the routine or trigger.
- The source information that is stored in the SQL catalogs is the obfuscated version of the source. This approach is again necessary to prevent users from seeing the source for the routine or trigger.
- The resulting program or service program cannot be restored to a release where obfuscation is not supported.

Example 3-6 shows a simple CREATE FUNCTION statement.

Example 3-6 Non-obfuscated CREATE FUNCTION statement

```
CREATE OR REPLACE FUNCTION  Discount(inpSales DECIMAL(11,2))
  RETURNS DECIMAL(11,2)
  LANGUAGE SQL
  DETERMINISTIC  NOT FENCED
BEGIN
IF inpSales>10000 THEN RETURN inpSales*.02;
ELSE RETURN 0;
END IF;
END;
```

Example 3-7 shows the obfuscated version of the same statement.

Example 3-7 Obfuscated CREATE FUNCTION statement

```
CREATE OR REPLACE FUNCTION DISCOUNT ( INPSALES DECIMAL ( 11 , 2 ) )
 WRAPPED QSQ07020
aacxW8plW8FvG8FbG8FjG8Fn68pL68:nl8FJY9VJ1qpdW8pdW8pdX9FjaqebaqebavMy2iRpuyO:CrOuziDwwqPvNS2
YHNNpAW4ASe1nR9WjkSmKkgGAl6SnOl95xNmfJppXfvBYqG:9:qrtPOlrlANAoRyxRCqMxwMoaHifbv4OsJf_nN2VFv
jUFT29RtexQzrnmhCym1lXpBnNOkK7TjW6M13ACOsP:SGwrDnXuL:lInqRiNvu7rid:bfGNkidjbDQQgPRQwKua ;
```

The scrambled version consists of a prefix of the original CREATE statement up to and including the routine signature or trigger name, which is followed by the keyword **WRAPPED**. This keyword is followed by information about the application server that invoked the function. The information has the form *pppvvrrm*, which is explained:

- The *ppp* identifies the product by using the following three characters:
  - QSQ for DB2 for i
  - SQL for DB2 for Linux, UNIX, and Windows
- The *vv* is a 2-digit version identifier, such as '07'.
- The *rr* is a 2-digit release identifier, such as '02'.
- The *m* is a 1-character modification level identifier, such as '0'.

For example, DB2 for i 7.2 is identified as 'QSQ07020'.

This application server information is followed by a string of letters (a - z and A - Z), digits (0 - 9), underscores, and colons. The encoded Data Definition Language (DDL) statement can be up to one-third longer than the plain text form of the statement. If the result exceeds the maximum length for SQL statements, an error is issued, which is unlikely because the maximum statement length is over 2 million bytes.

Note: The obfuscated version of the CREATE statement is portable across DB2 for i instances. It can be deployed to other environments without exposing any intellectual property.

Two methods exist to obfuscate a CREATE statement: WRAP and CREATE WRAPPED.

WRAP

The WRAP function in schema SYSIBMADM can be used to transform a readable CREATE statement into an obfuscated version of the statement. The wrap function takes a single input parameter that contains the CREATE statement and returns a result of character large object type CLOB(2M), which contains the scrambled version of the CREATE statement in the input value.

CREATE WRAPPED

The CREATE_WRAPPED procedure in schema SYSIBMADM can be used to first transform a readable DDL statement into an obfuscated DDL statement and then to create the object in the database.

Like WRAP, the CREATE_WRAPPED procedure in schema SYSIBMADM takes a single input that contains the CREATE statement that you want for the procedure, trigger, or function. It uses the same form of encoding that is used by WRAP and the same token to identify the database and version.

3.3 DB2 sample database

The examples in this book use the sample database, which can be created by executing or calling the procedure CREATE_SQL_SAMPLE that is provided by IBM and specifying the name of the schema to be created, as shown.

Note: The name must be specified in uppercase.

```
CALL QSYS.CREATE_SQL_SAMPLE ('MYSAMPLEDB')
```

Several examples use additional objects to illustrate the concepts behind procedures, triggers, and functions. For access to the complete scripts, including the additional objects, see Appendix B, “Additional material” on page 313.

3.4 Transactions

Any routine or trigger, including those routines and triggers that are written in the SQL programming language, can update a number of rows in the same table or a number of rows in different tables. Transactions address the question of how databases maintain the consistency of the data if an error occurs, including errors that are detected by the logic of the routine or trigger and errors that are the result of a less predictable event, such as a dropped connection.

Journaling, commitment control, and savepoints play a major role in maintaining the transactional consistency of the database. When they are used correctly, they enable the program to more easily confirm (commit) or reverse (roll back) changes to a logically consistent state. Savepoints enhance these capabilities by providing more granularity in transaction management.

3.4.1 Transaction terminology

A *transaction* is a set of operations to be completed at one time as though they are a single operation. A transaction must be fully completed, or not performed at all. An example of a transaction is the transfer of funds from a savings account to a checking account. To the user, this activity 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 savings account to be debited if your checking account is not credited.

Commitment control is a function to define and process a group of changes to resources, such as database files or tables, as a *unit of work*. A *unit of work*, which is also known as a *transaction*, *logical unit of work*, or *unit of recovery*, is a recoverable sequence of operations. Each commitment definition involves the execution of one or more units of work. At any specific time, a commitment definition has a single unit of work.

A *unit of work* is started either when the commitment definition is started, or when the previous unit of work is ended by a commit or rollback operation. A unit of work is ended by a commit operation, a rollback operation, or the ending of the activation group. A commit or rollback operation affects only the database changes that were made within the unit of work that the commit or rollback ends. The start and end of a unit of work define the points of consistency within an activation group.

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. It controls whether changes are visible to other activation groups that use different commitment definitions. The possible values for commitment definitions are listed:

- No Commit
- Uncommitted Read
- Cursor Stability
- Read Stability
- Repeatable Read

A *savepoint* is a marker or milestone within a transaction to which data and schema changes can be undone. A *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 a user application.

3.4.2 Transaction management

All SQL programs execute as part of an application process. On IBM i, 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*. The programs run under a non-default activation group or the default activation group.

Note: The programs and service programs for SQL procedures, triggers, and functions all run in the caller's activation group, which is equivalent to specifying ACTGRP(*CALLER) on an HLL compiler.

An application process that uses commitment control can run with one or more commitment definitions. A *commitment definition* provides a means to scope commitment control at an activation-group level or a job level. At any specific time, an activation group that uses commitment control is associated with only one of the commitment definitions.

DB2 for i provides the following SQL statements for the transaction management:

- COMMIT
- SAVEPOINT
- ROLLBACK and ROLLBACK TO SAVEPOINT
- RELEASE SAVEPOINT
- SET TRANSACTION

COMMIT

The COMMIT statement ends a unit of work and commits the database changes that were made by that unit of work.

Figure 3-10 shows the result of a COMMIT operation.

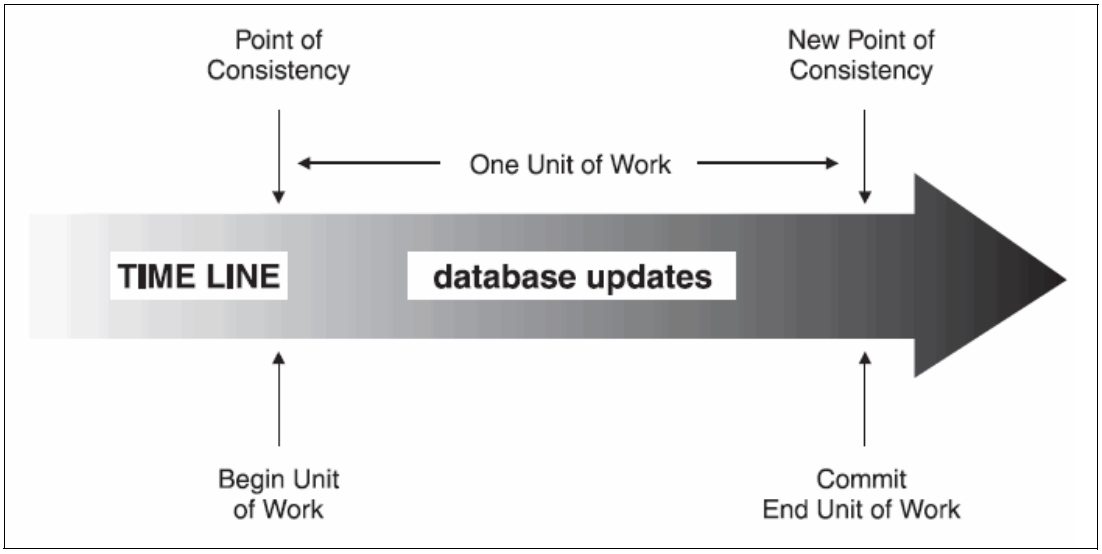

Figure 3-10 Unit of work with a COMMIT statement

SAVEPOINT

The SAVEPOINT statement is used to establish a milestone in the current unit of work. A name for the savepoint must be supplied. You can optionally specify whether that savepoint name needs to be unique. In that case, the savepoint cannot be reused across procedures, triggers, functions, and anything else that is involved in the unit of work.

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 changes to the beginning of the current transaction or to a specific savepoint. A ROLLBACK without SAVEPOINT backs out the current unit of work.

Figure 3-11 shows the ROLLBACK statement.

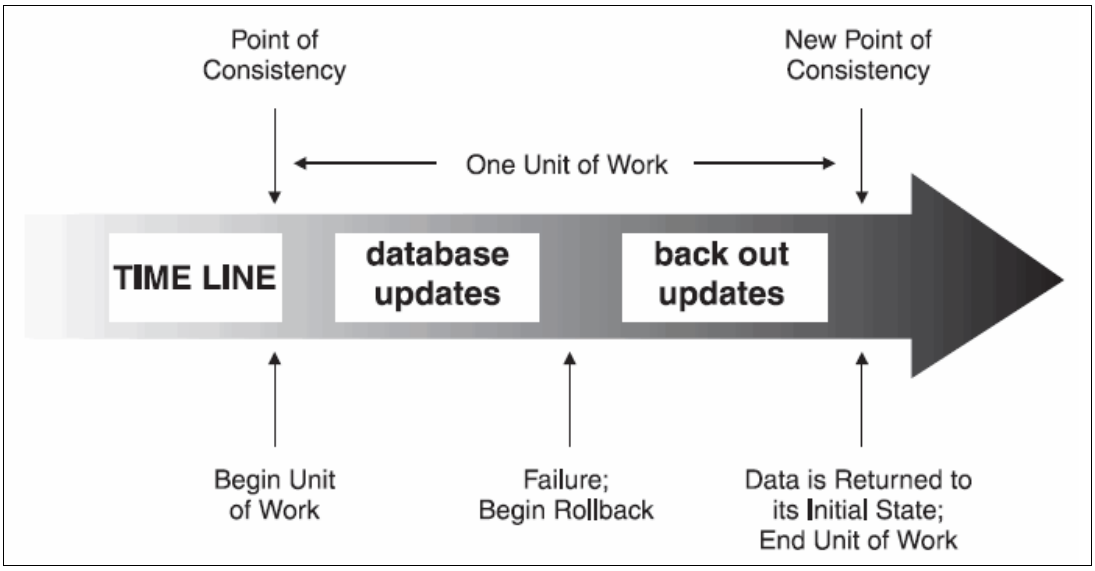

Figure 3-11 Unit of work with a ROLLBACK statement

When the ROLLBACK TO SAVEPOINT is used, the database is backed out to the last savepoint, as shown in Figure 3-12.

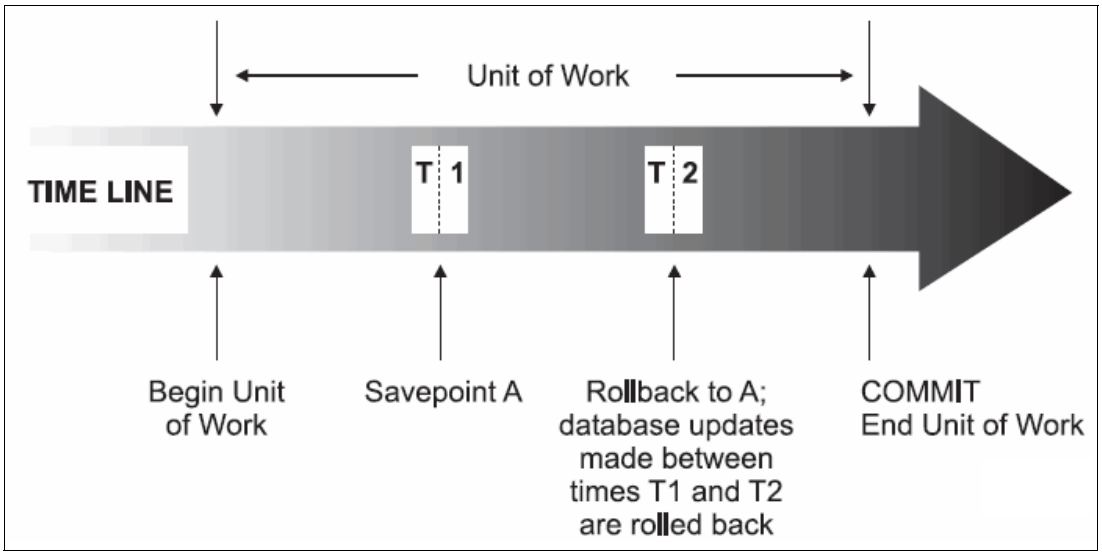

Figure 3-12 Using ROLLBACK TO SAVEPOINT

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. The isolation level can also be specified elsewhere, that is, as a program attribute or as a connection attribute for ODBC and JDBC.

Use the SET TRANSACTION statement 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.

3.4.3 Transaction management in compound statements

Use compound statements to group other statements in an SQL procedure, trigger, or function. Every compound statement starts with a BEGIN clause and ends with an END clause. In the BEGIN clause, you can specify the keyword **ATOMIC**. This keyword 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. NOT ATOMIC is the default behavior.

Note: If ATOMIC is specified, the COMMIT and ROLLBACK statements are not allowed.

UNDO can be specified in a handler in the compound statement only if ATOMIC was specified on the BEGIN clause.

The programs and service programs for SQL procedures, triggers, and functions all run in the *CALLER activation group. Therefore, any COMMIT or ROLLBACK operations also affect other database work within the activation group. For example, a database update that was performed before a procedure is called is also committed if the procedure performs a COMMIT.

Note: If ATOMIC is not specified, the COMMIT and ROLLBACK statements also affect any other operations in the activation group.

If ATOMIC is specified, a new savepoint is started so that only the database operations within the compound statements are committed or rolled back.

Note: If ATOMIC is specified, the implicit COMMIT and ROLLBACK operations affect operations in the compound statement only.

The following examples show an SQL procedure that was written to assign a new department number to an existing department. Because this information is contained in multiple tables, these procedures need to use commitment control to ensure that the database is left in a consistent state even if errors occur. Both versions of the procedure are functionally equivalent.

Example 3-8 shows the version of the ChangeDeptNo procedure that explicitly uses commitment control.

Example 3-8 ChangeDeptNo procedure by using NOT ATOMIC

```
CREATE OR REPLACE PROCEDURE ChangeDeptNo ( i_olddept CHAR(3), i_newdept CHAR(3))
  SPECIFIC ChgDeptNo
  LANGUAGE SQL
  SET OPTION COMMIT=*CHG                                                    1
```

```
BEGIN NOT ATOMIC                                                          2
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
      ROLLBACK;
      SIGNAL SQLSTATE '79001';
    END;                                                                  3
  INSERT INTO department (deptno, deptname, mgrno, admrdept, location)
    SELECT i_newdept, deptname, mgrno, admrdept, location FROM department
      WHERE deptno=i_olddept;                                             4
  UPDATE employee SET workdept=i_newdept WHERE workdept=i_olddept;
  UPDATE department SET admrdept=i_newdept WHERE admrdept=i_olddept;
  UPDATE project SET deptno=i_newdept WHERE deptno=i_olddept;             5
  DELETE FROM department WHERE deptno=i_olddept;                          6
  COMMIT;                                                                 7
END;
```

Notes about Example 3-8:

1 The ChangeDeptNo procedure takes two inputs: the old department number and the new department number. It is written in SQL, and it uses the SET OPTION statement to establish transactional control by using *CHG, which is the same as uncommitted read. The CREATE OR REPLACE syntax was used to allow the drop of an existing procedure, if any.

2 The BEGIN clause specifies NOT ATOMIC, which indicates that transaction management will be performed explicitly by the programmer.

3 A single handler exists for any SQL exception that will ROLLBACK the current transaction and signal an *SQLSTATE* of '79001' to the caller. It receives control if any of the following SQL statements fail.

4 The INSERT statement is used to insert a new row that contains the new department number into the department table. This action is performed first to avoid any issues with referential integrity constraints on the other tables.

5 Three UPDATE statements are used to update the corresponding columns in the tables: employee, department, and project.

6 The DELETE statement is used to remove the row from the table department that contains the old department number.

7 The COMMIT statement is used to commit the changes to all three tables.

Example 3-9 shows the version of the ChangeDeptNo procedure that uses ATOMIC on the BEGIN clause to implicitly use commitment control.

Example 3-9 ChangeDeptNo procedure by using ATOMIC

```
CREATE OR REPLACE PROCEDURE ChangeDeptNo ( i_olddept CHAR(3), i_newdept CHAR(3))
  SPECIFIC ChgDeptNo
  LANGUAGE SQL
  SET OPTION COMMIT=*CHG                                                  1
BEGIN ATOMIC                                                              2
  DECLARE UNDO HANDLER FOR SQLEXCEPTION SIGNAL SQLSTATE '79001';          3
  INSERT INTO department (deptno, deptname, mgrno, admrdept, location)
    SELECT i_newdept, deptname, mgrno, admrdept, location FROM department
      WHERE deptno=i_olddept;                                             4
  UPDATE employee SET workdept=i_newdept WHERE workdept=i_olddept;
  UPDATE DEPARTMENT SET ADMRDEPT=i_newdept WHERE ADMRDEPT=i_olddept;
  UPDATE PROJECT SET DEPTNO=i_newdept WHERE DEPTNO=i_olddept;             5
```

```
  DELETE FROM department WHERE deptno=i_olddept;                    6
END;                                                                7
```

Notes about Example 3-9: Changes from the preceding example are noted.

1 The ChangeDeptNo procedure takes two inputs: the old department number and the new department number. It is written in SQL, and it uses the SET OPTION statement to establish transactional control by using *CHG, which is the same as uncommitted read. The CREATE OR REPLACE syntax was used to allow the drop of an existing procedure, if any.

2 In this example, the BEGIN clause specifies ATOMIC rather than NOT ATOMIC, which indicates that transaction management will be performed implicitly by the database.

3 A a single handler exists for any SQL exception. It differs because it specifies UNDO, which implicitly results in a ROLLBACK. It will still signal an *SQLSTATE* of '79001' to the caller. It receives control if any of the following SQL statements fail.

4 The INSERT statement is used to insert a new row that contains the new department number into the department table. This action is performed first to avoid any issues with referential integrity constraints on the other tables.

5 Three UPDATE statements are used to update the corresponding columns in the tables: employee, department, and project.

6 The DELETE statement is used to remove the row from the table department that contains the old department number.

7 This line differs because no explicit COMMIT statement is listed. The database will commit the changes when the compound statements reach the END.

3.5 DB2 for i catalog views

DB2 for i 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 contain information about the following objects and types of information:

- Tables
- UDFs
- Distinct types
- Parameters
- Procedures
- Packages
- Views
- Indexes
- Aliases
- Sequences
- Variables
- Constraints
- Triggers
- XSR objects
- Languages that are supported by DB2 for i

The catalog also contains information about all relational databases that are accessible from this system.

Three classes of catalog views are available:

- IBM i catalog tables and views

  The IBM i catalog tables and views are modeled after but not identical to the American National Standards Institute (ANS) and International Organization for Standardization (ISO) catalog views. They are compatible with earlier releases of DB2 for i and exist in schemas QSYS and QSYS2. These catalog tables and views contain information about all tables, parameters, procedures, functions, distinct types, packages, XSR objects, views, indexes, aliases, sequences, variables, triggers, and constraints in the entire relational database. When an SQL schema is created, an additional subset of these views is created in the schema that contains information about only the tables, packages, views, indexes, and constraints in that schema.

- ODBC and JDBC catalog views

  The ODBC and JDBC catalog views are designed to satisfy ODBC and JDBC metadata application programming interface (API) requests. These views will be modified as ODBC and JDBC change their metadata APIs. These views exist in the SYSIBM schema.

- ANSI and ISO catalog views

  The ANSI and ISO catalog views are designed to comply with the ANSI and ISO SQL standard (the Information Schema catalog views) and will be modified as the ANSI and ISO standards change. Specific columns are reserved for future standard enhancements.

  Two versions of these views are available:

  – The first version of these views exists in the INFORMATION_SCHEMA schema. Only rows that are associated with objects to which the user has a privilege are included in the views. This version is compatible with the ANSI and ISO SQL standard. (INFORMATION_SCHEMA is the ANSI and ISO SQL standard schema name for the catalog views. It is a synonym for QSYS2.)

  – The second version of these views exists in the SYSIBM schema. All rows are included in these views whether or not the user has a privilege to the objects that are associated with rows in the views.

This book focuses on the IBM i catalog views. Table 3-2 summarizes the DB2 for i catalog views that contain information about SQL routines.

Table 3-2 DB2 for i catalog views for routines

Catalog view name	Description	Routine types
SYSPARMS	This catalog view contains one row for each parameter of a procedure or function. Also, this catalog view includes the result of a scalar function and the result columns of a table function.	Procedures and UDFs
SYSROUTINEAUTH	This catalog view contains one row for every privilege that is granted on a routine. This catalog view cannot be used to determine whether a user is authorized to a routine because the privilege to use a routine can be acquired through a group user profile or special authority, such as *ALLOBJ.	Procedures and UDFs
SYSROUTINEDEP	This catalog view records the dependencies of routines.	Procedures and UDFs

Catalog view name	Description	Routine types
SYSROUTINES	This catalog view contains one row for each procedure that is created by the CREATE PROCEDURE statement and each function that is created by the CREATE FUNCTION statement.	Procedures and UDFs
SYSPROCS	This catalog view contains one row for each procedure that is created by the CREATE PROCEDURE statement.	Procedures
SYSFUNCS	This catalog view contains one row for each function that is created by the CREATE FUNCTION statement.	UDFs
SYSTRIGCOL	This catalog view contains one row for each column that is either implicitly or explicitly referenced in the WHEN clause or the triggered SQL statements of a trigger.	Triggers
SYSTRIGDEP	This catalog view contains one row for each object that is referenced in the WHEN clause or the triggered SQL statements of a trigger.	Triggers
SYSTRIGGERS	This catalog view contains one row for each trigger in an SQL schema.	Triggers
SYSTRIGUPD	This catalog view contains one row for each column that is identified in the UPDATE column list, if any.	Triggers

The following considerations apply to the use of the catalog data:

- SYSROUTINES contains a ROW_TYPE column with possible values of 'PROCEDURE' and 'FUNCTION'. They can be used for information about either procedures or functions. SYSPARMS, SYSROUTINEDEP, and SYSROUTINEAUTH do not contain a column for row type but they can be joined to SYSROUTINES by using the columns SPECIFIC_NAME and SPECIFIC_SCHEMA.
- SYSPROCS and SYSFUNCS are views on SYSROUTINES that contain the subset of its columns that apply to the routine type, plus additional columns that are unique to the routine type.

Additional information about the catalog views is provided in the chapters that are specific to procedures, triggers, and functions.

4

Procedures

This chapter describes and illustrates how to code and implement Structured Query Language (SQL) procedures in IBM DB2 for i.

This chapter includes the following topics:

- Introduction to procedures
- Structure of a procedure
- Creating a procedure
- System catalog tables for procedures
- Procedure signature and procedure overloading
- Calling a procedure
- Producing and consuming result sets
- Handling errors
- Summary

4.1 Introduction to procedures

Two ways exist to implement procedures. The highly recommended choice for data-centric development is to write SQL-only procedures. The second approach is to write the procedure in any high-level language (HLL) program. This kind of procedure is described as an *external procedure*. With this approach, you can use a host language that you are familiar with, such as C, RPG, or COBOL.

Embedded SQL and external procedures are not covered in this book. For more information about embedding SQL in a host language program, see the IBM i Knowledge Center:

`http://www.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome`

For examples of using embedded SQL in external procedures, see *External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i*, SG24-6503.

Important: Although external procedures are not covered in this book, all procedures that are shown can be coded in an HLL by using embedded SQL. In certain cases, an external procedure might be a better choice. *When you choose to use external procedures, we strongly recommend that you use embedded SQL instead of traditional record-level access.*

The SQL CALL statement is used for the procedure invocation. The application waits for the procedure to terminate. Parameters can be passed back and forth, omitted, or listed in any order. Procedures can be called locally (on the same system where the application runs) or remotely on a different system.

4.2 Structure of a procedure

An SQL procedure consists of the following parts:

- A procedure name
- The declared parameters (if any) that are passed to or returned from the procedure
- The procedure properties (the definition of the number of result sets, whether the procedure is deterministic, and the kind of SQL access that is included in the stored procedure)
- A set of options that control how the stored procedure is created
- A routine body

SQL control statements are the basic programming constructs in most procedural languages. For an introduction to the SQL Persistent Stored Module (PSM) language, see Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5.

For a detailed description of the syntax and statements of SQL PSM, see the DB2 for i SQL reference, which is at this website:

`https://ibm.biz/Bd42dh`

4.3 Creating a procedure

The following section describes the syntax for CREATE PROCEDURE. The syntax is explained step-by-step with a description that follows each part of the syntax diagram. This information is taken directly from the DB2 for i SQL reference, which often contains additional footnotes about syntax. This information will be covered for each figure.

Note: Other tools might provide wizards or templates to facilitate the process of creating procedures.

4.3.1 CREATE PROCEDURE syntax

Figure 4-1 shows an overview of the CREATE PROCEDURE syntax.

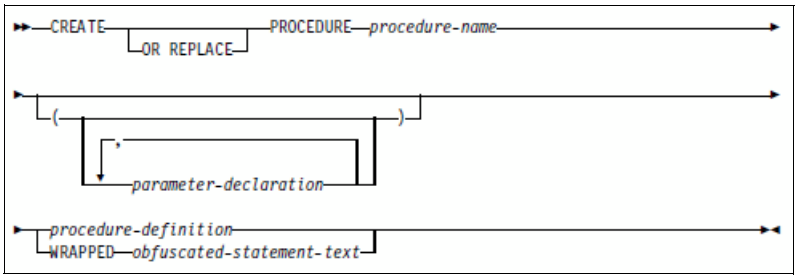

Figure 4-1 Syntax overview for CREATE PROCEDURE

The parameter-declaration and procedure-definition clauses are described in more detail later in this chapter. Certain options are described in more detail here.

Every procedure starts with CREATE PROCEDURE, which is followed by the name of the procedure. The name can be fully qualified. If the procedure name is not qualified, the default schema of the procedure follows the rules for deployment of unqualified database objects. For more information about unqualified object names, see Chapter 3, “SQL fundamentals” on page 47.

The OR REPLACE is an optional clause. It eliminates the need to use the DROP PROCEDURE statement when you deploy an existing procedure. It is a preferred practice to always code the OR REPLACE even if it is the first attempt at deployment.

The WRAPPED obfuscated-statement-text clause is common to all routines, and it is covered in more detail in Chapter 3, “SQL fundamentals” on page 47.

Parameter-declaration

The parameter declarations are enclosed within parentheses and immediately follow the procedure name. Figure 4-2 shows the syntax.

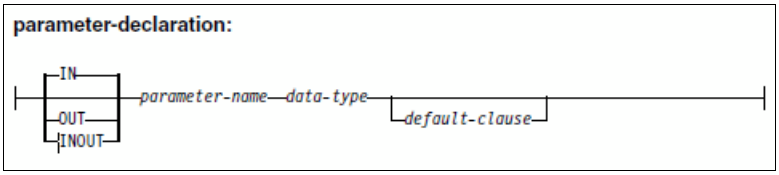

Figure 4-2 Syntax for parameter declaration

Figure 4-2 on page 75 shows the syntax of the parameter declaration clause. The purpose of each clause is described:

- A parameter can be declared as one of three types: IN, OUT, or INOUT. IN is the default type for the procedure. Although IN is not required, it is a preferred practice to explicitly code the IN when multiple types of parameters are declared in the same procedure.
- The name of the parameter can be up to 128 characters. It must be unique in the procedure. A preferred practice is to use a naming convention (perhaps a prefix, such as the generic `p_` or the letters `i`, `o`, or `io` to designate the parameter type) that prevents the possible duplication of the parameter name with a declared variable or table column name. For example, the table column name is WORKDEPT. However, the parameter name can be p_work_department, which avoids duplication and is more descriptive.
- Each parameter must have a valid data type. These data types will be described in more detail.
- Each parameter can have a default value. The default values are described in “Default clause” on page 79.

Data types

Figure 4-3 shows the syntax for the data type declaration.

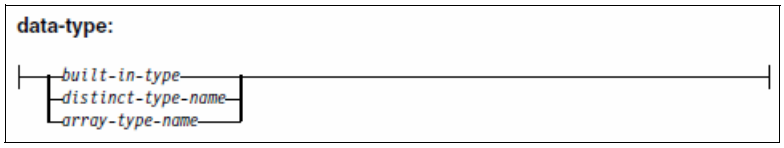

Figure 4-3 Syntax for data type declarations

The data type for a procedure parameter can be the built-in types, a distinct type (which is also known as a *user-defined type*), or an array type.

For more details about the built-in and distinct data types, see the DB2 for i SQL reference, which is at this website:

`https://ibm.biz/Bd42dh`

Array type

Arrays were used by application programmers in conventional programming languages for decades. Procedures (and functions) support parameters and variables of array types. Arrays are a convenient way to pass multiple like values between procedures by using a single parameter.

Within procedures and functions, arrays can be manipulated as arrays are manipulated in conventional programming languages. In addition, several functions exist that allow arrays to be easily converted into a table. Data from a table column can be aggregated into an array. For additional information about array types and supporting functions, see the IBM i Knowledge Center:

`http://www.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome`

Figure 4-4 shows the syntax of the CREATE TYPE SQL statement that is used to define and create an array type.

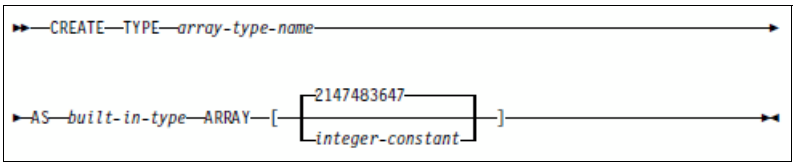

Figure 4-4 CREATE TYPE syntax

Currently, no OR REPLACE option exists for the CREATE TYPE statement. Therefore, a DROP TYPE is required to run first when you replace an existing array type. Example 4-1 contains the CREATE TYPE statement that is used to create an array of compensation amounts that can be used in a Payroll or Human Resources application. The actual number of array elements will be determined by the SQL procedure.

Example 4-1 Example of creating an array type

```
CREATE TYPE compensation_array AS DECIMAL(9,2) ARRAY[];
```

The usage of an array type is similar to the array type that is used in traditional languages by application programmers. One major difference between SQL arrays and traditional host language arrays is that the number of array elements does not need to be defined. If no value is specified, the maximum integer value of 2,147,483,647 is used.

Not specifying the array elements might sound like a good idea. However, by not specifying the array elements, you might affect the performance of highly used procedures. The preferred practice is to determine a reasonable maximum cardinality and code that cardinality on the CREATE TYPE.

Example 4-2 contains examples of creating an array type.

Example 4-2 Examples of creating array types

```
-- There are currently 3 compensation columns. Creating the array large enough for
-- 10 is probably enough to handle future compensation types.
CREATE TYPE compensation_array AS DECIMAL(9,2) ARRAY[10];
--There are currently 12 months in the Gregorian calendar
CREATE TYPE months_array AS SMALLINT ARRAY[12];
-- Assume there are 7 years of reporting history or 30 years in a calendar table.
-- A 100 elements may be more than enough but the size of each element is small.
CREATE TYPE reporting_years AS SMALLINT ARRAY[100];
```

An element of a user-defined array type can be referenced anywhere that an expression that returns the same data type as an element of that array can be used. Another difference between SQL and traditional host languages (RPG and COBOL) is that brackets [x] are used when you define an array that references an array element.

An array value can be empty (cardinality zero), null, or the individual elements in the array can be null or not null. An empty array is different than an array value of null, or an array for which all elements are the null value.

An array and its contents cannot be stored in the database.

Example 4-3 contains a procedure that updates the employee compensation by using the values that are contained in an array parameter. The values are defined based on the compensation array that is shown in Example 4-1 on page 77.

Example 4-3 The use of an array type within an UPDATE procedure

```
CREATE OR REPLACE PROCEDURE Change_Employee_Comp_Using_Array (      1
   IN p_empno CHAR(6),
   IN p_pay compensation_array                                      2
   )
   LANGUAGE SQL
P1: BEGIN
   DECLARE SQL_STRING CLOB (2M)
      DEFAULT 'UPDATE employee SET (SALARY, bonus, comm) = (?,?,?)  3
               WHERE empno = ?'
      ;
   DECLARE v_salary, v_bonus, v_comm DECIMAL(9,2);
   SET (v_salary, v_bonus, v_comm) = (p_pay[1], p_pay[2], p_pay[3]);4
   PREPARE PreparedStatement FROM sql_string;
   EXECUTE PreparedStatement USING v_salary, v_bonus, v_comm, p_empno;
END P1
```

Notes about Example 4-3:

1 This example is a rewrite of the PREPARE_THEN_EXECUTE procedure in Example 2-30 on page 27.

2 The input variable *p_pay* is defined based on the array compensation_array.

3 The parameter markers for the compensation values and search argument will be replaced by host variables as part of the EXECUTE statement.

4 The values in the array elements are assigned to the corresponding declared variables.

Example 4-4 contains an example of a procedure that populates the employee compensation array and calls the procedure that is shown in Example 4-3.

Example 4-4 Procedure to test the procedure that is shown in Example 4-3

```
CREATE OR REPLACE PROCEDURE Test_Change_Employee_Comp_Using_Array ()
   RESULT SETS 1
   LANGUAGE SQL
P1: BEGIN
   DECLARE v_pay compensation_array;                                1
   --Declare cursor
   DECLARE example13_c1 CURSOR WITH RETURN FOR - 2
       SELECT empno, salary, bonus, comm
       FROM employee
       WHERE workdept = 'D21';
   --Procedure logic begins here
   -- Update salary and set bonus and comm to zero
   SET v_pay = ARRAY[70000,0,0] ; 3
   CALL Change_Employee_Comp_Using_Array('650302', v_pay) ; 4
   -- Update salary and set bonus and comm to NULL
   SET v_pay = ARRAY[70000,NULL,NULL] ;                             --5
   CALL Change_Employee_Comp_Using_Array('650303', v_pay) ;         --4

   OPEN example13_c1;
END P1
```

Notes about Example 4-4 on page 78:

1 The host variable *v_pay* is defined based on the array compensation_array that was defined in Example 4-1 on page 77.

2 CURSOR is used to verify the result set to the calling application.

3 Update the salary and set the bonus and comm to zero.

4 The Change_Employee_Comp_Using_Array is called with the array passed as a variable.

5 This line is an example of populating an array with NULL values. Update the salary and set the bonus and comm to NULL.

Default clause

Procedures can be created with optional parameters that are similar in concept to omitted arguments that are used in the Integrated Language Environment (ILE) programming language. Optional procedure parameters are defined to have a default value immediately following the data type. If a default value is not defined, the parameter is required.

Figure 4-5 shows the syntax for the data type default clause.

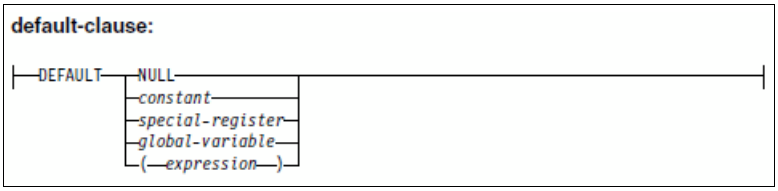

Figure 4-5 Default clause syntax

The use of default parameters provides the database engineer (DBE) with the capability to add parameters to existing procedures without breaking the existing application code. New applications can be written to use the new parameters, allowing mature applications to continue to run without change.

Parameters that are defined with the default clause can be omitted from the SQL CALL procedure statement. In addition, parameters can be specified in any order by specifying the parameter name in the call, which is referred to as a *named argument*.

For more information about named arguments, see the DB2 for i SQL reference at the following website:

`http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/sqlp/rbafyprocdefaults.htm?lang=en`

The parameter definition default clause consists of the **`DEFAULT`** keyword followed by the value to use if the parameter is omitted. The value can be any of the following values:

- NULL
- A constant value
- A special-register
- A global variable
- An expression

For more information about value types, see the `Language elements` section of the DB2 for i SQL reference:

`http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/db2/rbafzintro.htm?lang=en`

Example 4-5 shows the usage of NULL and the CURRENT DATE special register as default values.

Example 4-5 Add_New_Employee procedure

```
CREATE OR REPLACE PROCEDURE Add_New_Employee (
-- Required parameters                                                    1
   IN p_empno CHAR(6),
   IN p_firstnme VARCHAR(12),
   IN p_midinit CHAR(1),
   IN p_lastname VARCHAR(15),
   IN p_workdept CHAR(3),
   IN p_edlevel SMALLINT,
   IN p_salary DECIMAL(9,2),
-- Default parameters
   IN p_phoneno CHAR(4) DEFAULT NULL,                                     2
   IN p_hiredate DATE DEFAULT CURRENT DATE,                               3
   IN p_job CHAR(8) DEFAULT NULL,
   IN p_gender CHAR(1) DEFAULT NULL,
   IN p_birthdate DATE DEFAULT NULL,
   IN p_bonus DECIMAL(9,2) DEFAULT NULL,
   IN p_comm DECIMAL(9,2) DEFAULT NULL)

   SPECIFIC ADDNEWEMP                                                     4

P1 : BEGIN ATOMIC

   INSERT INTO employee (empno, firstnme, midinit, lastname, workdept,
            edlevel, salary, phoneno, hiredate, job, sex,
            birthdate, bonus, comm)
    VALUES (p_empno, p_firstnme, p_midinit, p_lastname, p_workdept,
          p_edlevel, p_salary, p_phoneno, p_hiredate, p_job, p_gender,
          p_birthdate, p_bonus, p_comm) ;
END P1
```

Notes about Example 4-5:

1 These parameters are all required.
2 This line is an example of a NULL default.
3 This line is an example of the CURRENT DATE special register default.
4 This line is an example of a short SPECIFIC name. For more information, see “SPECIFIC specific-name” on page 82.

Example 4-6 contains various CALL statements that contain required and default values.

Example 4-6 CALL statements with required and default parameters

```
Required parameters only
   CALL Add_new_employee (
      '650302' , 'Dan' , 'X' , 'Cruikshank' , 'D21' , 20 , 50000.00 );
All required plus phone and hire date
   CALL Add_new_employee ( '650303' , 'Jim' , 'Y' , 'Denton' , 'D21' , 24 ,
   60000.00, '1234', '2015-09-10' );
All required plus phone and bonus (named argument)
   CALL Add_new_employee ( '650305' , 'Hernando' , 'A' , 'Bedoya' , 'D21' , 24 ,
   60000.00, '5678', p_bonus => 6000.00 );
```

```
All required plus phone number and job description (named argument)
   CALL Add_new_employee ( '650306' , 'Rob' , 'O' , 'Bestgen' , 'D21' , 28 ,
   75000.00, '7890', p_job => 'Designer' );
```

Example 4-7 contains the results of creating the new employee row by using the procedure.

Example 4-7 Results of adding new employees by using DEFAULT

```
EMPNO  FIRSTNME I LASTNAME   WORKDEPT EDLEVEL SALARY   PHONENO HIREDATE   JOB
GENDER BIRTHDATE BONUS   COMM
------ -------- - ---------- -------- ------- -------- ------- ---------- --------
------ --------- ------- ----
650302 Dan      X Cruikshank D21           20 50000.00 NULL    NULL       NULL
NULL   NULL          NULL NULL
650303 Jim      Y Denton     D21           24 60000.00 1234    2015-09-10 NULL
NULL   NULL          NULL NULL
650305 Hernando A Bedoya     D21           24 60000.00 5678    2015-11-28 NULL
NULL   NULL       6000.00 NULL
650306 Rob      O Bestgen    D21           28 75000.00 7890    2015-11-28 Designer
NULL   NULL          NULL NULL
```

For examples of using global variables as defaults, see Chapter 8, “Creating flexible and reusable procedures” on page 227.

Procedure-definition

The SQL procedure definition follows the parameter declarations. It is made up of three main parts: an option list, an optional SET OPTION statement, and the SQL routine body.

Figure 4-6 contains the syntax diagram for the procedure definition.

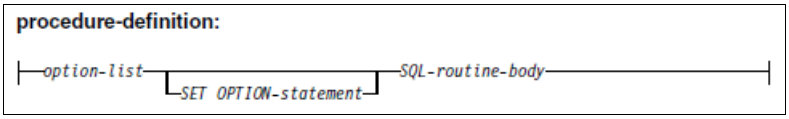

Figure 4-6 Syntax for procedure-definition

Option-list

This section contains details about the option list properties for an SQL procedure.

Figure 4-7 shows the syntax for the option-list clause for procedures.

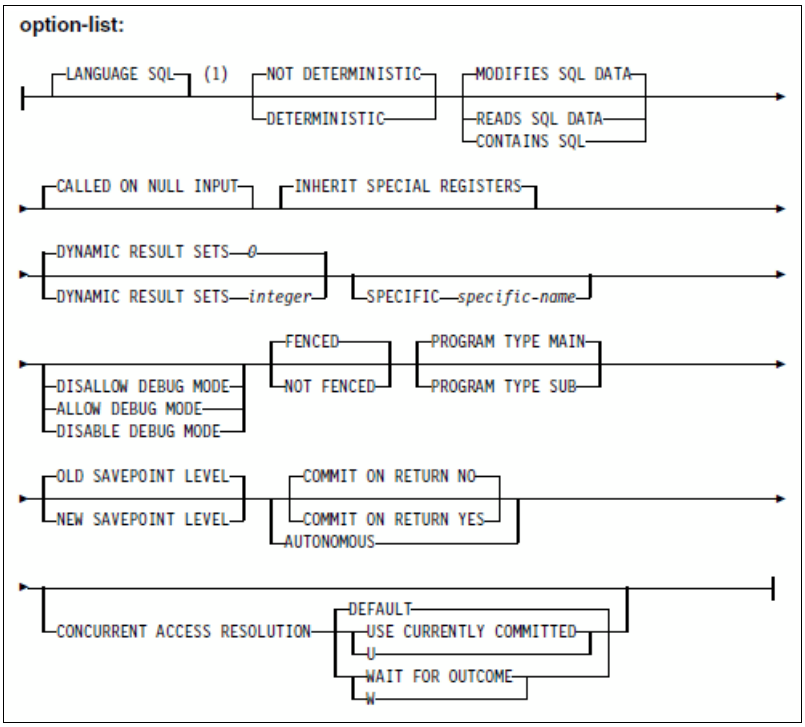

Figure 4-7 Option-list syntax diagram

The options are described in more detail next.

LANGUAGE SQL

LANGUAGE SQL is the default value for procedures. Therefore, it does not need to be coded. Certain development tools might generate this value automatically, which is acceptable and does not affect the generated procedure code.

SPECIFIC specific-name

A specific-name can be used with the procedure name to further identify the procedure. This specific-name can be helpful when you define multiple procedures with the same name and schema but with a different number of parameters.

When you use the OR REPLACE option to replace an existing procedure, the specific-name and procedure-name of the new definition must be the same as the specific-name and procedure-name of the old definition. Otherwise, another procedure is created.

Important: DB2 for i allows both long procedure/specific names (128 characters) and short (up to 10 characters) system names. The use of the short system name might be required to use certain IBM i products or to access information for system catalogs. Therefore, we recommend that you use 10-character specific names for procedures in addition to a meaningful long name.

DETERMINISTIC or NOT DETERMINISTIC

DETERMINISTIC or NOT DETERMINISTIC specifies whether the procedure returns the same results each time that the procedure is called with the same IN and INOUT arguments. The default is NOT DETERMINISTIC, which means that the same result might not be returned. In high-volume transaction processing environments, the database constantly changes. It is unlikely that the same results will be returned in this environment. If the results depend on the contents of a table that does not typically change, the attribute must be NOT DETERMINISTIC because the contents might change.

If the same procedure is called repetitively within the same session, perhaps a procedure is not the best tool for the job. Consider a function or a join in this case.

MODIFIES SQL DATA

MODIFIES SQL DATA specifies the classification of SQL statements that this procedure can execute, *which includes any routine that is called, or default expression that is used, by this procedure*. The database manager verifies that the SQL statements that are issued within this procedure, the SQL statements that are issued by routines that are called by this procedure, and any default expressions that are used by parameters are consistent with this specification. The default is MODIFIES SQL DATA, which is the least restrictive.

For more information about other options, see Chapter 3, “SQL fundamentals” on page 47.

CALLED ON NULL INPUT

CALLED ON NULL INPUT is the default value for procedures. Therefore, it does not need to be coded.

INHERIT SPECIAL REGISTERS

INHERIT SPECIAL REGISTERS is the default value for procedures. Therefore, it does not need to be coded.

DYNAMIC RESULT SETS integer

DYNAMIC RESULT SETS *integer* specifies the maximum number of result sets that can be returned from the procedure. The minimum value for integer is 0, and the maximum value is 32768. The default is DYNAMIC RESULT SETS 0.

Even though the default is 0, if DYNAMIC RESULT SETS is not specified, cursors that are left open when the procedure ends are returned as consumable result sets to the caller. A preferred practice is to explicitly code DYNAMIC RESULT SETS 0 and ensure that all cursors that opened within this procedure are closed before it returns to the caller.

For more information about result sets, see 4.7, “Producing and consuming result sets” on page 91.

DISALLOW, ALLOW, or DISABLE DEBUG MODE

These options are closely tied to the tool that is used for debugging. For more information about debugging procedures, functions, and triggers, see Chapter 7, “Development and deployment” on page 189.

FENCED or NOT FENCED

This parameter is ignored by DB2 for i.

PROGRAM TYPE MAIN or PROGRAM TYPE SUB

Use PROGRAM TYPE MAIN or PROGRAM TYPE SUB to designate the type of C program object to generate. PROGRAM TYPE MAIN generates a C program. PROGRAM TYPE SUB generates a C service program. Service programs might be better suited for highly reused procedures.

For more information, see Chapter 3, “SQL fundamentals” on page 47.

OLD SAVEPOINT LEVEL or NEW SAVEPOINT LEVEL

Savepoints are a way to specify an interval point within a procedure that might process a large transaction that contains several INSERT, UPDATE, DELETE, or MERGE statements or contains calls to procedures that perform INSERT, UPDATE, DELETE, or MERGE as part of the same unit of work. By using savepoints, you can control the point to which the procedure processing must return if a ROLLBACK occurs.

Old Savepoint Level means that any savepoints that are created within the procedure are scoped to the same level as the caller of the procedure. Savepoint names must be unique and cannot be the same as the calling procedure.

New Savepoint Level means that savepoint names in the called procedure are created at a different level than the calling procedure. Therefore, they can be the same names as the caller savepoints.

For more information, see Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5.

COMMIT ON RETURN

Certain heritage databases, such as DB2 for i, gave the application developers a choice to use commitment control or not. As a DBE, the use of commitment control is no longer a debatable option. It is a requirement for ensuring data integrity. The use of COMMIT ON RETURN can simplify the use of commitment control at the same time, minimizing the impact of commitment control on the system.

The specification of COMMIT ON RETURN YES for the outermost procedure that initiates a transaction ensures that all work that was performed by the procedure and any procedures that were called by this procedure are committed upon successful completion.

Conversely, if the procedure returns with an error, a commit is not issued. An implicit ROLLBACK occurs if BEGIN ATOMIC was used at the start of the SQL routine body.

If the procedure returns result sets, the cursors that are associated with the result sets must be defined in advance as WITH HOLD to be usable after the commit.

For an example of the use of COMMIT ON RETURN, see Chapter 8, “Creating flexible and reusable procedures” on page 227.

AUTONOMOUS

Within a session, a procedure runs within an activation-group. The activation-group-level commitment definition is the default scope. It starts implicitly when a procedure runs with an isolation level other than *NONE (no commit). The AUTONOMOUS option allows a procedure to run within an explicitly named activation group. This capability is important if the procedure needs to perform its own commitment control transactions without affecting any transactions that are performed by procedures that are running within the default activation group.

When you develop modular SQL applications, which are running under commitment control, it is possible that a procedure might be called that performs an INSERT, an UPDATE, or a DELETE operation that is independent of the current unit of work. In this case, the specification of AUTONOMOUS for that procedure allows the database to always commit or roll back the procedure's transactional work without affecting the pending transaction of the caller.

The decision to commit or to roll back is based on the *SQLSTATE* that is returned from the procedure. An *SQLSTATE* indication of unqualified success or warning commits the transaction. All other SQLSTATEs roll back the autonomous procedure's unit of work.

For more information, see the DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

CONCURRENT ACCESS RESOLUTION

For more information, see Chapter 3, "SQL fundamentals" on page 47.

SET OPTION statement

This section contains specific options with the preferred practice recommendations. For the syntax diagram of the SET OPTION statement and for more detailed information about common options, see Chapter 3, "SQL fundamentals" on page 47.

DBGVIEW

The DBGVIEW option is used to set the format of the procedure that is debugged. The syntax for this option is shown in Figure 4-8.

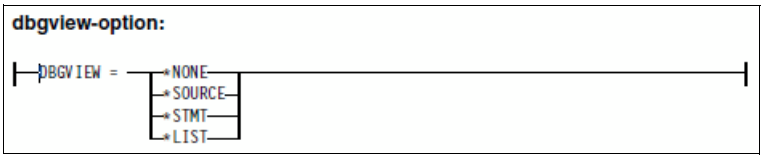

Figure 4-8 SET OPTION DBGVIEW syntax

The settings for this option might affect the debugger that is associated with your development tool of choice. For more information, see Chapter 7, "Development and deployment" on page 189.

DFTRDBCOL and DYNDFTCOL

The DFTRDBCOL option is used to specify the schema name to use for the unqualified names of tables, views, and SQL packages. See Figure 4-9.

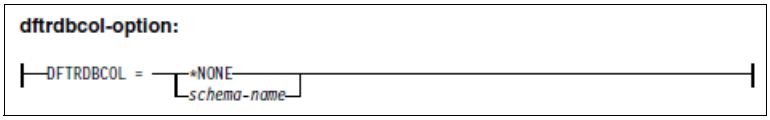

Figure 4-9 SET OPTION DFTRDBCOL syntax

When a schema name is provided, the schema name is used to override the naming convention that is specified on the OPTION precompile parameter or by the SET OPTION NAMING option.

DYNDFTCOL is used with DFTRDBCOL. The syntax is shown in Figure 4-10.

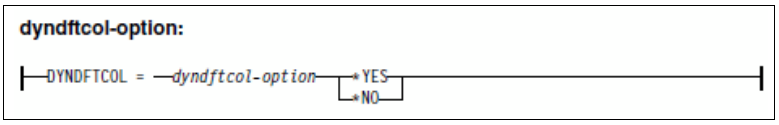

Figure 4-10 SET OPTION dyndftcol syntax

When the DYNDFTCOL value is set to YES, the schema name that is provided with DFTRDBCOL is also used for dynamic statements. A value of NO means that it will not use the DFTRDBCOL value.

This value can affect the execution of a procedure that is called from an external application, such as Java, Open Database Connectivity (ODBC), or .NET, depending on the connection properties that are used to connect to the database server.

For examples of connecting to the database server from different development tools, see Chapter 7, “Development and deployment” on page 189. For more information about unqualified object names, see Chapter 3, “SQL fundamentals” on page 47.

DYNUSRPRF

When a procedure is called by the client program, the statements in the 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 procedure, depending on how it was defined in the USRPRF attribute for that program object. For more information about the USRPRF option, see Chapter 3, “SQL fundamentals” on page 47.

The DYNUSRPRF option extends the authorities that are defined by the USRPRF option to use for dynamic SQL statements. The syntax for the DYNUSRPRF option is shown in Figure 4-11.

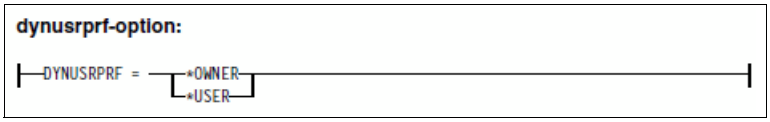

Figure 4-11 SET OPTION dynusrprf syntax

The value that you use depends on the security policies in use by your company. For unit testing, the value *USER suffices because the tester of the procedure is likely the owner. If the user is not the owner and might not possess the required permissions, *OWNER is appropriate.

EXTIND

Use the EXTIND option to go beyond the traditional indicators of 0 (set to current value) and -1 (set to NULL) when you perform INSERT or UPDATE operations. The extended indicator -5 can be used to set the value of a table column to the defined default value. The extended indicator -7 can be used to ignore the column in an UPDATE or INSERT statement.

Figure 4-12 shows the syntax for the EXTIND option.

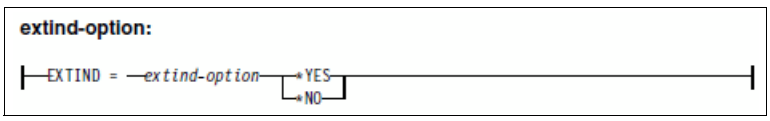

Figure 4-12 SET OPTION extind syntax

Use *YES to enable the use of extended indicators with SQL descriptors.

For an example of the usage of extended indicators with an SQL descriptor, see “Modifying data by using extended indicators” on page 254.

SQL routine body

The routine body of the procedure might consist of a single *SQL statement* (SELECT, UPDATE, INSERT, DELETE, and so on) or an *SQL compound statement*. An SQL compound statement might include assignment statements, flow-of-control statements, declaration of local variables, condition handlers, iterative statements, and SQL control statements. *SQL control statements* are the basic programming constructs in most procedural languages.

For an introduction to the SQL Persistent Stored Module (PSM) language, see Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5.

For a detailed description of the syntax and statements of SQL PSM, see the DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

4.4 System catalog tables for procedures

The IBM i catalog/DB2 for i maintains a set of catalog tables and views that contain information about procedures. The tables include SYSROUTINES, SYSROUTDEP, and SYSPARMS. The views include SYSPROCS and SYSROUTINEDEP. This section shows how to access the catalog information that relates to procedures.

For more information about the catalog views that are covered in this book, see Chapter 3, “SQL fundamentals” on page 47.

4.4.1 SYSROUTINES catalog

The SYSROUTINES catalog contains information that relates to both stored procedures and UDFs. The SYSPROCS catalog view contains information about procedures only.

Example 4-8 contains examples of the differences when you use SYSROUTINES versus SYSPROCS.

Example 4-8 SYSROUTINES versus SYSPROCS

```
SELECT routine_schema, routine_name, sql_data_access, specific_name,
   max_dynamic_result_sets, in_parms+out_parms+inout_parms AS total_parms, 1
   is_null_call
FROM qsys2.sysroutines
WHERE routine_schema = 'DANCRUIK'
   AND routine_type = 'PROCEDURE' AND routine_body = 'SQL'                2
ORDER BY specific_name;

SELECT Routine_schema, routine_name, sql_data_access, specific_name,
   result_sets, in_parms+out_parms+inout_parms AS total_parms             1
FROM qsys2.sysprocs
WHERE routine_schema = 'DANCRUIK' AND routine_body = 'SQL';
ORDER BY specific_name;
```

Notes about Example 4-8 on page 87:

1 The column max_dynamic_result_sets in SYSROUTINES is the same as the column result_sets in SYSPROCS.

2 The local selection on column routine_type is part of the SYSPROCS view.

Figure 4-13 contains the results after the second SQL statement from Example 4-8 on page 87 is run. The data in the result set is the same for either query.

Schema	Name	SQL Access	Specific	Result Sets	Total parms
DANCRUIK	ADD_NEW_EMPLOYEE	MODIFIES	ADD_N00001	0	7
DANCRUIK	ADD_DEPARTMENT_TRANSACTION	MODIFIES	ADDDPTTRNS	0	5
DANCRUIK	ADD_NEW_EMPLOYEE	MODIFIES	ADDNEWEMPL	0	14
DANCRUIK	ADD_TRANSACTION	MODIFIES	ADDTRNSACT	0	7
DANCRUIK	ALLOCATE_GLOBAL_DESCRIPTOR	MODIFIES	ALCGBLDESC	0	2
DANCRUIK	CHANGE_EMPLOYEE_COMP_USING_ARRAY	MODIFIES	CHANGE_EMP	0	2
DANCRUIK	CHANGE_EMPLOYEE_COMPENSATION	MODIFIES	CHGEMPCOMP	0	4
DANCRUIK	CHANGE_EMPLOYEE_INFORMATION	MODIFIES	CHGEMPFORM	0	5
DANCRUIK	CHANGE_DEPARTMENT_TRANSACTION	MODIFIES	CHGDPTTRNS	0	6
DANCRUIK	CHANGE_EMPLOYEE_DATA	MODIFIES	CHGEMPDATA	0	7
DANCRUIK	CHANGE_EMPLOYEE_COMP_USING_XTNDED_INDS	MODIFIES	CHGEXTNDIS	1	3
DANCRUIK	CONSUME_FLEXIBLE_RESULT_SET	MODIFIES	COMRESULTS	0	0
DANCRUIK	CONSTRUCT_CALL_STATEMENT	MODIFIES	DCALLSTMNT	1	2
DANCRUIK	DECLARE_AND_PREPARE	MODIFIES	DCLPRPDYNQ	1	1
DANCRUIK	DECLARE_AND_PREPARE	MODIFIES	DCLANDPREP	1	0
DANCRUIK	DESCRIBE_TABLE_DESCRIPTOR	MODIFIES	DESCRBEVAR	2	4
DANCRUIK	DESCRIBE_CURSOR_DESCRIPTOR	MODIFIES	DSCRIBECSR	2	2
DANCRUIK	DESCRIBE_OUTPUT_DESCRIPTOR	MODIFIES	DSCRIBEOUT	0	2
DANCRUIK	DESCRIBE_TABLE_DESCRIPTOR	MODIFIES	DSCRIBETBL	0	2

Figure 4-13 Query results for SYSPROCS (partial list)

4.4.2 SYSPARMS catalog

The SYSPARMS catalog table contains parameters for both UDFs and stored procedures. In addition, SYSPARMS contains the specific schema and name, not the routine schema and name, which eliminates duplicate routine names for overloaded procedures. However, you must know the specific name for the procedure to access the parameter details.

Example 4-9 contains the SQL statement that is used to display the parameter information for each parameter that is defined in a specific procedure.

Example 4-9 SQL statement to display the parameter information in a specific procedure

```
select * from qsys2.sysparms T1
where specific_schema = 'DANCRUIK' and specific_name = 'ADDNEWEMP';
```

Figure 4-14 shows the results after the query in Example 4-9 on page 88 is run.

```
SCHEMA     ROUTINE_NAME                                SQL_ACCESS SPECIFIC_NAME                    SETS PARMS
---------  ------------------------------------------- ---------- -------------------------------- ---- -----
DANCRUIK   ADD_NEW_EMPLOYEE                            MODIFIES   ADD_N00001                          0     7
DANCRUIK   ADD_DEPARTMENT_TRANSACTION                  MODIFIES   ADDDPTTRNS                          0     5
DANCRUIK   ADD_NEW_EMPLOYEE                            MODIFIES   ADDNEWEMP                           0    14
DANCRUIK   ADD_TRANSACTION                             MODIFIES   ADDTRNSACT                          0     7
DANCRUIK   ALLOCATE_GLOBAL_DESCRIPTOR                  MODIFIES   ALCGBLDESC                          0     2
DANCRUIK   CHANGE_EMPLOYEE_COMP_USING_ARRAY            MODIFIES   CHANGE_EMPLOYEE_COMP_USING_ARRAY    0     2
DANCRUIK   CHANGE_EMPLOYEE_COMPENSATION                MODIFIES   CHANGE_EMPLOYEE_COMPENSATION        0     4
DANCRUIK   CHANGE_EMPLOYEE_INFORMATION                 MODIFIES   CHANGE_EMPLOYEE_INFORMATION         0     5
DANCRUIK   CHANGE_DEPARTMENT_TRANSACTION               MODIFIES   CHGDPTTRNS                          0     6
DANCRUIK   CHANGE_EMPLOYEE_DATA                        MODIFIES   CHGEMPDATA                          0     7
DANCRUIK   CHANGE_EMPLOYEE_COMP_USING_XTNDED_INDS      MODIFIES   CHGEXTNDIS                          1     3
DANCRUIK   CONSUME_FLEXIBLE_RESULT_SET                 MODIFIES   COMRESULTS                          0     0
DANCRUIK   CONSTRUCT_CALL_STATEMENT                    MODIFIES   CONSTRUCT_CALL_STATEMENT            1     2
DANCRUIK   DECLARE_AND_PREPARE                         MODIFIES   DCLPRPDYNQ                          1     1
DANCRUIK   DECLARE_AND_PREPARE                         MODIFIES   DECLARE_AND_PREPARE                 1     0
DANCRUIK   DESCRIBE_TABLE_DESCRIPTOR                   MODIFIES   DESCRBEVAR                          2     4
DANCRUIK   DESCRIBE_CURSOR_DESCRIPTOR                  MODIFIES   DESCRIBE_CURSOR_DESCRIPTOR          2     2
DANCRUIK   DESCRIBE_OUTPUT_DESCRIPTOR                  MODIFIES   DESCRIBE_OUTPUT_DESCRIPTOR          0     2
```

Figure 4-14 SPARMS query result set (headings edited to fit)

4.5 Procedure signature and procedure overloading

Procedure overloading allows two or more procedures with the same name to exist in the same schema if they have different signatures. The signature of the 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:

- `MyProc( char(5), int)`
- `MyProc( int)`

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

- `MyProc( char(5))`
- `MyProc( int)`

The OR REPLACE option that is combined with the procedure-specific name eliminates the need to manually drop overloaded procedures. For more information, see “SPECIFIC specific-name” on page 82.

4.6 Calling a procedure

After the SQL procedure is created, you need to use the SQL CALL statement to execute it. The CALL statement names the procedure that is called and lists all required parameters that need to be passed to the procedure. Optional parameters can be passed on the CALL statement in any order. For more information, see 8.2.1, “Global variables as default parameters” on page 232.

4.6.1 CALL statement syntax

An SQL procedure must be called through an SQL CALL statement.

Figure 4-15 shows the syntax for the CALL statement.

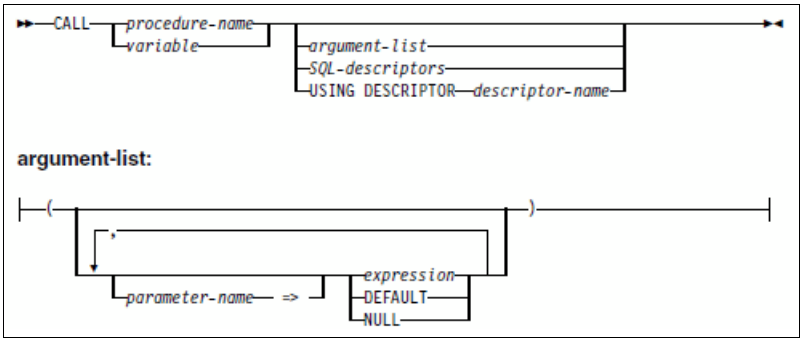

Figure 4-15 CALL statement syntax

On the CALL statement, the name of the stored procedure must be the same as the name that is specified on the CREATE PROCEDURE statement. Arguments can be constants, special registers, host variables, or expressions. When a procedure that contains INOUT or OUT parameters is called, the caller must be a procedure that contains local variables to receive the results of the called procedure.

SQL descriptors can be used instead of specifying individual variables. The descriptor must be allocated large enough to contain all of the IN and INOUT parameters that are defined by the called procedure. Figure 4-16 shows the syntax for the SQL DESCRIPTOR clause.

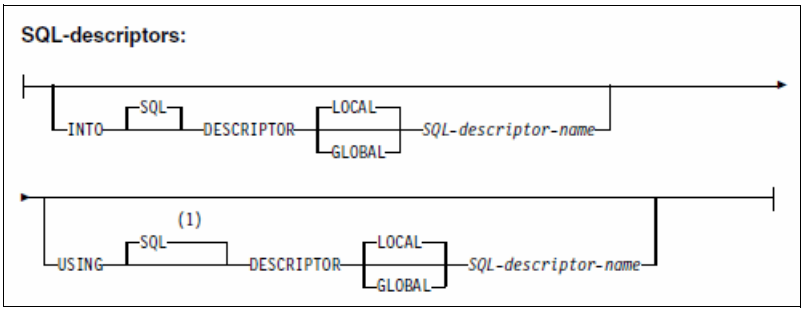

Figure 4-16 SQL DESCRIPTOR clause

The use of INTO SQL DESCRIPTOR versus USING SQL DESCRIPTOR is described.

INTO SQL DESCRIPTOR

The INTO SQL DESCRIPTOR clause can be used to receive the values for all of the parameters that are defined as INOUT or OUT. Example 4-10 contains an example of the assignment of a value to a local variable after the return from a called procedure.

Example 4-10 Assigning a value to an input descriptor

```
CREATE OR REPLACE proc2 (OUT p1 INTEGER) LANGUAGE SQL...
ALLOCATE SQL DESCRIPTOR ‘in_inout_parms’;
CALL proc1 () INTO SQL DESCRIPTOR ‘in_inout_parms’;
GET SQL DESCRIPTOR ‘in_inout_parms’ VALUE 1 vdata = DATA;
```

USING SQL DESCRIPTOR

The USING SQL DESCRIPTOR clause can be used to pass the values for the IN and INOUT parameters that are defined in the called procedure. Example 4-11 contains an example of the assignment of a value to an IN parameter before procedure proc1 is called.

Example 4-11 Assigning a value to a local variable from a descriptor

```
CREATE OR REPLACE proc1 (IN p1 CHAR(3)) LANGUAGE SQL...
ALLOCATE SQL DESCRIPTOR 'in_inout_parms';
SET SQL DESCRIPTOR 'in_inout_parms' VALUE 1 DATA = 'xyz';
CALL proc1 () USING SQL DESCRIPTOR 'in_inout_parms';
```

If all of the parameters of the procedure are INOUT parameters, the same descriptor can be used for both input and output.

For more information about input and output descriptors, see Appendix A, "Allocating, describing, and manipulating descriptors" on page 285.

4.7 Producing and consuming result sets

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

Whether a result set gets returned depends on the *return ability attribute* of the cursor. By default, cursors that are opened in a stored procedure are defined with a return ability attribute of RETURN TO CALLER. You define this return ability attribute by adding either of the following return ability attributes to the DECLARE CURSOR or SET RESULT SET statement:

- WITH RETURN or WITH RETURN TO CALLER is the default and returns result sets to the immediate caller.

  In the following example, when the cursors are opened, the result sets are available to the calling procedure:

  - `DECLARE specifict1 CURSOR WITH RETURN TO CALLER FOR SELECT * FROM t1`
  - `DECLARE specifict2 CURSOR FOR SELECT * FROM t2`

- WITH RETURN TO CLIENT 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. In the following example, three result sets are declared. When the cursors are opened, the result sets are returned in the order that they are specified on the SET RESULT SETS statement:

  - `DECLARE specifict1 CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM t1`
  - `DECLARE specifict2 CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM t2`
  - `DECLARE specifict3 CURSOR FOR WITH RETURN TO CLIENT SELECT * FROM t3`
  - `SET RESULT SETS CURSOR specifictt3, specifictt1, specifictt2`

For cursors whose result sets are never returned to caller or client, the return ability attribute of WITHOUT RETURN needs to be specified on the DECLARE CURSOR statement.

This syntax provides flexibility to design whether the result sets of a stored procedure, which is called in nested procedures, are returned to the *immediate calling procedure* or to the calling procedure at the *beginning of the calling chain,* as shown in Figure 4-17.

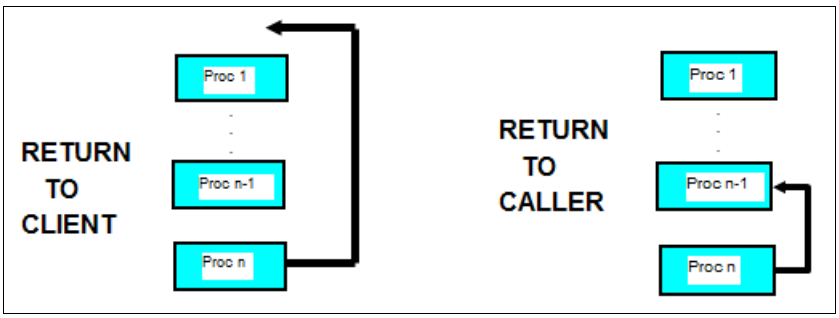

Figure 4-17 Returning result sets to a caller versus to a client

Result sets that are defined as RETURN TO CLIENT will be returned to the outermost caller of a procedure, if the caller is an external application that can consume the result set. ASSOCIATE LOCATOR and DESCRIBE PROCEDURE statements find result sets that are returned to only that invocation. If a procedure calls another procedure that uses the RETURN TO CLIENT attribute, an exception occurs.

4.7.1 Creating result sets in an SQL procedure

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

Figure 4-18 contains the syntax diagram for the DECLARE CURSOR statement.

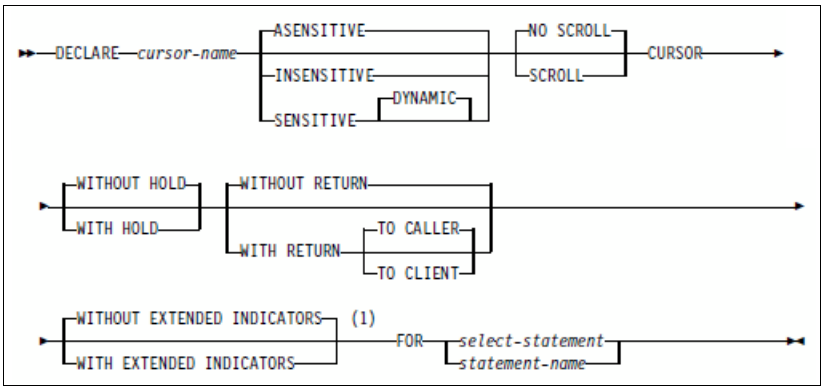

Figure 4-18 DECLARE CURSOR statement syntax

The cursor name can be up to 128 characters. The cursor name must be descriptive and unique. Avoid the use of a non-descriptive generic name, such as C1 or Cursor1. One approach is to append a suffix to the procedure (or specific) procedure name.

For information about the other clauses, see the DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

Returning result sets example

Example 4-12 is an example of returning result sets.

Example 4-12 Procedure that returns result sets

```
CREATE PROCEDURE GetCusName()
   RESULT SETS 1 1
   LANGUAGE SQL
BEGIN
     DECLARE GetCusName_c1 CURSOR WITH RETURN FOR 2
     SELECT cusnam FROM customer ORDER BY cusnam;
     OPEN GetCusName_c1; 3
 END
```

Notes about Example 4-12:

1 When a result set returns in your procedure, include the clause RESULT SETS in your CREATE PROCEDURE declaration. RESULT SETS specifies the maximum number of result sets that can be returned from the procedure.

2 This statement declares a cursor for a SELECT statement. The WITH RETURN clause indicates that the cursor is intended for use as a result set from a procedure.

3 You must open the cursor before it can be returned to the caller. The calling procedure is responsible for closing the cursor.

4.7.2 Retrieving result sets in the caller

To consume result sets from a called procedure, you must code the following SQL statements in the calling procedure (caller):

- DECLARE RESULT_SET_LOCATOR
- CALL
- ASSOCIATE LOCATORS
- ALLOCATE CURSOR
- FETCH
- CLOSE

You can get more detailed information about the process of writing a procedure to consume result sets at the following website:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/sqlp/rbafyreceiveresult.htm?lang=en

The following sections provide more detail about the SQL statements that are used to allow consumption of the result sets that are created in a called procedure.

DECLARE... RESULT_SET_LOCATOR

A *result set locator* is a variable that is used to contain the location of a result set that is opened in a called procedure. The calling procedure must declare one or more result locators for each result set that is returned by the called procedure. Each result set locator must have a unique name. Example 4-13 declares two result set locators within a procedure.

Example 4-13 Declaring two result set locators within a procedure

```
DECLARE comresults_l1, comresults_l2 RESULT_SET_LOCATOR VARYING;
```

CALL

The SQL CALL statement is used by the calling procedure to invoke the SQL procedure that contains the result sets to be consumed. The called procedure must use the RETURN TO CALLER returnability attribute. The attribute can be set dynamically by the calling procedure and passed to the called procedure or exported as a global variable, allowing the called procedure to build the cursor as return to caller or return to client. For more information, see 8.2.1, “Global variables as default parameters” on page 232.

Example 4-14 is an example of calling a procedure by using a variable for the procedure name.

Example 4-14 Calling a procedure by using a variable for the procedure name

```
SET v_procname = 'PROC1';
CALL v_procname;
```

Note: When you use a variable, the name must be uppercase.

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement is used to assign the result set locator variables to the procedures that were previously called.

Figure 4-19 contains the syntax diagram for ASSOCIATE LOCATORS.

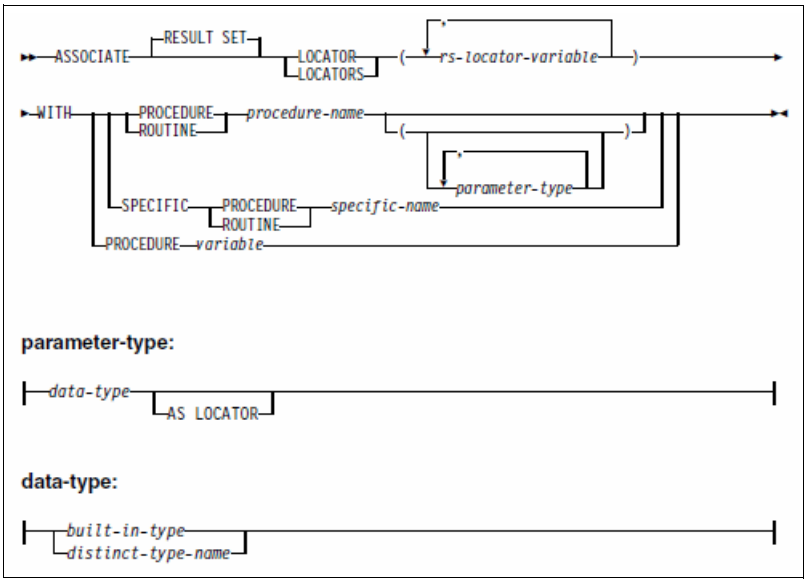

Figure 4-19 ASSOCIATE LOCATORS syntax

The WITH clause identifies a called procedure that can return one or more result sets. The procedure can be identified by its name, specific name, or a variable that contains the procedure name or specific name. The procedure name can be fully qualified. If the procedure name is not qualified, the default schema of the procedure follows the rules for deployment of unqualified database objects. For more information about unqualified object names, see Chapter 3, “SQL fundamentals” on page 47.

The procedure name can be identified by its signature, that is, the procedure name followed by the data types of the parameters that were declared when the procedure was defined. All of the parameter types that are declared must be listed, including parameters that are defined with defaults. In most cases, this method is not used. However, this method is allowed because it might be useful with overloaded procedures.

Example 4-15 contains a code snippet example of associating locators with the results that are produced by calling procedure proc1.

Example 4-15 Code snippet for associating result set locators

```
DECLARE comresults_l1, comresults_l2 RESULT_SET_LOCATOR VARYING;
SET v_procname = 'PROC1';
CALL v_procname;
ASSOCIATE LOCATORS(comresults_l1,comresults_l2) WITH PROCEDURE v_procname;
```

The procedure name, which is defined by the variable *v_procname*, must return two result sets.

ALLOCATE CURSOR

The ALLOCATE CURSOR statement defines a pseudonym for the result set that is associated with a result set locator variable. This pseudo cursor is now available for processing by an SQL FETCH statement.

Figure 4-20 contains the syntax diagram for the SQL ALLOCATE CURSOR statement.

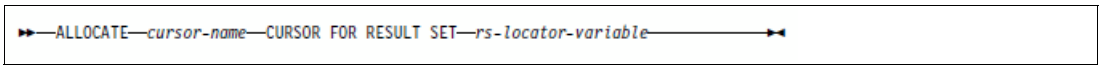

Figure 4-20 ALLOCATE CURSOR statement syntax

Example 4-16 shows an example of the ALLOCATE CURSOR statement.

Example 4-16 ALLOCATE CURSOR statement

```
ALLOCATE comresults_r1 CURSOR FOR RESULT SET comresults_l1;
```

FETCH

The SQL FETCH statement is used to retrieve one or more rows from a result set. The FETCH statement is flexible about consuming the result set because the format of the result set does not matter to FETCH. FETCH merely reads the data from memory and assigns it to the host variables.

Figure 4-21 shows the syntax of the FETCH statement.

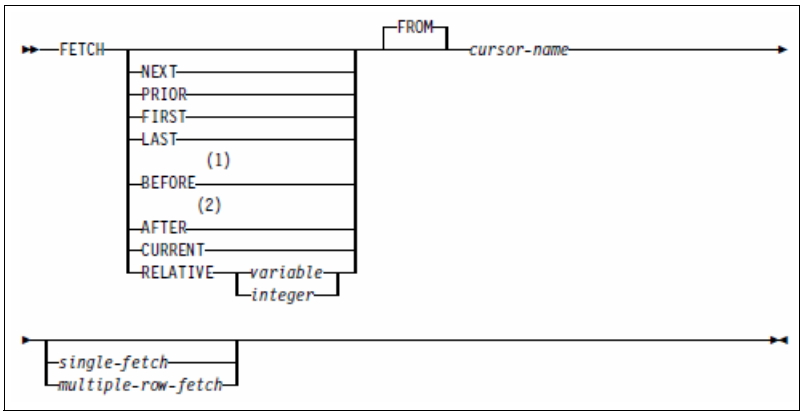

Figure 4-21 FETCH statement syntax

When the result set is allocated, the cursor is positioned either before the first row or on the last row (empty set). FETCH either positions the cursor on the row and assigns the data to variables according to the INTO clause or returns an *SQLSTATE* of '02000'. If the *SQLSTATE* is '00000', the procedure continues to fetch until the last row of the result set is processed.

The cursor name is the pseudo name that is defined by ALLOCATE CURSOR. The FETCH statement can retrieve a single row from the allocated cursor (external procedures only).

Note: The default position keyword **NEXT** is the only keyword that does not require a scrollable cursor.

Fetching a single row from a result set

The INTO clause is used to identify the target storage area for the data in the result set. This target storage area can be a list of variables or an SQL descriptor. Figure 4-22 shows the INTO clause for single fetch.

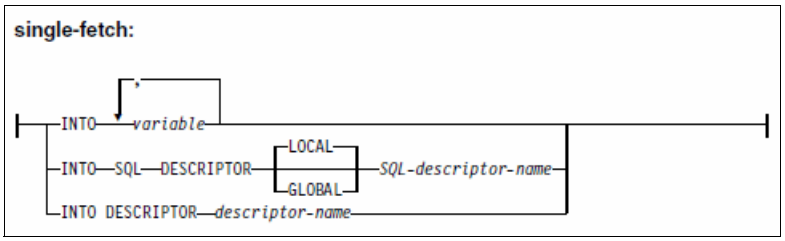

Figure 4-22 INTO clause

The variable and SQL descriptor are described in more detail:

- A variable must be declared within the procedure for each column in the result set. This approach requires one FETCH statement for each result set. This approach is probably the most inflexible way to consume result sets.
- An SQL descriptor provides flexibility because the descriptor can be described dynamically by using the DESCRIBE CURSOR statement. In addition, the descriptor can be allocated globally, allowing the data from the result set to be exported and used by another procedure further up in the call stack.

Example 4-17 contains a code snippet of fetching from an allocated result set cursor into local variables that are defined within the procedure.

Example 4-17 FETCH INTO host variables

```
   DECLARE v_empno CHAR(6);
   DECLARE v_lastname VARCHAR(15);
   DECLARE v_salary DECIMAL(9,2);
   DECLARE SQLSTATE CHAR(5) DEFAULT ‘00000’;                    1
   -- Procedure logic
   FETCH allocated_cursor INTO v_empno, v_lastname, v_salary;   2
   -- Loop through result set
W1:WHILE (SQLSTATE = '00000') DO                                3
-- Do some work
      FETCH allocated_cursor INTO v_empno, v_lastname, v_salary; 4
   END WHILE;
   -- Close cursor when no longer required
   CLOSE allocated_cursor;
```

Notes about Example 4-17:

1 The *SQLSTATE* variable needs to be declared before it can be referenced.

2 A priming FETCH is performed before the WHILE loop. If the result set is empty, the *SQLSTATE* is '02000'.

3 Perform the WHILE loop as long as *SQLSTATE* is '00000'.

4 FETCH the next row in the result set. If no more rows are available, CLOSE the allocated cursor.

Fetching multiple rows

Fetching single rows from a result set is acceptable when the result set is small. Processing large quantities (millions if not billions) of rows by using a single row fetch for retrieval is not a preferred practice.

The ability to fetch more than one row (up to 32,677 rows) from the result set is not supported in the SQL programming language. However, it is supported by using embedded SQL or the SQLextendedFetch function, which is included in the Call Level Interface (CLI) APIs. The process that was described previously (including global descriptors with multiple row fetch) can be coded in your HLL of choice. The called procedure can be an SQL or external procedure that uses embedded SQL.

For more information about embedded SQL, CLI, and multiple row fetch, see the IBM i Knowledge Center:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome

CLOSE

A preferred practice is to always close the SQL cursor after it is fully processed. A CLOSE to an allocated cursor also closes the cursor that was opened in the called procedure. By closing the cursor, you make the cursor available for another result set.

Failure to close the allocated cursor might result in errors the next time that the SQL procedure is called within the same session. In addition, an open cursor is eligible for consumption unless DYNAMIC RESULT SET 0 was explicitly specified.

Due to the simplicity of the CLOSE statement, a syntax diagram is not needed. Example 4-17 on page 97 contains a CLOSE statement.

4.8 Handling errors

Writing and deploying SQL procedures is easy. In fact, many procedures contain SQL statements that can be tested in a stand-alone manner (for example, INSERT, UPDATE, and DELETE). Sometimes, this simplicity can give a developer a false sense of perfection. Like all programs, SQL procedures can, and do, break. It is the developer's responsibility to ensure that errors are handled correctly.

We review how to manage and report database error conditions. The following topics are covered:

- Basic database error indicators
- Handling errors within procedures
- Tailoring error messages

4.8.1 Basic database error indicators

Two variables are used by the database management system (DBMS) to return feedback: *SQLCODE* and *SQLSTATE*. These values are implicitly set after the execution of an SQL statement in an SQL procedure. These values indicate the success or failure of an SQL statement.

To access these values in an SQL procedure, you must define variables with the same name. Example 4-18 shows a code snippet that defines variables with the same name.

Example 4-18 Define variables with the same name

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

When an SQL procedure encounters an error, the *SQLCODE* that is returned is negative, and the first two digits of the *SQLSTATE* differ from '00', '01', and '02'. If SQL encounters a warning while SQL processes the SQL statement, but the warning is a valid condition, 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'.

Example 4-19 shows the code for a simple procedure that inserts new employee data into the employee table.

Example 4-19 Error_handling_basics procedure

```
CREATE OR REPLACE PROCEDURE Error_Handling_Basics (
   IN P_EMPNO CHAR(6),
   IN P_FIRSTNME VARCHAR(12),
   IN P_MIDINIT CHAR(1),
   IN P_LASTNAME VARCHAR(15),
   IN P_WORKDEPT CHAR(3),
   IN P_EDLEVEL SMALLINT,
   IN P_SALARY DECIMAL(9,2)
   )
```

```
P1 : BEGIN ATOMIC
   --Error handling variables
   DECLARE SQLCODE INTEGER DEFAULT 0;
1
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
1
   INSERT INTO employee (empno, firstnme, midinit, lastname, workdept, edlevel,
salary)
    VALUES (P_EMPNO, P_FIRSTNME, P_MIDINIT, P_LASTNAME, P_WORKDEPT, P_EDLEVEL,
P_SALARY) ;

   If SQLCODE <> 0 THEN
2
      RETURN -1;
   END IF;


END P1
```

Notes about Example 4-19 on page 98:

1 The *SQLCODE* and *SQLSTATE* variables are declared.
2 The *SQLCODE* variable is checked for a value less than 0. If true, assign -1 as the return value for the procedure. The use of RETURN is described later in this chapter.

To test the procedure that is shown in Example 4-19 on page 98, an attempt is made to add an employee row with an existing employee number. The procedure fails, and the return value is -803. What? Isn't the value -1 based on the procedure code? And, where did -803 come from?

A review of the error log answers these questions, as shown in Example 4-20.

Example 4-20 Error log

```
{? = call ERROR_HANDLING_BASICS(?,?,?,?,?,?,?)}
Duplicate key value specified.. SQLCODE=-803, SQLSTATE=23505, DRIVER=3.68.61
Run of routine failed.
 - Roll back completed successfully.
```

The value -803 is the *SQLCODE* that corresponds to *SQLSTATE* 23505. So, if the *SQLCODE* is negative, why wasn't the return value -1? And why isn't the error text that is associated with the *SQLSTATE* more explicit? To what key value does the text refer?

The answer to the first question is that whenever an error occurs, the SQL procedure terminates, and control is passed to the calling application, unless a *condition handler* is declared within the failing procedure. We describe this situation in more detail in the next section.

In addition, by handling the error in the procedure, the developer can send a more detailed message to the application. The ability to tailor messages to fit the procedure is described in 4.8.3, "Tailoring error messages" on page 104.

4.8.2 Handling errors within procedures

As shown in Example 4-19 on page 98, certain typical situations require explicit error handling, for example:

- Handling duplicate key value errors that might occur during the execution of an INSERT statement
- Handling constraint violations (for example, referential integrity, constraint, primary, or unique) that might occur during the execution of INSERT, UPDATE, or DELETE statements
- Handling "`row not found`" conditions when a cursor is opened
- Handling "`record not found`" and "`end of the file`" conditions when a FETCH statement is processed

You can handle these and other situations by declaring condition variables and a handler for each condition. You can use the condition declaration to declare a meaningful condition name for a corresponding *SQLSTATE* value.

For a complete list of *SQLCODE* and *SQLSTATE* values, see the topic `SQL messages and codes` in the IBM i Knowledge Center:

`https://ibm.biz/Bd4qv8`

To use a condition name, you need to declare a handler. Figure 4-23 shows the syntax for declaring handlers.

```
handler-declaration

|--DECLARE--+-CONTINUE-+--HANDLER FOR---------------------------->
            +-EXIT-----+
            '-UNDO-----'

                                  (2)
>--+-specific-condition-value-+------SQL-procedure-statement----|
   '-general-condition-value--'
```

Figure 4-23 Handler declaration syntax

A *condition handler* is an SQL statement that is executed when an exception or completion condition occurs within the procedure. 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 a previously declared condition

Three types of condition handlers are available:

- CONTINUE: When this condition is specified, after the SQL statement in the handler is successfully executed, 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 and then the handler is invoked. When the handler is invoked successfully, control is returned to the end of the compound statement that defines the handler.

Important: The UNDO handler can be defined in an ATOMIC compound statement only.

Figure 4-24 contains the syntax for the specific condition value.

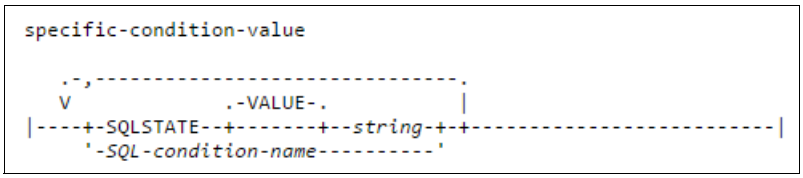

Figure 4-24 Specific condition value syntax

Example 4-21 shows how to code condition declarations.

Example 4-21 Condition declarations for various SQLSTATE codes

```
DECLARE duplicate_key
        CONDITION FOR '23505'
DECLARE record_not_found
        CONDITION FOR '02000';
DECLARE check_constraint_error
       CONDITION FOR '23513';
```

The following details refer to Example 4-21:

- The condition declaration *duplicate_key* corresponds to *SQLCODE* -803, which has an *SQLSTATE* of '23505', as shown in 2.6.2, “Conditions and handlers” on page 31.
- The condition declaration *record_not_found* corresponds to *SQLCODE* +100, which has an *SQLSTATE* of '02000'.
- The condition declaration is called *check_constraint_error*, and it corresponds to *SQLCODE* -545 and to *SQLSTATE* '23513'. This condition occurs when an UPDATE or INSERT violates a check constraint that was defined for one of the columns in the row that is inserted or updated.

Another technique is to declare a condition name for a specific *SQLSTATE*. The condition name is used instead of the *SQLSTATE* value. The procedure is now responsible for handling the error. Consider the code fragment in Example 4-22.

Example 4-22 Declaring a condition name for an SQLSTATE

```
DECLARE DUPLICATE_KEY CONDITION FOR SQLSTATE '23505' ;
DECLARE EXIT HANDLER FOR DUPLICATE_KEY ...
```

Figure 4-25 contains the syntax for the general condition value.

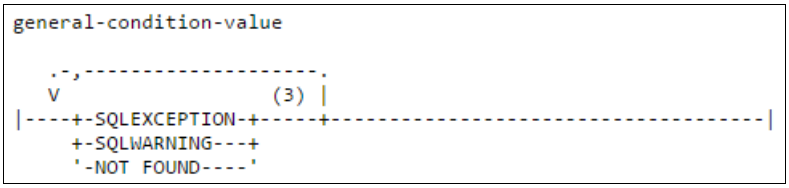

Figure 4-25 General condition value syntax

Three general conditions are associated with different SQLSTATEs:

- SQLEXCEPTION specifies that the handler is invoked when an SQL exception occurs. This condition corresponds to an *SQLSTATE* with a class value other than '00', '01', and '02'. The *SQLSTATE* class is defined by its first two characters.
- SQLWARNING specifies that the handler is invoked when an SQL warning occurs. This condition corresponds to *SQLSTATE* class '01'.
- NOT FOUND specifies that the handler is invoked when a NOT FOUND condition occurs. This condition corresponds to *SQLSTATE* class '02'.

One approach to handling errors is to create a catchall handler by using the EXIT condition. This technique might be useful in an agile environment where the procedure can be put into production quickly.

Example 4-23 contains the modified version of Example 4-19 on page 98 that uses an exit handler.

Example 4-23 Using an exit handler

```
CREATE OR REPLACE PROCEDURE Error_Handling_Basics (
   IN P_EMPNO CHAR(6),
   IN P_FIRSTNME VARCHAR(12),
   IN P_MIDINIT CHAR(1),
   IN P_LASTNAME VARCHAR(15),
   IN P_WORKDEPT CHAR(3),
   IN P_EDLEVEL SMALLINT,
   IN P_SALARY DECIMAL(9,2)
   )

P1 : BEGIN ATOMIC
   --Error handling variables
   DECLARE SQLCODE INTEGER DEFAULT 0;
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
1
     RETURN -1;
2

   INSERT INTO employee (empno, firstnme, midinit, lastname, workdept, edlevel,
salary)
    VALUES (P_EMPNO, P_FIRSTNME, P_MIDINIT, P_LASTNAME, P_WORKDEPT, P_EDLEVEL,
P_SALARY);

END P1
```

In Example 4-23, the exit handler was declared to handle any exception (1). When the exception occurs, a -1 is returned to the calling program (2), and this procedure is terminated.

Example 4-24 shows the use of this technique to obtain the results of executing the procedure with a duplicate empno.

Example 4-24 Results of executing the procedure with a duplicate empno

```
CALL error_handling_basics ( '650302' , 'Im' , 'A' , 'Dupe' , 'D21' , 20 ,
50000.00 )
Return Code = -1
Statement ran successfully
```

We now see that the -1 value is returned to the calling application. However, it now becomes the calling application's responsibility to check the return code after it calls the procedure. Assuming that you are using modular, reusable procedures (as described in Chapter 8, "Creating flexible and reusable procedures" on page 227), this approach might be acceptable.

Not every exception requires the termination of the procedure. In certain cases, the condition that causes the exception can be handled within the procedure in which it occurred. Consider Example A-3 on page 289, which is the allocate_global_descriptor procedure example from Appendix A, "Allocating, describing, and manipulating descriptors" on page 285. The intent of this procedure is to allocate a global descriptor for future reference, but what if a global descriptor was already allocated with that name? This situation is where you use the *continue handler*.

The continue handler allows the procedure to continue to execute when certain types of conditions occur. In the previous scenario, because a global descriptor exists and it is intended for reuse (indicated by the term *global*), the procedure can choose to ignore the condition and continue to follow the statement in error.

Example 4-25 shows the modified version of allocate_global_descriptor.

Example 4-25 Using ALLOCATE GLOBAL DESCRIPTOR

```
CREATE OR REPLACE PROCEDURE Allocate_Global_Descriptor (
    IN p_Global_Descriptor VARCHAR(128))
P1: BEGIN
-- Error handling
1
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
   DECLARE descriptor_already_allocated INTEGER DEFAULT 0 ;
2
   -- Declare condition
   DECLARE c_descriptor_already_allocated CONDITION FOR SQLSTATE '33000' ;
3
   -- Declare handlers
   DECLARE CONTINUE HANDLER FOR c_descriptor_already_allocated
      SET v_descriptor_already_allocated = 1 ;
4
-- Procedure logic begins here
   ALLOCATE SQL DESCRIPTOR GLOBAL p_global_descriptor;
   -- Construct SQL statement
   SET gv_global_descriptor = p_global_descriptor;
5

END P1
```

Notes about Example 4-25 on page 103:

1 The *SQLSTATE* variable is declared to allow explicit reference to the *SQLSTATE* value.

2 The integer variable *v_descriptor_already_allocated* is used in the continue handler and it can be used anywhere within the compound statement that contains the handler.

3 The condition c_descriptor_already_allocated is used as a pseudonym for *SQLSTATE* 33000.

4 When the condition c_descriptor_already_allocated occurs, the variable *v_descriptor_already_allocated* is set to 1. This variable can be used, if necessary, in the procedure logic.

5 The condition is ignored, and the SET *gv_global_descriptor* statement is executed.

4.8.3 Tailoring error messages

When you work with dynamic SQL, it is possible that a single, reusable procedure can execute multiple types of non-select SQL statements (INSERT, UDPATE, DELETE, MERGE, and so on). In this case, it makes more sense for the calling procedure to display the correct message. This section describes the ways in which SQL exception messages can be personalized for each application.

Example 4-26 contains the exit handler code to create a tailored message for the procedure that is shown in Example 4-23 on page 102.

Example 4-26 Exit handler with tailored message

```
DECLARE EXIT HANDLER FOR SQLSTATE '23505'                                    1
   BEGIN
      SET v_message_text = p_empno CONCAT ' is a duplicate employee number'; 2
      SIGNAL SQLSTATE '70001'                                                3
          SET MESSAGE_TEXT= v_message_text;                                  4
 END;
```

Notes about Example 4-26:

1 The exit handler is declared for SQL state '23505', which is the generic message “Duplicate key value specified”.

2 The p_empno value is concatenated to a text string to create a tailored message for the application.

3 A non-SQL state code is used to identify that this message is tailored for the application.

4 The MESSAGE_TEXT parameter of the SIGNAL statement is assigned the value of variable *v_message_text*.

Example 4-27 contains the results of calling the modified version of the error_handling_basics procedure that was described in Example 4-23 on page 102.

Example 4-27 Results of calling a procedure with tailored messages

```
CALL error_handling_advanced ('650301','Ima','','Dupe','D21',25,75000.00 )
-- Job log information
SQL State: 70001
Vendor Code: -438
Message: [SQL0438] 650301 is a duplicate employee number
```

The job log information shows that the SQL state and message match the SQL state and message that are provided by the procedure exit handler. The vendor code -438 is provided by DB2.

4.9 Summary

This chapter covered the following topics:

- Introduction to procedures
- Structure of a procedure
- Creating a procedure
- System catalog tables for procedures
- Procedure signature and procedure overloading
- Calling a procedure
- Producing and consuming result sets
- Handling errors
- Summary

In most cases, after a CREATE PROCEDURE statement completes successfully, the SQL procedure runs. Of course, extensive work is performed before the procedure is ready. This work includes testing, debugging, development, and deployment strategies, which are described in later chapters.

This book will explain how a DBE can take advantage of procedures to develop a data access layer that consists of reusable modules that are called from all applications that manipulate database access.

5

Triggers

Triggers are a powerful part of IBM DB2 for i. They are not only useful to enforce complex business rules, but they are also important to track database activity.

This chapter includes the following topics:

- Trigger concepts
- Trigger types
- Introduction to triggers
- Defining triggers
- Trigger examples
- Additional trigger considerations
- Trigger-related catalogs

5.1 Trigger concepts

The concept of a trigger is relatively simple. A trigger defines a set of actions that are executed automatically by the database whenever a delete, an insert, or an update operation occurs on a specified table or view. When the operation is executed, the trigger is said to be activated. It is also common to refer to a trigger as "*fired*", but this chapter uses the terms *activated* or *invoked*.

Triggers represent a powerful extension to conventional application programming. Although they are written by the application programmer, they are called directly by the database rather than directly by application programs. They have the following properties:

- Triggers are activated regardless of the database interface that is used. Triggers include Structured Query Language (SQL) operations, such as DELETE, INSERT, and UPDATE, in addition to record-level access operations in high-level language (HLL) programs, such as WRITE and UPDATE.
- Triggers are activated regardless of the user interface.
- No changes are required to the application programs or to the user's behavior. In many cases, they are not aware that the trigger exists.

These properties provide important functional benefits:

- Implementation of business rules:
  - Triggers can be used to consistently apply business rules to any database operations from any application, user, or user interface.
  - Business rules can be implemented in a single place rather than in potentially many application programs, which simplifies both initial development and future changes to the business rules. If a rule is changed in a trigger, all database access transparently and automatically complies to the new rules.
  - Triggers can implement business rules that reflect both the current and future column values. For example, a business rule that salary cannot be increased by more than 10% must compare the current value with the new salary value.
  - Triggers can be used to validate data that is placed in a database table. They can be configured to intercept and modify the data before it goes into the database.
  - Triggers can also implement business rules that span multiple tables. For example, a trigger can check a customer's credit rating in a credit history table before it allows any new orders in the order table.
- Improved database consistency across tables

  Triggers can ensure that data is consistently reflected across multiple database tables by reflecting changes in one table to other tables.
- Improved auditing and controls:
  - Triggers make it more difficult for a user or even a knowledgeable database engineer (DBE) to make an unauthorized or inappropriate change in the database.
  - Triggers can apply a layer of auditing and report database operations that might require additional investigation.
- Improved integration of existing applications and non-database functionality

  Triggers can perform non-database functions, such as sending a confirmation text message, sending a message to the system operator, or sending an email.

- Extend the lifespan of existing applications

  Internal and third-party applications might pass out of service. Triggers provide a way of extending their lifespan by performing additional actions when table changes occur without any change to the application.

- Triggers can be cascaded.

  For example, a trigger, which is defined on table1, can change table2, which results in the activation of any triggers on table2. This powerful capability can result in many changes to the database based on a single change in a single table.

- Triggers have a unique value because they allow insert, update, and delete operations against the view instead of directly to the underlying tables.

Triggers need to be part of your overall database design with careful attention to which events on which tables can take advantage of their many benefits. Think of triggers in terms of the broader environment rather than merely their relationship to the needs of a specific application. Consider the potential performance impacts of implementing triggers on tables with many insert, update, and delete operations.

Note: Triggers complement the functionality that is provided by referential integrity or check constraints. Triggers must not be used to implement business rules that can instead be implemented as constraints. Unlike constraints, triggers do not enforce business rules against data that is already in the database.

Triggers represent a powerful technique to ensure that your database always complies with your business needs. Triggers provide consistent checking and the correct actions every time that data changes.

5.2 Trigger types

DB2 for i supports two types of triggers: SQL triggers and external triggers.

5.2.1 SQL triggers

For an SQL trigger, the program that performs the tests and actions is written by using the SQL programming language. The SQL CREATE TRIGGER and ALTER TRIGGER statements define how the database management system will actively control, monitor, and manage a group of tables whenever an insert, an update, or a delete operation is performed. An SQL trigger can call stored procedures or user-defined functions (UDFs) to perform additional processing when the trigger is activated.

Like all triggers, SQL triggers are never called directly. Instead, an SQL trigger is invoked by the database manager upon the execution of a triggering INSERT, UPDATE, or DELETE operation on the subject table or view.

SQL triggers offer significant advantages over traditional external triggers:

- SQL syntax is more powerful so SQL triggers can be created more quickly and easily.
- SQL triggers support column-level granularity so they can be activated only when a column or set of columns are updated.
- SQL triggers offer more options to manage operations that modify more than one row in the subject table or view.

- SQL triggers include “INSTEAD OF” triggers for views that are not available for external triggers.
- SQL syntax makes it easier to write and configure one trigger to process multiple events.
- SQL triggers are more portable.

5.2.2 External triggers

For an external trigger, the program that contains the set of trigger actions can be defined in any supported HLL that creates a *PGM object. SQL can be embedded in the trigger program. External triggers are managed by using control language (CL) commands, such as Add Physical File Trigger (**ADDPFTRG**).

This book covers SQL triggers. For information about external triggers, see *External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i*, SG24-6503.

5.3 Introduction to triggers

The implementation of SQL triggers is based on the SQL standard that supports constructs that are common to most programming languages. These constructs include the declaration of local variables and constants, statements to control the flow of the procedure, assignment of expression results to variables, and error handling. The SQL standard allows the capability to easily embed SQL statements in a program. For more information, see Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5.

The SQL CREATE TRIGGER statement is similar to the support in other database products, which provides greater portability.

This section provides a high-level view of the syntax to create a trigger and several simple examples. The options are described in more detail after the examples. Example 5-1 shows a high-level summary of the syntax for the CREATE TRIGGER statement.

Example 5-1 CREATE TRIGGER syntax summary

```
CREATE TRIGGER trigger-name                          1
   trigger-activation-time                           2
   trigger-event ON subject-table                    3
   transition-variables / transition-table           4
   trigger-granularity                               5
   trigger-mode                                      6
   triggered-action                                  7
```

Notes about Example 5-1 on page 110:

1. Every trigger creation statement starts with CREATE TRIGGER and the trigger name. Either the name must not match the name of an existing trigger in the schema, or CREATE OR REPLACE TRIGGER can be specified to drop the existing trigger when the new trigger is created.
2. The trigger activation time indicates when the trigger will be activated relative to the trigger event. It can be BEFORE, AFTER, or INSTEAD OF.
3. The trigger event is the operation that results in the activation of the trigger. It can be INSERT, UPDATE, or DELETE. Multiple events can be specified by using the keyword **OR**, for example, INSERT OR DELETE. The *subject table* is the table or view for which the trigger is defined.
4. *Transition variables* or *transition tables* provide the trigger with the before and after values for any columns and rows that are involved in the trigger event.
5. Trigger granularity can be either FOR EACH STATEMENT or FOR EACH ROW and controls whether the triggered action is executed for each row or for each statement that was changed. For example, an UPDATE operation can update seven rows, and the trigger is activated either one time or seven times, depending on this attribute.
6. Trigger mode MODE DB2ROW specifies that the trigger will be activated after each row operation. MODE DB2SQL activates the trigger after all row operations.
7. The triggered action contains the action to execute when the trigger is activated. It can be simple or complex. It optionally starts with a WHEN clause, which can contain any true/false condition.

The following examples are simple. Example 5-2 shows the CREATE TRIGGER statement for a trigger that is named new_hire. After the INSERT operation on the employee table, the trigger increments the count of employees in the table company_stats. In this case, the trigger body consists of a single statement so the **BEGIN** and **END** keywords are optional.

Example 5-2 Using a trigger to manage the employee count

```
CREATE TRIGGER new_hire
AFTER INSERT ON employee
FOR EACH ROW MODE DB2ROW
UPDATE company_stats SET empcount = empcount + 1;
```

Example 5-3 shows the SQL statement to create the equivalent trigger by using the **BEGIN** and **END** keywords.

Example 5-3 Using a trigger to manage the employee count, including BEGIN/END

```
CREATE or replace TRIGGER new_hire
AFTER INSERT ON employee
FOR EACH ROW MODE DB2ROW
BEGIN
  UPDATE company_stats SET empcount = empcount + 1;
END;
```

The interactions between the trigger definition attributes can be complex. First, we explain the syntax for the statement, followed by examples that demonstrate how the attributes interact.

5.4 Defining triggers

Next, the syntax is described for the SQL CREATE TRIGGER and ALTER TRIGGER statements. The syntax is explained step-by-step with a description after each part of the syntax diagram. This information is taken directly from the DB2 for i SQL reference, which often contains additional footnotes about syntax. The DB2 for i SQL reference is available at this website:

https://ibm.biz/Bd42dh

Note: Various development tools can provide wizards or templates to facilitate the process of creating triggers.

Figure 5-1 shows the first part of the CREATE TRIGGER syntax, which starts with CREATE TRIGGER and the name of the trigger.

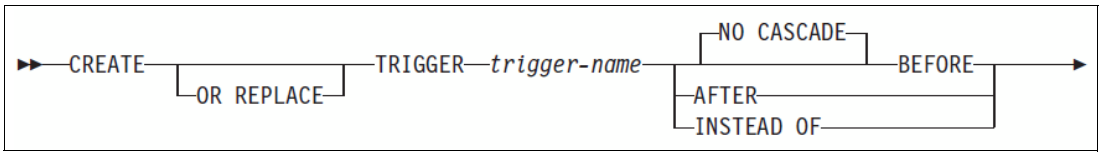

Figure 5-1 CREATE TRIGGER syntax to create the trigger and define the trigger activation time

Trigger names follow these rules:

- The trigger name can be either schema-qualified or unqualified.
- Triggers cannot be schema-qualified with QTEMP.
- Each trigger must have a unique name within the schema.
- Where the trigger is created depends on the naming convention:
  - Under SQL naming, the trigger is created in the schema that is specified by the implicit or explicit qualifier. For more information about the explicit qualifier, see Chapter 3, “SQL fundamentals” on page 47.
  - Under system naming, the trigger is created in the schema that is specified as the qualifier. If the schema is not qualified, the trigger is created in the same schema as the subject table or view.

The optional OR REPLACE clause can be specified to drop an existing trigger when the new trigger is created.

The next clause corresponds to the trigger activation time and indicates when to activate the trigger:

- BEFORE specifies that the trigger is activated before the action that triggers it, for example, an INSERT operation. The following attributes apply to BEFORE triggers:
  - BEFORE triggers can verify and modify column values that are inserted or updated into the table.
  - BEFORE triggers can prevent changes by signaling an error.
  - BEFORE triggers cannot be specified if the subject table is an SQL view.
  - Historically, BEFORE triggers did not allow changes to other tables, which ensured that no additional triggers were activated. This restriction appears in older versions of DB2 for i documentation but it was removed in IBM i 7.1 and later releases. The NO CASCADE clause is associated with this old behavior and ignored on DB2 for i.
- AFTER specifies that the trigger is activated after the changes, which are caused by an INSERT, a DELETE, or an UPDATE operation, are applied to the subject table. AFTER cannot be specified if the subject table is an SQL view.
- INSTEAD OF indicates that the trigger is executed instead of the triggering event so that the triggered action effectively takes the place of the INSERT, UPDATE, or DELETE operations. Special restrictions apply to this trigger:
  - INSTEAD OF is allowed only when the subject table of the triggering event is an SQL view. They cannot be created for data description specifications (DDS) logical files.
  - Only one INSTEAD OF trigger is allowed for each kind of operation on a certain subject view.
  - The triggered action cannot contain the WHEN clause.

Figure 5-2 shows the placement of the trigger event clause and the ON clause that identifies the table or view that is defined as the subject table for the trigger.

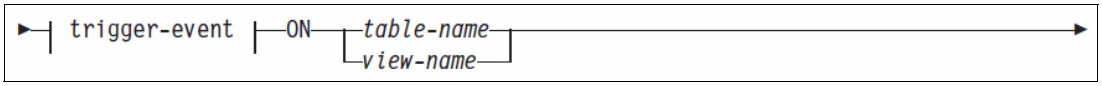

Figure 5-2 CREATE TRIGGER syntax to trigger the event and show the subject table or view

The syntax in Figure 5-2 has these restrictions:

- If it is a table name, the table must exist and it cannot be a catalog table, a table in the `QTEMP` library, or a declared temporary table.
- If it is a view name, the view must exist and it cannot be a catalog or a view in the library `QTEMP`.

A view name must not specify a view that is defined by using WITH CHECK OPTION or a view on which a WITH CHECK OPTION view is defined, either directly or indirectly. On CREATE VIEW, the CHECK OPTION clause controls whether a row that is inserted or updated in a view needs to conform to the definition of the view. A row that does not conform to the view definition cannot be read by using the view. This situation can occur when the view contains a WHERE clause for filtering. For more information, see the `CREATE VIEW` documentation in the DB2 for i SQL reference, which is available at this website:

`https://ibm.biz/Bd42dh`

Figure 5-3 show the details of the trigger event clause.

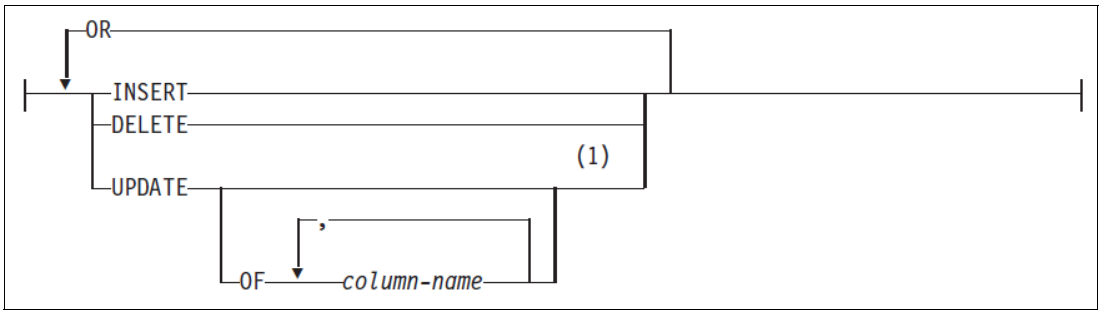

Figure 5-3 CREATE TRIGGER syntax to trigger event details

The trigger event can be one of three possible values:

- INSERT specifies that the trigger will be activated on an insert operation on the subject table or view.
- DELETE specifies that trigger will be activated on a delete operation on the subject table or view. A DELETE trigger cannot be added to a table with a referential constraint of ON DELETE CASCADE. That is, the subject table might have a foreign key constraint that is defined with a CASCADE rule so that the row is deleted if the row that contains the parent key value, which can be either a primary or unique key constraint, is deleted. No DELETE trigger is allowed in this case.
- UPDATE specifies that the trigger will be activated on an update operation on the subject table or view. UPDATE also has an optional clause for column-level granularity that indicates that the trigger is activated only if a specific column or specific columns are included in the update operation on the subject table. The trigger is still activated when the column is updated even if the value of that column does not change. If no column names are specified, the trigger is activated for update of any column in the subject table. If no column names are specified, the trigger is activated for update of any column in the subject table. It has these restrictions:
  - This column granularity is not allowed for an INSTEAD OF trigger.
  - An UPDATE trigger event cannot be added to a table with a referential constraint of ON DELETE SET NULL or ON DELETE SET DEFAULT.

Note: A single trigger can process multiple events so it is possible to specify more than one event by using the **OR** keyword. The (1) indicates a footnote in the DB2 for i SQL reference that states that each trigger event can be specified one time only.

Figure 5-4 shows the syntax for correlation names (transition variables) and transition tables.

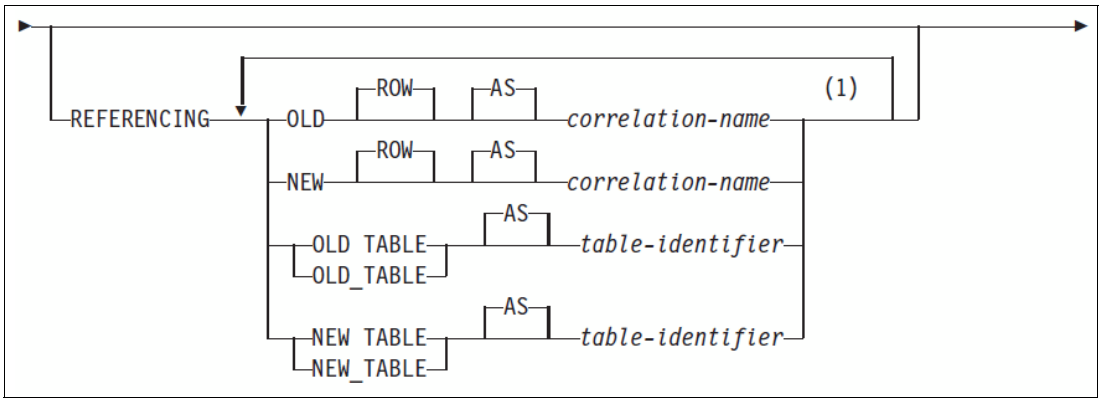

Figure 5-4 CREATE TRIGGER syntax for the correlation names and transition tables

Figure 5-4 shows an optional clause and allows access to the old values and the new values of any columns and rows that are involved in the trigger event. The keyword OLD reflects the current values and NEW reflects the new values. The availability of OLD and NEW values depends on the triggering event:

- INSERT operations have NEW values only.
- UPDATE operations have both OLD and NEW values.
- DELETE operations have OLD values only.

OLD ROW and NEW ROW are used to provide a correlation name, which can be used to refer to the column values in the old and new rows. The syntax is flexible because the keywords **ROW** and **AS** are optional. In an INSERT, these column values are null for the OLD row. For a DELETE, these column values are null for the NEW row.

OLD TABLE and NEW TABLE refer to virtual (rather than physical) tables that contain the old and new row values. The syntax is flexible because it allows underscores between the keywords OLD and TABLE, for example. The keyword AS is also optional. For INSERT, the old table is empty. For DELETE, the new table is empty.

Both the correlation names and the transition table names have a maximum length of 128 characters.

The following restrictions apply to these clauses:

- The same clause cannot be specified more than one time, which was indicated in the DB2 for i SQL reference (original document) in footnote (1).
- All of the correlation names and table identifiers must be unique within the trigger.

The availability of the row and table images relates to the triggering event, trigger granularity, and trigger mode, which is shown in Figure 5-5.

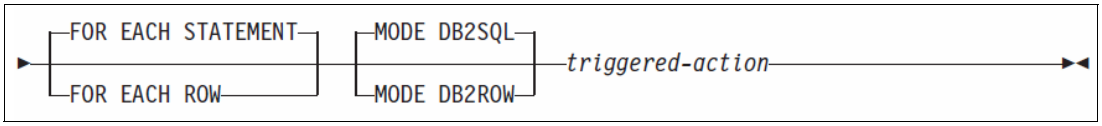

Figure 5-5 CREATE TRIGGER syntax for trigger granularity and trigger mode

Figure 5-5 on page 115 shows the syntax for the trigger granularity and trigger mode. Trigger granularity defaults to FOR EACH STATEMENT, which indicates that the trigger is activated one time for the statement even if the trigger action results in changes to multiple rows. For example, an UPDATE statement can change values in seven rows but a trigger that specifies FOR EACH STATEMENT is activated one time only. The trigger is also activated if no rows are changed by the statement. FOR EACH STATEMENT cannot be specified on a BEFORE trigger. It also cannot be specified on a MODE DB2ROW trigger.

A trigger that is created with FOR EACH ROW is activated one time for each row that is changed. If no rows are modified, it is not activated.

Note: In an UPDATE trigger, the trigger is still activated when the column is part of the update operation even if the old and new values are the same. That is, if the salary column in the employee table contains an old value of 20000 and it is set to a matching value of 20000, a trigger that specifies FOR EACH ROW is still activated.

DB2 for i supports two trigger modes: MODE DB2SQL and MODE DB2ROW. The existence of these modes is a frequent source of confusion but they exist to address different implementations within the DB2 family. DB2 for i already implemented triggers for traditional record-level access by using the Add Physical File Trigger (**ADDPFTRG**) CL command and application programs. For that reason, DB2 for i wanted to provide SQL triggers that were compatible with traditional triggers but the DB2 family also needed different behaviors. DB2 for i supports both trigger modes.

The trigger mode defaults to MODE DB2SQL because DB2SQL is compatible with the other DB2 implementations:

- MODE DB2SQL BEFORE triggers are activated on each row operation. MODE DB2SQL is valid for BEFORE triggers only if a REFERENCING clause is not specified and the subject table is not referenced in the triggered action.
- MODE DB2SQL is valid for AFTER triggers, and these triggers are activated after all of the row operations occur. This mode is less efficient because each row is processed twice. For example, the database processes updates to seven rows and then must go back to logically revisit those rows when the trigger is activated.

MODE DB2ROW triggers are activated on each row operation, which is consistent with the implementation of traditional record-level access triggers:

- This mode is valid for both BEFORE and AFTER triggers.
- DB2ROW cannot be specified for triggers that specify FOR EACH STATEMENT. Its behavior is similar to traditional external triggers.

The relationships among the trigger granularity, trigger mode, and trigger activation time are often confusing.

Table 5-1 summarizes the combinations of these attributes and their impact on the trigger's activation.

Table 5-1 Trigger granularity, trigger mode, and trigger activation time

Trigger granularity	Trigger mode	Trigger activation time	Trigger activation behavior
FOR EACH ROW	MODE DB2SQL	BEFORE	Trigger is activated for each row before each row is modified (converted to MODE DB2ROW). Subject table cannot be referenced.
		AFTER	N/A
		INSTEAD OF	Trigger is activated for each row.
	MODE DB2ROW	BEFORE	Trigger is activated for each row before each row is modified.
		AFTER	Trigger is activated for each row after each row is modified.
		INSTEAD OF	Trigger is activated for each row.
FOR EACH STATEMENT	MODE DB2SQL	BEFORE	N/A
		AFTER	Trigger is activated one time after all row changes.
		INSTEAD OF	Trigger is activated for each statement.
	MODEDB2ROW	BEFORE	N/A
		AFTER	N/A
		INSTEAD OF	N/A

If you specify MODE DB2SQL on a BEFORE trigger, it is automatically converted to MODE DB2ROW. This trigger mode conversion is reported in the job log as an SQL7051 warning, as shown in Figure 5-6.

```
SQL State: 01647
Vendor Code: 7051
MODE DB2SQL before trigger converted to MODE DB2ROW.
Cause: MODE DB2SQL before triggers are not supported.  The SQL trigger MODECHG in JIMD
will be converted from MODE DB2SQL to MODE DB2ROW.
```

Figure 5-6 Message that indicates the trigger mode conversion

The availability of transition variables and transition tables is a function of the trigger mode, the trigger activation time, and the triggering operation, which is shown in the following tables. Table 5-2 shows the availability of transition values for a trigger that is defined to be activated FOR EACH ROW.

Table 5-2 CREATE TRIGGER transition values FOR EACH ROW triggers

MODE	Activation time	Triggering operation	Correlation variables allowed	Translation tables allowed
DB2ROW	BEFORE	DELETE	OLD	NONE
		INSERT	NEW	
		UPDATE	OLD, NEW	
	AFTER or INSTEAD OF	DELETE	OLD	
		INSERT	NEW	
		UPDATE	OLD, NEW	
DB2SQL	BEFORE	DELETE	OLD	
		INSERT	NEW	
		UPDATE	OLD, NEW	
	AFTER or INSTEAD OF	DELETE	OLD	OLD TABLE
		INSERT	NEW	NEW TABLE
		UPDATE	OLD, NEW	OLD TABLE, NEW TABLE

Table 5-3 shows the availability of transition values for a trigger that is defined to be activated FOR EACH STATEMENT. This table is short because FOR EACH STATEMENT triggers cannot specify a mode of DB2ROW and cannot be BEFORE triggers.

Table 5-3 CREATE TRIGGER transition values FOR EACH STATEMENT triggers

Mode	Activation time	Triggering operation	Correlation variables allowed	Transition tables allowed
DB2SQL	AFTER or INSTEAD OF	DELETE	NONE	OLD TABLE
		INSERT		NEW TABLE
		UPDATE		OLD TABLE, NEW TABLE

Figure 5-7 shows the syntax for the trigger action.

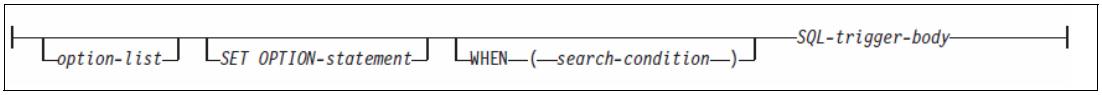

Figure 5-7 CREATE TRIGGER syntax for the trigger action

The option list is covered next. SET OPTION is optional. It is described in “SET OPTION” on page 57.

The SET OPTION clause has specific restrictions for triggers:

- The following options are not allowed in the CREATE TRIGGER statement:
  - CNULIGN
  - CNULRQD
  - COMPILEOPT
  - NAMING
  - SQLCA

  Support for EXTIND was recently added by program temporary fix (PTF).
- The COMMIT option is allowed, but ignored.
- The USRPRF and DYNUSRPRF options are ignored. Triggers always run under the owner's authority.
- The options DATFMT, DATSEP, TIMFMT, and TIMSEP cannot be used if OLD ROW or NEW ROW is specified.

The optional WHEN clause can be used to specify a condition that evaluates to true, false, or unknown. The SQL statements in the trigger body are executed only if this condition evaluates as true (often used on an UPDATE trigger to check whether a column value changed). Compound conditions can be specified with AND and OR operators. A WHEN clause cannot be specified on an INSTEAD OF trigger.

The SQL trigger body specifies a single SQL procedure statement, which might consist of a compound statement that starts with the **BEGIN** keyword and ends with the **END** keyword. For more information, see Chapter 2, "Introduction to SQL Persistent Stored Module" on page 5.

Specific restrictions apply to triggers:

- Triggers cannot contain statements or call procedures that modify the connection for the current activation group, including issuing a CONNECT, SET CONNECTION, RELEASE, or DISCONNECT. This restriction also applies to the use of implicit or explicit three-part name support.

  The statements in the triggered-action can invoke a procedure or a UDF that can access a server other than the current server if the procedure or UDF runs in a different activation group.
- Triggers cannot contain call procedures that modify the isolation level or transactional integrity, including COMMIT, ROLLBACK, and SET TRANSACTION.

  Triggers cannot contain COMMIT or ROLLBACK. They can use SET TRANSACTION to manage the isolation level. They can also specify ATOMIC on the BEGIN statement to ensure that the statements that are executed within the compound statement are either committed or rolled back. For more information, see Chapter 2, "Introduction to SQL Persistent Stored Module" on page 5.
- Triggers cannot use an UNDO handler. For more information about handlers, see Chapter 2, "Introduction to SQL Persistent Stored Module" on page 5.
- All tables, views, aliases, distinct types, global variables, UDFs, sequences, and procedures that are referenced in the trigger body must exist at the current server when the trigger is created. The table or view that an alias refers to must also exist when the trigger is created.

  This rule includes objects in library `QTEMP`. Although objects in `QTEMP` can be referenced in the trigger body, dropping those objects in `QTEMP` does drop the trigger. If the trigger is activated and the object in `QTEMP` is not re-created, the trigger fails with an error that indicates that the object was not found. Also, the trigger fails if a new instance of the object is created but its column definitions are incompatible with the original instance.

- All transition variable names are column names of the subject table or view. System column names of the subject table or view cannot be used as transition variable names.
- The trigger body of a BEFORE trigger on a column of type XML can invoke the XMLVALIDATE function through a SET statement, leave values of type XML unchanged, or assign them to NULL by using a SET statement.

One additional restriction, which is not listed in the DB2 for i SQL reference, is documented in the SQL0751 message: "`The RETURN statement is not allowed in an SQL trigger`."

Figure 5-8 provides the details for the option list that is referenced in Figure 5-7 on page 118.

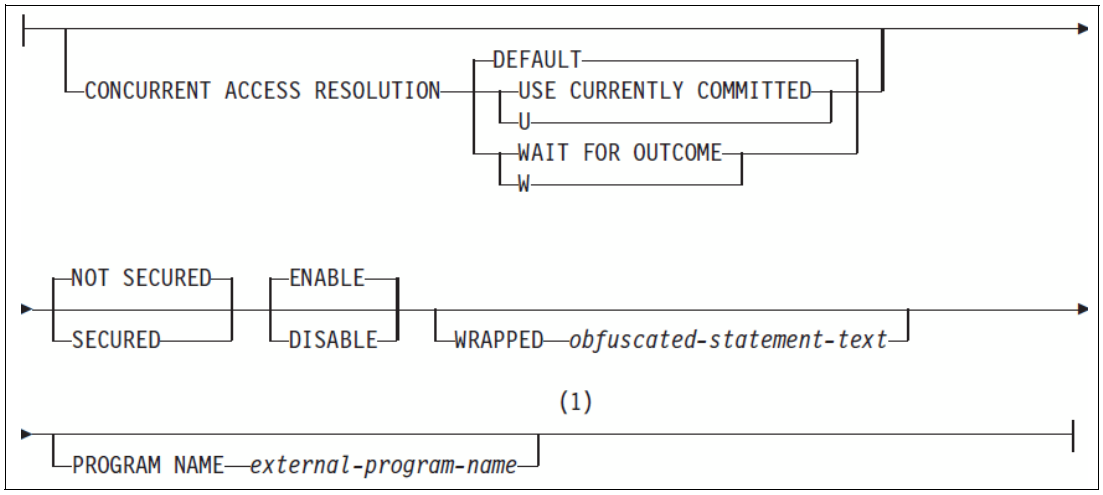

Figure 5-8 CREATE TRIGGER syntax option list details

CONCURRENT ACCESS RESOLUTION is explained in "CONCURRENT ACCESS RESOLUTION" on page 60.

The SECURED or NOT SECURED attribute is new for IBM i 7.2. It specifies whether the trigger is considered secure for row and column access control (RCAC). A trigger must be secure if its subject table uses row access control and column access control.

SECURED must also be specified for a trigger that is created for a view and one or more of the tables that are referenced in the view definition use activated row access control or column access control.

If a secure trigger references UDFs, DB2 assumes that those functions are secure without validating that they have the SECURED attribute. For more information about RCAC, see "NOT SECURED and SECURED" on page 61.

ENABLE or DISABLE determines whether the trigger was enabled at the time that it was created. This option defaults to ENABLE. This attribute can be modified later by using the ALTER TRIGGER SQL statement, which is described in Figure 5-9 on page 121.

WRAPPED is an optional clause. It corresponds to the DB2 for i support for creating a scrambled version of SQL statement source, which makes it more difficult for someone other than the writer to extract any intellectual property that is contained in the source. This support was added for SQL triggers in IBM i 7.2. For more information, see "WRAPPED" on page 62.

The (1) in Figure 5-8 is associated with a footnote in the DB2 for i SQL reference that indicates that these options can be specified in any order.

The optional PROGRAM NAME clause is used to specify the name of the program object that was generated by DB2 for i to implement the trigger's functionality. This clause is explained in 3.2.1, "Routine and trigger creation process" on page 55.

Figure 5-9 shows the syntax for the ALTER TRIGGER statement.

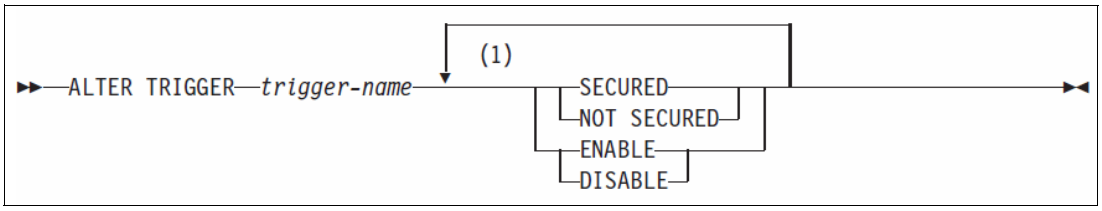

Figure 5-9 ALTER TRIGGER syntax

Figure 5-9 shows the syntax for the ALTER TRIGGER statement. ALTER TRIGGER can be used only to change the SECURED/NOT SECURED attribute and whether the trigger needs to be enabled or disabled. The (1) corresponds to a footnote in the DB2 for i SQL reference that indicates that each option can be specified one time only.

5.5 Trigger examples

This section contains examples of triggers that highlight their power and flexibility. The following examples are included:

- Simple trigger examples
- Use of correlation names for column values
- Multiple event triggers
- Changing row values in a BEFORE trigger
- Calling a procedure from a trigger
- Using transition tables
- Signaling an error
- Self-referencing triggers
- DB2ROW versus DB2SQL triggers
- INSTEAD OF triggers

The examples are based on the sample schema that was created by the CREATE_SQL_SAMPLE procedure that is provided by IBM. The examples are cumulative and build on the concepts that are demonstrated in previous examples. Because of the referential integrity rules for SQL triggers, it is necessary to change the sample schema to allow the creation of triggers on the employee table.

The ALTER TABLE statements are shown in Example 5-4.

Example 5-4 Schema changes that are needed for the sample schema

```
ALTER TABLE EMPLOYEE DROP CONSTRAINT RED ;
ALTER TABLE EMPLOYEE
      ADD CONSTRAINT RED
                FOREIGN KEY( WORKDEPT )
                REFERENCES DEPARTMENT ( DEPTNO )
         ON DELETE NO ACTION
         ON UPDATE NO ACTION ;
```

5.5.1 Simple trigger examples

The following trigger examples appeared earlier in this book. Now, we provide additional notes about their functions.

Example 5-5 shows an example that uses a trigger that is named new_hire. After an INSERT operation on the employee table, it increments the count of employees in the table company_stats. The trigger body consists of a single statement.

Example 5-5 Using a trigger to manage the employee count

```
CREATE TRIGGER new_hire
AFTER INSERT ON employee
FOR EACH ROW MODE DB2ROW
UPDATE company_stats SET empcount = empcount + 1;                          1

SELECT empcount FROM company_stats;                                        2

INSERT INTO employee (empno, firstnme, midinit, lastname, workdept, phoneno,
   hiredate, job, edlevel, sex, birthdate, salary, bonus, comm )
VALUES ( 100501, 'DAN', 'P', 'ANDERSON', 'A00', 5541,
   '1992-06-09', 'Analyst', 16, 'M', '1974-07-01',  27600, 400, 1922 );     3

SELECT empcount FROM company_stats;                                        4
```

Notes about Example 5-5 on page 122:

1 The trigger new_hire is created to be activated after INSERT operations on the employee table. It is a row-level trigger, and it specifies mode DB2ROW so it will be activated for each row that is inserted. Each time that it is activated, it will update the company_stats table and increment the column empcount by one.

2 This select statement can be used to verify the current value of the employee count. For the sample database, it returns the value 42.

3 The INSERT statement inserts a row for a new employee that is named Dan P. Anderson with all of the information about departments, phone number, and so on.

4 This select statement can be used again. This time, it returns the value 43, which indicates that the company_stats table was updated by the trigger.

Example 5-6 shows the SQL statement to create the equivalent trigger by using the **BEGIN** and **END** keywords.

Example 5-6 Using a trigger that includes BEGIN and END to manage the employee count

```
CREATE or replace TRIGGER new_hire
AFTER INSERT ON employee
FOR EACH ROW MODE DB2ROW
BEGIN                                                          1
  UPDATE company_stats SET empcount = empcount + 1;
END;                                                           1
```

Note about Example 5-6:

1 The **BEGIN** and **END** keywords are used to indicate the start and end of the triggered action.

5.5.2 Use of correlation names for column values

Enforcing business rules often requires checking both the old and new values for columns in the subject table. *Old values exist for UPDATE or DELETE events. New values exist for UPDATE and INSERT events only.*

The trigger that is shown in Example 5-7 uses the optional REFERENCING clause so it has access to the old and new values.

Example 5-7 Using a trigger to audit salary increases

```
CREATE OR REPLACE TRIGGER audit_salary
AFTER UPDATE ON employee
REFERENCING NEW ROW AS n OLD ROW AS o
FOR EACH ROW MODE DB2ROW                                                    1
WHEN (n.salary<>o.salary)                                                   2
BEGIN
    IF n.salary > o.salary THEN
        INSERT INTO salary_audit (empno, old_salary, new_salary, change_user, change_time)
            VALUES (o.empno, o.salary, n.salary, USER, CURRENT_TIMESTAMP);   3
    END IF;
END;
```

Notes about Example 5-7 on page 123:

1 The trigger audit_salary is defined so that it is activated after UPDATE operations on the employee table. It is a row-level trigger and specifies mode DB2ROW so that it will be activated as each row is modified. The definition uses the REFERENCING clause so that it can refer to the old and new column values for the row that is updated. The `n` is used as the correlation name for new values and the `o` is used as the correlation for old values.

2 The optional WHEN clause is used so that the body of the trigger is activated only when the new salary differs from the old salary.

3 If the new salary value is greater than the old salary value, a row is inserted into a table that is named salary_audit to record the employee number that is updated, the old and new salary values, the user that made the change, and the time stamp of the change.

The definition of the query can be changed so it is invoked only when the salary value changes. The revised syntax is shown in Example 5-8.

Example 5-8 Using a trigger to audit salary increases with OF clause

```
CREATE OR REPLACE TRIGGER audit_salary
AFTER UPDATE OF salary ON employee                                          1
REFERENCING NEW ROW AS n OLD ROW AS o
FOR EACH ROW MODE DB2ROW
WHEN (n.salary<>o.salary)                                                   2
BEGIN
    IF n.salary > o.salary THEN
        INSERT INTO salary_audit
             (empno, old_salary, new_salary, change_user, change_time)
          VALUES (o.empno, o.salary, n.salary, USER, CURRENT_TIMESTAMP);
    END IF;
END;
```

Notes about Example 5-8 on page 123:

1 The OF clause is used to specify that the trigger is invoked only when the value of the salary column changes.

2 The WHEN clause is still required because the trigger will be activated whenever the salary column is part of the update operation, even if the value is not changed.

Consider this additional information about transition variables and their values:

- The new values represent the value that is specified on the triggering INSERT or UPDATE operation, plus any modifications to the value by other triggers that were activated before the current trigger.
- DB2 for i supports a column attribute HIDDEN, which (among other things) specifies that the column is included in the column list of a SELECT only when it is specified explicitly. For example, it is not included in the column list when SELECT * is specified. These implicitly hidden columns are not treated any differently in the body of a trigger. They can be referenced as transition variables and as columns in transition tables.
- In IBM i 7.2, DB2 for i added support for row and column access control (RCAC). All of the columns and rows are visible to triggers, including those columns and rows whose access is otherwise controlled. The SECURED attribute for the trigger serves as an assertion that declares that the user established an audit procedure for all of the processing that is contained in the trigger to ensure that no data security policies are circumvented. For more information about RCAC, see “NOT SECURED and SECURED” on page 61.

5.5.3 Multiple event triggers

Multiple event triggers are defined so that they are activated for a combination of insert, update, or delete operations. Multiple event triggers make it easier for the programmer or DBE to define one trigger instead of multiple triggers. Multiple event triggers can use trigger event predicates so that the triggered action can reflect the trigger event that resulted in the activation of the trigger. These predicates can be specified anywhere in the triggered action. The predicates are listed:

- DELETING is true only if the trigger was activated by a delete operation.
- INSERTING is true only if the trigger was activated by an insert operation.
- UPDATING is true only if the trigger was activated by an update operation.

Example 5-9 shows the use of these predicates to consolidate events that can be separate triggers.

Example 5-9 Using a multiple event trigger to manage employee information

```
CREATE TRIGGER manage_employee
AFTER INSERT OR DELETE OR UPDATE OF salary ON employee                  1
REFERENCING NEW ROW AS n OLD ROW AS o
FOR EACH ROW MODE DB2ROW
BEGIN ATOMIC                                                            2
    IF INSERTING THEN                                                   3
        UPDATE company_stats SET empcount = empcount + 1;
    END IF;
    IF DELETING THEN                                                    4
        UPDATE company_stats SET empcount = empcount - 1;
    END IF;
    IF UPDATING AND n.salary > o.salary THEN                            5
        INSERT INTO salary_audit (empno, old_salary, new_salary, change_user, change_time)
            VALUES (o.empno, o.salary, n.salary, USER, CURRENT_TIMESTAMP);
```

```
    END IF;
END;
```

Notes about Example 5-9:

1. The OR clause is used to specify that the trigger will be activated for insert, delete, or update operations on the employee table. The column-level granularity is still allowed. Only those updates that change salary activate the trigger. An update to another column, such as workdept, does not activate the trigger.
2. ATOMIC is specified on the BEGIN to ensure that either both tables are modified or no tables are modified in a failure if the triggering application or user runs under commitment control.
3. The INSERTING predicate is specified on the IF statement. If the trigger is activated for an insert, it increments the count of employees in the company_stats table.
4. The DELETING predicate is specified on the IF statement. If the trigger is activated for a delete, it decrements the count of employees in the company_stats table.
5. The UPDATING predicate is specified on the IF statement in addition to the predicate checking so that you can see whether the new salary value is greater than the old salary value. As shown in a previous example, it inserts a row into the salary_audit table to record information about the change.

5.5.4 Changing row values in a BEFORE trigger

Because BEFORE triggers are activated before the triggering event occurs, they can be used on insert or update operations to change column values before they are inserted or updated in the database. This technique can be used for data cleansing or to help manage extensions to a database.

Example 5-10 shows a simple trigger to ensure that any new projects that are inserted into the project table have a project name value in uppercase letters.

Example 5-10 Using a BEFORE trigger to cleanse input data to ensure uppercase for project names

```
CREATE OR REPLACE TRIGGER validate_project_data
BEFORE INSERT ON project
REFERENCING NEW ROW AS new
FOR EACH ROW MODE DB2ROW                                    1
BEGIN
  SET new.projname=UPPER(new.projname);                     2
END;
```

Notes about Example 5-10:

1. Create the trigger validate_project_data so that it is activated before insert operations on the project table. It is a row-level trigger, and it specifies mode DB2ROW so that the trigger will be activated for each row that is inserted. The `new` is used to refer to the new column values.
2. The built-in function UPPER is used to ensure that the new value for projname is converted to uppercase before the insert occurs. The trigger simply modifies the new column value and returns control to DB2 for i, which performs the actual insert operation.

The same technique can be extended to UDFs. For example, we can add a mailing_address column to our employee table.

Example 5-11 shows a simple trigger to ensure that any values for the mailing_address column are normalized by a UDF. (Applications often need to normalize addresses. This task includes standardizing variations in spelling and syntax, such as `St` for street and `N` for north.)

Example 5-11 Using a BEFORE trigger to cleanse input data to normalize an address

```
CREATE OR REPLACE TRIGGER normalize_address
BEFORE INSERT OR UPDATE ON employee
REFERENCING NEW ROW AS new
FOR EACH ROW MODE DB2ROW                                          1
BEGIN
  SET new.mailing_address=mail.normalize(new.mailing_address);  2
END;
```

Notes about Example 5-11:

1 Create the trigger normalize_address so that it is activated before insert operations on the table project. It is a row-level trigger, and it specifies mode DB2ROW so that the trigger is activated for each row that is updated. The `new` is used to refer to the new column values.

2 The UDF that is called `normalize` in schema mail is used to ensure that the mailing_address column contains a valid value that is based on normalizing the address. As in the previous example, the trigger modifies the new column value and returns control to DB2 for i, which performs the actual insert or update operation.

You can use triggers to manage extensions to the database, such as adding or expanding columns to a file or table. A simple example of field expansion is based on this fictional business scenario:

- The fictional company ABC acquired another company that is named XYZ.
- The current department number in the database is expressed as a 3-character value.
- The ABC company wants to switch to the use of a new longer version of the department number (currently six characters) that consists of the company division name (“ABC” or “XYZ”) concatenated to the existing department number.
- This change will occur gradually. Initially, both divisions will continue to use the same applications and user interfaces to manage their own employees. New applications will be written over time to provide more consolidated access to the data.

To implement these requirements, the company takes the following actions:

- Alter the table dept to add a column that is called extdeptno for the new extended department number. The new column defaults to the null value.
- Use database modernization techniques to minimize changes to existing applications. SQL INSERT statements that specify a column list will not require any change. Record-level access programs can use a surrogate view or logical file to expose only the existing columns. This approach minimizes and often eliminates any need to recompile the programs.
- Create the trigger that is shown in Example 5-12 to populate the new extdeptno column.

Example 5-12 Using a BEFORE trigger to extend the data model

```
CREATE OR REPLACE TRIGGER ext_dept
BEFORE INSERT ON dept
REFERENCING NEW ROW AS new
FOR EACH ROW MODE DB2ROW                                          1
```

```
BEGIN
  SET new.extdeptno = gv_div CONCAT new.deptno;                2
END;
```

Notes about Example 5-12:

1 Create the trigger ext_dept so that it is activated before insert operations on the table dept. It is a row-level trigger, and it specifies mode DB2ROW so that it is activated for each row that is inserted. The `new` is used to refer to the new column values.

2 The new column extdeptno that reflects the extended department number is set to a value that is derived from concatenating a global variable (*gv_div*), which contains the division, with the existing deptno column. The global variable must be created in advance with an initial value that is based on the environment where the application programs are running.

This technique is flexible. You can use this technique in many other ways.

5.5.5 Calling a procedure from a trigger

Calling procedures from triggers is useful. This capability provides many advantages:

- It allows better modularity and reuse because each program can have a well-defined function and can be called from multiple places in the application environment. The same procedure can even be called from multiple triggers.
- Calling procedures from triggers can provide better concurrency. Modifying an existing trigger requires an exclusive lock on the subject table or view. If the algorithms are implemented as SQL procedures, they can be modified without requiring an exclusive lock on the table or view.
- Procedures can use SQL statements, such as CONNECT, that are not allowed in the trigger body.
- External procedures can be invoked to access functions that are not part of the database, including application programming interface (API) calls. Because they are called every time that the trigger is activated, these procedures need to be relatively short-running procedures.

The following example uses the CALL statement to call the appropriate procedures based on the type of change that is made. The CALL statement is covered in 4.6, “Calling a procedure” on page 89. SQL triggers restrict the use of correlation name-qualified columns in certain cases:

- BEFORE triggers can use new column values as parameters on CALL statements because these values are still eligible to be changed as shown in the previous section.
- BEFORE triggers cannot use old column values as parameters on CALL statements. DB2 for i does not currently consider whether the parameter was declared as input only in the procedure definition.
- AFTER triggers cannot use either old or new column values on CALL statements. This rule applies even if the parameter was declared as input only in the procedure definition.

The circumvention for this restriction is to retrieve column values into local variables, which can be specified as parameters on the CALL statement.

Example 5-13 shows a new variation of the trigger that was first shown in Example 5-9 on page 124. This version calls procedures to manage the three events on the employee table.

Example 5-13 Using a multiple event trigger to manage employee information with procedure calls

```
CREATE or replace TRIGGER manage_employee
AFTER INSERT OR DELETE OR UPDATE OF salary ON employee
REFERENCING NEW ROW AS n OLD ROW AS o
FOR EACH ROW MODE DB2ROW
PROGRAM NAME mngemp                                              1
BEGIN
DECLARE v_empno CHAR(6);
DECLARE v_osalary,v_nsalary DECIMAL(9,2);                        2
    IF INSERTING THEN
        SET v_empno = n.empno;
        CALL new_employee (v_empno);                             3
    END IF;
    IF DELETING THEN
        SET v_empno = o.empno;
        CALL exit_employee (v_empno);                            4
    END IF;
    IF UPDATING AND n.salary > o.salary THEN
          SET v_empno = o.empno;
          SET v_osalary = o.salary;
          SET v_nsalary = n.salary;
          CALL salary_audit (v_empno, v_osalary, v_nsalary);     5
    END IF;
END;
```

Notes about Example 5-13:

1 Create the trigger manage_employee so that it is activated after insert, delete, or updates on the employee table. For updates, the trigger is activated only if the column salary is included on the update. It is a row-level trigger, and it specifies mode DB2ROW so that it will be activated for each row that is inserted, deleted, or updated. The `new` is used to refer to the new column values. The program name clause is used to specify that the generated C program needs to be named mngemp.

2 Declare local variables for the employee number, old salary, and new salary values.

3 If the trigger was activated for an insert, use the SET statement to copy the new empno value to the local variable. Call the procedure that is named new_employee that passes the local variable.

4 If the trigger was activated for a delete, use the SET statement to copy the new empno value to the local variable. Call the procedure that is named exit_employee that passes the local variable.

5 If the trigger was activated for an update of the salary, use the SET statement to copy the new empno, plus the old and new salary variables, to the local variables. Call the procedure that is named salary_audit that passes the local variables.

The procedures that are called can be written in any language that is supported by the SQL CREATE PROCEDURE syntax and take the necessary action. For example, salary_audit can be written in RPG and call an API to send an email to the payroll manager.

5.5.6 Using transition tables

An SQL trigger might need to refer to all of the affected rows for an insert, an update, or a delete operation. This situation is true, for example, if the trigger needs to apply aggregate functions, such as MIN or MAX, to a specific column of the affected rows. The old and new transition tables can be used for this purpose.

OLD TABLE and NEW TABLE refer to virtual (rather than physical) tables that contain the old and new row values for any rows that changed. For INSERT, the old table is empty. For DELETE, the new table is empty.

Transition tables are only generally available on triggers that are defined with a granularity of FOR EACH STATEMENT.

Example 5-14 shows a trigger that monitors the total bonus after an update operation that changes one or more rows.

Example 5-14 Using transition tables for an aggregated value

```
CREATE or replace TRIGGER total_bonus
AFTER UPDATE OF bonus ON employee
REFERENCING NEW TABLE as newtbl
            OLD TABLE as oldtbl
FOR EACH STATEMENT MODE DB2SQL                                          1
BEGIN
DECLARE old_totalbonus DECIMAL(13,2);
DECLARE new_totalbonus DECIMAL(13,2);                                   2
SELECT SUM(bonus) INTO old_totalbonus FROM oldtbl;
SELECT SUM(bonus) INTO new_totalbonus FROM newtbl;                      3
IF new_totalbonus > old_totalbonus THEN
    INSERT INTO bonus_log (ototbonus, newtotbonus,
      bonusdesc, bonusdate)
   VALUES(old_totalbonus, new_totalbonus,
      'New bonus total exceeds old bonus total.', CURRENT_DATE);        4
END IF;
END;
```

Notes about Example 5-14:

1 Create the trigger total_bonus so that it is activated after update operations on the employee table. The virtual table that contains new row data will be called `newtbl`. The virtual table that contains old row data will be called `oldtbl`. It is a statement-level trigger. It specifies mode DB2SQL so it is activated one time after the row updates.

2 Declare local variables for the old and new total bonus.

3 Use SELECT statements with the aggregate function SUM on the old and new tables to calculate the sum of all bonuses and store them in the local variables. The `oldtbl` and `newtbl` are referenced as though they were physical tables.

4 If the new total bonus is greater than the old total bonus, insert a row into the bonus_log that records the event and the current date.

The use of transition tables is not limited to aggregated values. As virtual tables, they can be used in other ways within a query. Example 5-15 shows a trigger that logs changes in bonuses, including the employee number and the name of the employee's immediate manager. To derive this information, it implements a three-way join between the old and new transition tables and the data that is derived from the employee table. The trigger uses both subqueries and common table expressions (CTEs). Despite its length, it is a single SQL statement.

Example 5-15 Using transition tables within joins

```
CREATE or replace TRIGGER employee_bonus
AFTER UPDATE OF bonus ON employee
REFERENCING NEW TABLE as newtbl
            OLD TABLE as oldtbl
FOR EACH STATEMENT MODE DB2SQL                                    1
BEGIN
INSERT INTO employee_bonus_log                                    2
   WITH olddta (oldemp, oldbonus, olddept) AS
      (SELECT empno, bonus, workdept FROM oldtbl ),               3
     newdta (newemp, newbonus ) AS
      (SELECT empno, bonus FROM newtbl ),                         4
     mgrdta (mgrdept, mgrempno, mgrname) AS
      (SELECT workdept, empno, lastname FROM employee
         WHERE empno IN
            (SELECT empno FROM employee WHERE JOB='MANAGER'
         AND workdept in (SELECT olddept FROM olddta)    )     )  5
   SELECT oldemp, oldbonus, newbonus, olddept, mgrempno, mgrname
      FROM ( olddta INNER JOIN newdta ON oldemp=newemp )
         LEFT OUTER JOIN mgrdta ON olddept=mgrdept ;              6
END;
```

Notes about Example 5-15:

1 Create the trigger employee_bonus so that it is activated after update operations on the employee table. The virtual table that contains new row data will be called newtbl. The virtual table that contains old row data will be called oldtbl. It is a statement-level trigger and specifies mode DB2SQL so that it will be activated one time after the row updates.

2 The employee_bonus_log table is the target of the insert operation.

3 The first CTE, which is called `olddta`, represents the employee number, the bonus value, and the department number from the old transition table. The CTE uses assigned column names to make the statement more readable.

4 The second CTE, which is called `newdta`, represents the employee number and the bonus value from the new transition table. The CTE uses assigned column names to make the statement more readable.

5 The third CTE, which is called `mgrdta`, represents the department number, employee number, and the last names of all of the managers that relate to this query. This logical set is derived from the employee table with subqueries that are used to subset the list to the employees with the job of manager and the employees that manage employees that are involved in this update operation based on the list of departments that was derived from the old transition table.

6 Now, the sets are well-defined and can be logically processed. The final part of the query uses an inner join of the old and new data based on employee number. This result is then left outer-joined to the logical set of managers. Left outer join is used in case no manager was found for the employee. This combined join gathers the information that is needed for the employee bonus log: employee number, old bonus, new bonus, department, the manager's employee number, and the manager's name.

Transition tables can also be referenced directly by using normal SQL programming techniques, such as OPEN, FETCH, and CLOSE. This technique is not covered by an example because you really need to ask whether it is more appropriate to use a FOR EACH ROW trigger in this case. Even the previous example might cause you to ask the same thing.

5.5.7 Signaling an error

Because triggers are programs that are written in SQL, they can use all of the exception handling and signaling capabilities that are supported by the SQL programming language:

- Triggers can monitor for error conditions and programmatically decide what to do. The only major restriction is that they cannot use an UNDO handler. Information about error handling is described in the Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5.
- Triggers can explicitly signal errors to the application or user that activated the trigger. The execution of a SIGNAL statement in the trigger body results in the return of an *SQLCODE* -438 and the *SQLSTATE* that is specified on the SIGNAL statement:
  - A BEFORE trigger can signal an error that forces the operation to end in error without making any of the corresponding changes.
  - An AFTER trigger can also signal an error but because the triggering change already happened, this AFTER trigger alone does not result in the failure of the triggering event. Instead, it is only reported to the individual who made the change and the individual decides what to do, which might include a ROLLBACK to remove the changes.
- Any other errors that occur during the execution of the SQL trigger body result in the return of an *SQLCODE* -723 and an *SQLSTATE* 09000.

  In this case, the message contains information about the SQL exception that caused the trigger to fail.

Example 5-16 shows the use of transition variables in a trigger to enforce a business rule that a bonus cannot exceed the salary.

Example 5-16 Using a trigger to limit bonuses

```
CREATE OR REPLACE TRIGGER limit_bonus
BEFORE UPDATE ON employee
REFERENCING NEW ROW AS n OLD ROW AS o
FOR EACH ROW MODE DB2ROW                                          1
BEGIN
  IF n.bonus > o.salary THEN                                      2
      SIGNAL SQLSTATE '87001'                                     3
      SET MESSAGE_TEXT = 'Bonus exceeds employee salary.';        4
  END IF;
END;
```

Notes about Example 5-16 on page 131:

1. The trigger limit_bonus is defined so that it is activated before UPDATE operations on the employee table. It is a row-level trigger, and it specifies mode DB2ROW so that it is activated as each row is updated. The definition uses the REFERENCING clause so that it can refer to the old and new column values for the row that is updated.
2. The IF statement is used to check whether the new value for the bonus column is greater than the old salary value.
3. If the new value for the bonus column is greater than the old salary value, the trigger signals an *SQLSTATE* of 87001 to the SQL statement that performs the update. Because this trigger is a BEFORE trigger, the update operation fails and this row is not updated.
4. The trigger also includes an explanatory message that is reported back to the program or user that attempted to violate the business rule. This information is available to the activating program in the SQL communication area (SQLCA) or by using the GET DIAGNOSTICS statement.

Attempting an insert that violates this business rule results in the error that is shown in Figure 5-10.

```
SQL State: 87001
SQL Code: -438
SQL trigger LIMIT_BONUS in JIMD failed with SQLCODE -438 SQLSTATE 87001.
Cause:   An error has occurred in a triggered SQL statement in trigger LIMIT_BONUS in schema
JIMD. The SQLCODE is -438, the SQLSTATE is 87001, and the message is Bonus exceeds
company maximum.
```

Figure 5-10 Error that is signaled by a trigger as the result of a SIGNAL statement

Now, we force an SQL statement to fail in one of our previous trigger examples. Example 5-17 shows the audit_salary trigger that inserts rows into the salary_audit table when a salary is increased. This time, we use the LOCK TABLE SQL statement in another session to ensure that the trigger will not be able to update the table.

Example 5-17 Using a trigger to audit salary increases

```
CREATE OR REPLACE TRIGGER audit_salary
AFTER UPDATE ON employee
REFERENCING NEW ROW AS n OLD ROW AS o
FOR EACH ROW MODE DB2ROW
WHEN (n.salary<>o.salary)
BEGIN
    IF n.salary > o.salary THEN
        INSERT INTO salary_audit
             (empno, old_salary, new_salary, change_user, change_time)
          VALUES (o.empno, o.salary, n.salary, USER, CURRENT_TIMESTAMP);
    END IF;
END;
```

An attempt to update the employee table to increase an employee's salary will now report a failure because the trigger cannot access the salary_audit table. The error is shown in Figure 5-11.

```
SQL State: 09000
SQL Code: -723
Message: SQL trigger AUDIT_SALARY in JIMD failed with SQLCODE -913 SQLSTATE 57033.
Cause:   An error has occurred in a triggered SQL statement in trigger AUDIT_SALARY in schema
JIMD.  The SQLCODE is -913, the SQLSTATE is 57033, and the message is Row or object
SALAR00001 in JIMD type *FILE in use.
```

Figure 5-11 Error that is signaled by a trigger as the result of an SQL error

The RETURN is not allowed in a trigger so the logic must reflect exiting the trigger after an error is signaled. Although GOTO is not considered a preferred programming practice, GOTO is allowed.

5.5.8 Self-referencing triggers

A *self-referencing trigger* directly references the subject table in the body of the trigger:

- Read-only access of the table based on SELECT statements. This usage includes operations against specific rows or against a logical combination of rows by using aggregate functions, such as AVG or SUM. These examples are less complicated because they do not result in any cascading/iterative activations of the trigger. An example of this type of self-referencing trigger is included in Example 5-19 on page 136.
- Changes to the table that are based on INSERT, UPDATE, and DELETE statements. This usage requires more careful programming because these situations can result in activating the trigger multiple times in a recursive fashion.

Assume that we must follow this business rule, "For every bonus an employee gets, the employee's immediate manager also gets a bonus that equals 10% of the employee's bonus." This business rule applies only to the first line of management and it does not result in bonuses at higher levels. To implement this rule, the trigger must find the employee's manager and increment the manager's bonus by 10% of the employee's bonus.

Example 5-18 shows a trigger to implement this business rule.

Example 5-18 Self-referencing trigger to apply manager commissions

```
CREATE OR REPLACE TRIGGER manager_comm
AFTER UPDATE OF comm ON employee
REFERENCING NEW ROW AS new OLD ROW AS old
FOR EACH ROW MODE DB2ROW                                    1
WHEN (new.comm<>old.comm AND new.job<>'MANAGER')            2
BEGIN
    DECLARE v_mgrcomm DECIMAL(9,2);
    DECLARE v_mgrempno CHAR(6);                             3
    IF new.comm > old.comm THEN                             4
        SET v_mgrcomm = (new.comm-old.comm)/10;             5
        SET v_mgrempno = (SELECT mgrno FROM dept
           WHERE deptno=
               (SELECT workdept FROM employee
                   WHERE empno=new.empno) );                6
        IF v_mgrempno IS NOT NULL THEN                      7
          UPDATE employee SET comm=comm+v_mgrcomm
             WHERE empno=v_mgrempno;                        8
        END IF;
    END IF;
END;
```

Notes about Example 5-18:

1 Create the trigger manager_comm so that it is activated after update operations of the comm column on the employee table. The correlation value for new rows is `new`, and the correlation value for old rows is `old`. It is a row-level trigger, and it specifies mode DB2ROW so it will be activated one time for each row update.

2 The WHEN clause determines whether the comm value changed and whether the employee is a manager. If the value did not change or the employee is a manager, the trigger ends its processing.

3 Declare local variables for the manager's commission and for the manager's employee number.

4 Check whether the new comm value is greater than the old comm value. If it is not, the trigger ends its processing.

5 Calculate the manager's commission as the difference between the employee's new commission and the employee's old commission divided by 10.

6 Determine the manager's employee number by retrieving it from the dept table based on the employee's department number.

7 Determine whether the manager's employee number was successfully retrieved. If not, the trigger ends its processing.

8 Use the update statement to increment the manager's comm based on the previous calculation. This insert activates but it does not execute the trigger body because the WHEN clause ensures that the update is performed for non-managers only.

Self-referencing triggers are restricted because they are not allowed to update or delete the triggering row.

5.5.9 DB2ROW versus DB2SQL triggers

DB2 for i supports two trigger modes. MODE DB2SQL is the default, and it is more consistent with other DB2 implementations. MODE DB2ROW is nonstandard, but it is more consistent with traditional external triggers. The difference (if any) between the two modes depends on the definition of the trigger:

- Triggers that are defined with a granularity of FOR EACH ROW and an activation time of BEFORE do not have any point of comparison because they are automatically converted to MODE DB2ROW. This situation is reported as an SQL7051.
- Triggers that are defined FOR EACH STATEMENT do not have any point of comparison because they only can be MODE DB2SQL and AFTER.
- The only situation where an observable difference exists is if the triggers are defined FOR EACH ROW with an activation time of AFTER:
  - MODE DB2SQL triggers are activated after all of the row operations occur so that all of the rows will contain their new values.
  - MODE DB2ROW triggers are activated for each row as the rows are changed so the rows have a combination of their old and new values.

Figure 5-12 and Figure 5-13 show the difference in trigger activation for MODE DB2SQL and MODE DB2ROW for triggers that specify FOR EACH ROW.

```
Modify row 1
Modify row 2
...
Modify last row
Activate trigger for row 1
Activate trigger for row 2
...
Activate trigger for last row
```

Figure 5-12 MODE DB2SQL trigger activation sequence

```
Modify row 1
Activate trigger for row 1
Modify row 2
Activate trigger for row 2
...
Modify last row
Activate trigger for last row
```

Figure 5-13 MODE DB2ROW trigger activation sequence

This difference in behavior is significant when the logic in the trigger body depends on values in other rows that are potentially changed as part of the same triggering operation. The other rows can be referenced in the trigger either directly or indirectly by using an aggregate function, such as AVG or SUM.

Assume that we use a bonus log that records the bonuses any time that an update occurs to the employee table. The trigger enforces a business rule that the bonus for one employee cannot be greater than two times the average bonus for the employees in the department. This example is intentionally and artificially restricted to department C01, which has four employees, to simplify this demonstration of the difference in behavior.

Example 5-19 shows a DB2SQL trigger that implements this set of business rules.

Example 5-19 MODE DB2SQL trigger

```
CREATE OR REPLACE TRIGGER employee_bonus_db2sql
AFTER UPDATE ON employee
REFERENCING OLD AS o NEW as n
FOR EACH ROW MODE DB2SQL                                                    1
  DECLARE v_avgbonus DECIMAL(9,2);
  SET v_avgbonus
      = (SELECT AVG(bonus) FROM employee WHERE workdept='C01');             2
  INSERT INTO employee_bonus_log (pass, empno, bonus, avgbonus)
    SELECT gv_trigger_pass, empno, bonus, v_avgbonus FROM employee
       WHERE workdept='C01';                                                3
  SET gv_trigger_pass = gv_trigger_pass+1;                                  4
END;
```

Notes about Example 5-19:

1 The trigger employee_bonus_db2sql is defined so that it is activated after UPDATE operations on the employee table. It is a row-level trigger, and it specifies mode DB2SQL so it will be activated *n* times after all of the rows are updated. The definition uses the REFERENCING clause so that it can refer to the old and new column values for the row that is updated.

2 The query is self-referencing because it calculates the average bonus for the employees based on the current rows in the employee table.

3 The bonus log is populated with the current pass through the trigger, the employee number, the new bonus value, and the average bonus that was calculated previously. The value in the column pass reflects how many times the trigger was activated because the trigger increments the global variable *gv_trigger_pass* by one on each activation.

4 The global variable that tracks the number of passes through the trigger is incremented by 1.

Figure 5-14 shows the values for the employees in department C01 before any update operations.

WORKDEPT	EMPNO	BONUS
C01	000030	800.00
C01	000130	500.00
C01	000140	600.00
C01	200140	600.00

Figure 5-14 Department C01 employees before update

Example 5-20 shows the update operation that was used to set the bonus for the employees in department C01 to 1200.

Example 5-20 UPDATE statement

```
UPDATE employee SET bonus=1200 WHERE workdept='C01';
```

Figure 5-15 shows the contents of the employee_bonus_log after the execution of the update statement. Consider the following information:

- The trigger was activated a total of four times.
- Each activation of the trigger created log entries for the four employees in department C01.
- The bonus and the average_bonus in every row is always equal to 1200 because a MODE DB2SQL FOR EACH ROW trigger is activated after all of the rows are updated. Even the first time that the trigger is activated, all of the employee bonuses were already updated to 1200.

PASS	EMPNO	BONUS	AVGBONUS
1	000030	1200.00	1200.00
1	000130	1200.00	1200.00
1	000140	1200.00	1200.00
1	200140	1200.00	1200.00
2	000030	1200.00	1200.00
2	000130	1200.00	1200.00
2	000140	1200.00	1200.00
2	200140	1200.00	1200.00
3	000030	1200.00	1200.00
3	000130	1200.00	1200.00
3	000140	1200.00	1200.00
3	200140	1200.00	1200.00
4	000030	1200.00	1200.00
4	000130	1200.00	1200.00
4	000140	1200.00	1200.00
4	200140	1200.00	1200.00

Figure 5-15 The employee_bonus_log after updates with a DB2SQL trigger

Example 5-21 shows an almost identical trigger definition that varies only because it specifies MODE DB2ROW rather than MODE DB2SQL.

Example 5-21 MODE DB2ROW trigger

```
CREATE OR REPLACE TRIGGER bonus_log_db2row
AFTER UPDATE ON employee
REFERENCING OLD AS o NEW as n
FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE v_avgbonus DECIMAL(9,2);
  SET v_avgbonus = (SELECT AVG(bonus) FROM employee WHERE workdept='C01');
  INSERT INTO employee_bonus_log (pass, empno, bonus, avgbonus)
    SELECT gv_trigger_pass, empno, bonus, v_avgbonus FROM employee
      WHERE workdept='C01';
  SET gv_trigger_pass = gv_trigger_pass+1;
END;
```

Figure 5-16 shows the contents of the employee_bonus_log after the execution of the same update statement that was shown previously. (The employee bonus values were reset to their original values between these two examples.) Consider the following information:

- The trigger was activated a total of four times.
- Each activation of the trigger created log entries for the four employees in department C01.
- The bonus and the average_bonus are no longer always equal to 1200. On the first pass, one row was updated with the new value of 1200 and the average is now 725. On the second pass, two rows were updated to the new value of 1200, which increases the average to 900. Each time that the trigger is activated, the data shows that one more row is updated, which changes the average value because a MODE DB2ROW FOR EACH ROW trigger is activated after each of the rows is updated.

PASS	EMPNO	BONUS	AVGBONUS
1	000030	1200.00	725.00
1	000130	500.00	725.00
1	000140	600.00	725.00
1	200140	600.00	725.00
2	000030	1200.00	900.00
2	000130	1200.00	900.00
2	000140	600.00	900.00
2	200140	600.00	900.00
3	000030	1200.00	1050.00
3	000130	1200.00	1050.00
3	000140	1200.00	1050.00
3	200140	600.00	1050.00
4	000030	1200.00	1200.00
4	000130	1200.00	1200.00
4	000140	1200.00	1200.00
4	200140	1200.00	1200.00

Figure 5-16 The employee_bonus_log after updates with a DB2ROW trigger

One mode is not clearly better than the other mode. It depends on how the business defines its needs. In certain cases, one mode is a better fit than the other mode. In this case, the trigger can enforce a business rule that no employee will receive a bonus that is more than twice the average bonus for the department. The trigger mode greatly affects the implementation of any business rule that relies on an aggregated value.

Important: If a business rule can be implemented in either mode, DB2ROW performs better because each row can be processed a single time.

5.5.10 INSTEAD OF triggers

INSTEAD OF triggers are allowed on an SQL view. INSTEAD OF triggers offer more flexibility for insert, update, and delete operations that are directed against an SQL view. The triggered action effectively takes the place of those operations. The following restrictions apply to INSTEAD OF triggers:

- INSTEAD OF triggers are allowed on SQL views only. They are not allowed on DDS logical files.
- Only one INSTEAD OF trigger is allowed for each of the triggering events, which are INSERT, UPDATE, or DELETE.
- The triggered action cannot use the WHEN clause.
- FOR EACH STATEMENT cannot be specified.

The following scenario illustrates how an INSTEAD OF trigger makes a view that can be updated rather than read only. Currently, the employee table contains columns, such as empno, firstnme, lastname, workdept, and phoneno. It also contains columns for birthdate, salary, bonus, and comm. The company determined that the last four columns are sensitive personal information and decided to move them to a different table for more granular control of the employees that can see this data.

Many ways exist to implement this decision. This approach is only one alternative. We move the columns to a new table that is called employee_spi, which uses the same empno value for its primary key and contains the following columns, that used to be part of the employee table:

- `birthdate`
- `salary`
- `bonus`
- `comm`

A new administrative view, which is named employee_admin, joins the employee table to the employee_spi table based on the empno column. Its definition is shown in Example 5-22.

Example 5-22 CREATE VIEW statement for the employee_admin view

```
CREATE VIEW employee_admin AS
(SELECT e.empno, e.firstnme, e.midinit, e.lastname,
e.workdept, e.phoneno, e.hiredate, e.job, e.edlevel, e.sex,
spi.birthdate, spi.salary, spi.bonus, spi.comm                  1
FROM employee e INNER JOIN employee_spi spi                     2
ON e.empno=spi.empno);                                          3
```

Notes about Example 5-22:

1. The view selects the columns from the two tables in the same sequence as they appear in the original employee table.
2. An inner join is used to associate the two tables: employee and employee_spi.
3. The join is performed by using the employee number from the employee and employee_spi tables. This relationship is an associative, one-to-one relationship.

This employee_admin view will be used by administrative personnel with the authority to manage sensitive personal information. This view serves to isolate these users from any future changes to the data model. Because it contains a join, this view allows read-only access to the underlying data, and it cannot be used directly for inserts, updates, or deletes.

INSTEAD OF triggers can be used to effectively “deconstruct” the triggering event to make the appropriate changes to the employee and employee_spi tables:

- During an INSERT operation, the trigger must insert the appropriate rows in both tables and ensure that they have the same value for the empno column.
- During an UPDATE operation, the trigger must update the appropriate columns in both tables.
- During a DELETE operation, the trigger must delete the appropriate rows in both tables.

By using the triggering event predicates for the multiple event triggers that were described, all of these actions can be configured as part of a single trigger, as shown in Example 5-23.

Example 5-23 Creating an INSTEAD OF trigger to change two underlying tables

```
CREATE TRIGGER employee_admin
INSTEAD OF
INSERT OR UPDATE OR DELETE ON employee_admin
REFERENCING NEW ROW AS n OLD ROW AS o
FOR EACH ROW MODE DB2ROW
PROGRAM NAME empadmin                                                    1
BEGIN ATOMIC                                                             2
  IF INSERTING THEN
      INSERT INTO employee (empno,firstnme,midinit,lastname,
          workdept,phoneno,hiredate,job,edlevel,sex)
        VALUES (n.empno,n.firstnme,n.midinit,n.lastname,
            n.workdept,n.phoneno,n.hiredate,n.job,n.edlevel,n.sex);
      INSERT INTO employee_spi (empno,birthdate,salary,bonus,comm)
        VALUES (n.empno,n.birthdate,n.salary,n.bonus,n.comm);              3
  END IF;
  IF UPDATING THEN
      IF n.empno<>old.empno THEN
        SIGNAL SQLSTATE '87007'
        SET MESSAGE_TEXT
          = 'Employee number in EMPLOYEE cannot be updated.';              4
      END IF;
      UPDATE employee SET
        firstnme=n.firstnme,midinit=n.midinit,lastname=n.lastname,
        workdept=n.workdept,phoneno=n.phoneno,hiredate=n.hiredate,
        job=n.job,edlevel=n.edlevel,sex=n.sex
         WHERE empno=old.empno;
      UPDATE employee_spi SET
         birthdate=n.birthdate,salary=n.salary,bonus=n.bonus,comm=n.comm
          WHERE empno=old.empno;                                           5
  END IF;
  IF DELETING THEN
      DELETE FROM employee     WHERE empno=old.empno;
      DELETE FROM employee_spi WHERE empno=old.empno;                      6
  END IF;
END;
```

Notes about Example 5-23 on page 140:

1 Create the trigger employee_admin so that it is activated instead of any insert, update, or delete operations on the table employee admin. The correlation value for new rows is n and the correlation value for old rows is `o`. It is a row-level trigger, and it specifies mode DB2ROW so that it will be activated one time for each row modification.

2 ATOMIC is specified on the BEGIN to ensure that either both tables are modified or no tables are modified in a failure if the triggering application or user is running under commitment control.

3 If the trigger was activated because of an insert, perform inserts into the employee and employee_spi tables. The statements deconstruct the columns that now go to each table. The `empno` is included in both tables because it is the associative key.

4 If the trigger was activated because of an update, check for any attempt to update the employee number column empno. If the user attempted it, signal an *SQLSTATE* of '`87007`' and exit.

5 If the trigger was activated because of an update, perform updates to the employee and employee_spi tables. The statements again deconstruct the columns that belong to each table. They do not update the employee number.

6 If the trigger was activated because of a delete, delete the rows with the matching employee number from the employee and employee_spi tables.

Special considerations exist for INSTEAD OF triggers for new transition variables that are all nullable.

On an INSTEAD OF INSERT trigger, they can contain the following values:

- If a value was specified for the column on the INSERT statement, the transition variable or the transition table column will contain that value.
- If a value is not explicitly specified for the column on the INSERT statement or the **`DEFAULT`** keyword was specified, the corresponding new transition variable or the new transition table column will be set:
  - It contains the default value of the underlying table column if the view column can be updated (even without the INSTEAD OF trigger) and the column is not based on an auto-generated column (identity, row change timestamp, or ROWID).
  - Otherwise, the value is `null`.

On an INSTEAD OF UPDATE trigger, the new transition variables can contain the following values:

- If a value is explicitly specified for a column on the UPDATE statement, the new transition variable or new transition table column contains that value.
- If the **`DEFAULT`** keyword is explicitly specified for a column on the UPDATE statement, the corresponding new transition variable or new transition table column is set:
  - It contains the default value of the underlying table column if the view column can be updated (without the INSTEAD OF trigger) and the column is not based on an auto-generated column (identity column or ROWID).
  - Otherwise, the value is `null`.
- Otherwise, the corresponding new transition variable or new transition table column is the existing value of the column in the row.

5.6 Additional trigger considerations

This section describes additional trigger considerations.

5.6.1 Trigger limits

DB2 for i has two relevant limits that relate to triggers:

- The maximum number of triggers on a table is limited to 300.
- The maximum runtime depth of cascading triggers is limited to 200 or until the job/session exceeds its maximum storage, whichever comes first.

It is unlikely that your environment will challenge these limits.

5.6.2 Qualifying references

When triggers are created, the triggered action is modified to qualify all unqualified references to enhance their integrity and to prevent any unintended consequences from changes in the environment or in the referenced objects.

When the trigger is created, all tables, views, aliases, distinct types, global variables, UDFs, sequences, and procedures that are referenced in the triggered-action must exist on the current server. The table or view that an alias refers to must also exist when the trigger is created. This requirement also includes objects in library `QTEMP`. Although objects in `QTEMP` can be referenced in the triggered-action, dropping those objects in `QTEMP` will not cause the trigger to be dropped. This rule also applies to global temporary tables. The trigger will function correctly if the new instances of these objects with the same column names were created before the trigger was activated.

When the time the trigger is created, the modified trigger action reflects the following changes:

- The naming mode is always set to SQL naming.
- All unqualified references are explicitly qualified.
- All implicit column lists are modified to contain the actual list of column names, for example:
  - SELECT statements that specify * as the column list
  - INSERT statements that do not specify a column list
  - UPDATE statements that use the SET ROW clause

By default, the modified triggered action is stored in the catalog. The modifications are also apparent based on looking at the generated C program or regenerating the SQL source.

Example 5-24 shows the CREATE TRIGGER statement, which decrements the employee count and logs the information for the deleted employee in a table that is named former_employee, which has the same column definitions as the employee table.

Example 5-24 Original CREATE TRIGGER statement with unqualified references

```
CREATE TRIGGER exit_employee
AFTER DELETE ON employee
REFERENCING OLD ROW AS oldrow
FOR EACH ROW MODE DB2ROW
BEGIN
  UPDATE company_stats SET empcount = empcount - 1;
  INSERT INTO former_employee SELECT * FROM employee WHERE empno=oldrow.empno;
END;
```

Example 5-25 shows the CREATE TRIGGER statement after all of the references are fully qualified. It is formatted for readability.

Example 5-25 System-generated CREATE TRIGGER statement with fully qualified references

```
CREATE TRIGGER JIMD.EXIT_EMPLOYEE
   AFTER DELETE ON JIMD.EMPLOYEE                                    1
   REFERENCING OLD AS OLDROW
   FOR EACH ROW
   MODE DB2ROW
   SET OPTION  ALWBLK = *ALLREAD ,
      ALWCPYDTA = *OPTIMIZE ,
      COMMIT = *CHG ,
      DECRESULT = (31, 31, 00) ,
      DFTRDBCOL = JIMD ,
      DYNDFTCOL = *NO ,
      DYNUSRPRF = *USER ,
      SRTSEQ = *HEX                                                 2
BEGIN
  UPDATE JIMD . COMPANY_STATS SET EMPCOUNT =
   JIMD . COMPANY_STATS . EMPCOUNT - 1 ;                            3
  INSERT INTO JIMD . FORMER_EMPLOYEE
   ( EMPNO , FIRSTNME , MIDINIT , LASTNAME , WORKDEPT ,
      PHONENO , HIREDATE , JOB ,
      EDLEVEL , SEX , BIRTHDATE , SALARY , BONUS , COMM )           4
   SELECT JIMD . EMPLOYEE . EMPNO , JIMD . EMPLOYEE . FIRSTNME ,
      JIMD . EMPLOYEE . MIDINIT , JIMD . EMPLOYEE . LASTNAME ,
      JIMD . EMPLOYEE . WORKDEPT , JIMD . EMPLOYEE . PHONENO ,
      JIMD . EMPLOYEE . HIREDATE , JIMD . EMPLOYEE . JOB ,
      JIMD . EMPLOYEE . EDLEVEL , JIMD . EMPLOYEE . SEX ,
      JIMD . EMPLOYEE . BIRTHDATE , JIMD . EMPLOYEE . SALARY ,
      JIMD . EMPLOYEE . BONUS , JIMD . EMPLOYEE . COMM
         FROM JIMD . EMPLOYEE
            WHERE JIMD . EMPLOYEE . EMPNO = OLDROW . EMPNO ;        5
END  ;
```

Notes about Example 5-25 on page 143:

1 The reference to the employee table on the ON clause is fully qualified.

2 The SET OPTION statement reflects environmental attributes that were “bound” at the time that the trigger was created.

3 The UPDATE statement references to the table company_stats and its column empcount are fully qualified.

4 Now, the INSERT statement explicitly lists the target columns.

5 The SELECT part of the INSERT statement explicitly qualifies all of the column names and the employee table.

The default schema for the static SQL statements in the trigger can be changed by using the DFTRDBCOL option on SET OPTION. To change the default schema for dynamic SQL, it is also necessary to specify DYNDFTCOL(*YES) on SET OPTION.

5.6.3 Trigger program attributes

Additional attributes apply to the program that is generated for an SQL trigger to enhance their integrity and to prevent any unintended consequences from changes in the environment or in the referenced objects:

- SQL triggers always run in the same activation group as the triggering event. The generated program for an SQL trigger always shows ACTGRP(*CALLER).
- The values of the special registers are saved before a trigger is activated, and they are restored on return from the trigger. The values of the special registers are inherited from the triggering SQL operation.
- The program is created with STGMDL(*SNGLVL). If the trigger runs on behalf of an application that uses STGMDL(*TERASPACE) and also uses commitment control, the entire application needs to run under a job-scoped commitment definition (STRCMTCTL CMTSCOPE(*JOB)).

Note: SQL triggers always run in the caller’s activation group.

5.6.4 Adding columns to tables

If a column is added to the subject table after triggers are defined, the following rules apply:

- If the trigger is an UPDATE trigger that was defined without an explicit column list, an update to the new column causes the activation of the trigger.
- If the SQL statements in the triggered-action refer to the triggering table, the new column is not accessible to the SQL statements until the trigger is re-created.
- The OLD_TABLE and NEW_TABLE transition tables contain the new column, but the column cannot be referenced unless the trigger is re-created.

If a column is added to any table that is referenced by the SQL statements in the triggered-action, the new column is not accessible to the SQL statements until the trigger is re-created.

5.6.5 Dropping or revoking privileges on tables

If an object, such as a table, view, or alias, that is referenced in the triggered-action is dropped, the access plans of the statements that reference the object are rebuilt when the trigger is activated. If the object does not exist at that time, the corresponding INSERT, UPDATE, or DELETE operation on the subject table fails.

If a privilege that the creator of the trigger is required to have for the trigger to execute is revoked, the access plans of the statements that reference the object are rebuilt when the trigger is activated. If the appropriate privilege does not exist at that time, the corresponding INSERT, UPDATE, or DELETE operation on the subject table fails.

5.6.6 Renaming or moving a table

Any table (including the subject table) that is referenced in a triggered-action can be moved or renamed. However, the triggered-action will continue to reference the old name or schema. An error occurs if the referenced table is not found when the triggered-action is executed. Therefore, you must drop the trigger and then re-create the trigger so that it refers to the renamed or moved table.

5.6.7 Transaction isolation

All triggers, when they are activated, perform a SET TRANSACTION statement unless the isolation level of the application program that invokes the trigger is the same as the default isolation level of the trigger program. This SET TRANSACTION statement is necessary 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 can put the user's own SET TRANSACTION statements in an SQL-control-statement in the SQL-trigger-body of the trigger. If the user places a SET TRANSACTION statement within the SQL-trigger-body of the trigger, the trigger runs 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.

This approach is not recommended. Consider this approach carefully. It is also a preferred practice to create the trigger under the isolation level that will most often be used by the application programs that activate the trigger. SET OPTION can be used to explicitly choose the isolation level.

Note: By default, SQL triggers run at the same isolation level as the triggering event.

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 be run under commitment control and will 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 they cannot be rolled back.

If both system triggers that are defined by the Add Physical File Trigger (**ADDPFTRG**) CL command and SQL triggers that are defined by the CREATE TRIGGER statement are defined for a table, we recommend that the system triggers perform a SET TRANSACTION statement so that they are run with the same isolation level as the original application that caused the triggers to be activated.

Also, we recommend that the system triggers run in the activation group of the calling application. If system triggers run in a separate activation group (ACTGRP(*NEW)), those system triggers will not participate in the unit of the work for the calling application or 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 that are defined by the CREATE TRIGGER statement always run in the caller's activation group.

Note: SQL triggers always run in the caller's activation group.

If the triggering application is running with commitment control, the operations of an SQL trigger, and any cascaded SQL triggers, are captured in 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 are 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, also participate in the sub-unit of work. If the triggering application is running without commit control, the operations of the SQL triggers 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, an UPDATE, or a DELETE statement that encounters an error during the CREATE TRIGGER execution of the statement, no other system and SQL triggers will still be activated after the error for that operation.

However, a number of changes were already 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.

5.6.8 Datetime considerations

If OLD ROW or NEW ROW is specified, the date or time constants and the string representation of dates and times in variables that are used in SQL statements in the triggered-action must have a format of ISO, EUR, JIS, USA. Or, they must match the date and time formats that were specified when the table was created if it was created by using DDS and the **CRTPF** CL command. If the DDS specifications contain multiple different date or time formats, the trigger cannot be created.

5.6.9 Triggers and traditional record-level access

Important functional differences exist for SQL triggers compared to traditional record-level access triggers that are defined with the Add Physical File Trigger (**ADDPFTRG**) command. The following functions are available in SQL triggers that are not available in traditional external triggers:

- SQL syntax is more powerful so SQL triggers can be created more quickly and easily.
- SQL triggers support column-level granularity so that they can be activated only when a column or set of columns are updated. This granularity is better than the granularity that is provided by the Trigger Update Condition (**TRGUPDCND**) keyword on **ADDPFTRG**.
- SQL triggers offer more options to manage operations that modify multiple rows in the subject table or view.
- SQL triggers support both DB2SQL and DB2ROW modes.
- SQL triggers include INSTEAD OF triggers for views that are not available for external triggers.
- SQL syntax makes it easier to write and configure one trigger to process multiple events.
- SQL triggers are more portable.

Traditional external triggers offer features that are not available in SQL:

- Information is provided to trigger programs that is not available to SQL, but this information is not relevant or useful to SQL:
  - Trigger time
  - Commit level
  - Relative record number
- SQL triggers do not support the concept of Read triggers.
- SQL triggers do not provide an equivalent option to the Allow Repeated Change (**ALWREPCHG**) parameter on **ADDPFTRG**, which traditional external triggers use to control whether the trigger can modify row data. SQL triggers always allow these modifications in BEFORE triggers. This capability corresponds to the ALWREPCHG(*YES) value.

The following considerations apply to the use of SQL triggers in an environment where programs use traditional record-level access:

- For native database changes, no difference exists between row level and statement level because native databases can perform only single row operations. Even MODE DB2SQL triggers are executed on each row simply because record-level access does not support multiple row updates. Each operation is for a single record, but it is considered a statement.

5.6.10 Multiple triggers on the same table

DB2 for i allows multiple triggers for the same operation and the same activation time. For example, three triggers will be activated before an update on a specific table. To configure them correctly, it is important to understand that they will be activated based on their mode and the order in which they are created. The rules are shown:

- MODE DB2ROW triggers are activated first in the order in which they were created. This rule also includes traditional external triggers that are configured by using the Add Physical File Trigger (**ADDPFTRG**) CL command.
- MODE DB2SQL triggers are activated next, and they are also activated in the order in which they were created.

When a trigger is replaced by using the CREATE OR REPLACE syntax, its position in the activation order is not maintained. This rule operates the same as a DROP TRIGGER that is followed by a CREATE TRIGGER.

Note: Do not define triggers that depend on the order in which they are executed if you can avoid it.

5.7 Trigger-related catalogs

The SQL catalogs include four catalogs that are specific to SQL triggers. The instances in the library QSYS2 pertain to all triggers in the relational database. The instances in SQL schemas provide information about triggers in that schema only. The trigger catalogs are shown.

SYSTRIGGERS

The SYSTRIGGERS view contains one row for each trigger. It includes columns that indicate the trigger name, the object for which the trigger was configured (table or view), the trigger event, and the trigger timing.

Example 5-26 shows a sample query for the SYSTRIGGERS view.

Example 5-26 SYSTRIGGERS sample query

```
SELECT trigger_name, event_manipulation, event_object_table, action_timing
   FROM systriggers WHERE trigger_schema='JIMD'  ;
```

Figure 5-17 shows the output of the SYSTRIGGERS sample query.

TRIGGER_NAME	EVENT_MANIPULATION	EVENT_OBJECT_TABLE	ACTION_TIMING
EXT_DEPT	INSERT	DEPARTMENT	BEFORE
MANAGE_EMPLOYEE	MULTI	EMPLOYEE	AFTER
NORMALIZE_ADDRESS	MULTI	EMPLOYEE	BEFORE
VALIDATE_PROJECT_DATA	INSERT	PROJECT	BEFORE

Figure 5-17 Output of the SYSTRIGGERS sample query

SYSTRIGCOL

The SYSTRIGCOL view contains a row for each column that is referenced in the WHEN clause or in the body of a trigger.

Example 5-27 shows a sample query for the SYSTRIGCOL view.

Example 5-27 SYSTRIGCOL sample query

```
SELECT trigger_name, table_name, object_type, column_name
   FROM systrigcol WHERE trigger_schema = 'JIMD';
```

Figure 5-18 shows the output of the SYSTRIGCOL sample query.

TRIGGER_NAME	TABLE_NAME	OBJECT_TYPE	COLUMN_NAME
EXT_DEPT	DEPARTMENT	TABLE	EXTDEPTNO
EXT_DEPT	DEPARTMENT	TABLE	DEPTNO
MANAGE_EMPLOYEE	EMPLOYEE	TABLE	EMPNO
MANAGE_EMPLOYEE	EMPLOYEE	TABLE	SALARY
NORMALIZE_ADDRESS	EMPLOYEE	TABLE	MAILING_ADDRESS
VALIDATE_PROJECT_DATA	PROJECT	TABLE	PROJNAME

Figure 5-18 Output of the SYSTRIGCOL sample query

The column OBJECT_TYPE contains the type of object that contains the column. The possible values are FUNCTION (for a table function), MATERIALIZED QUERY TABLE, TABLE, and VIEW.

SYSTRIGDEP

The SYSTRIGDEP view contains one row for each object that is referenced in the WHEN clause or in the body of a trigger.

Example 5-28 shows a sample query for the SYSTRIGDEP view.

Example 5-28 SYSTRIGDEP sample query

```
SELECT trigger_name, object_name, object_type
   FROM systrigdep WHERE trigger_schema = 'JIMD';
```

Figure 5-19 shows the output of the SYSTRIGDEP sample query.

TRIGGER_NAME	OBJECT_NAME	OBJECT_TYPE
EXT_DEPT	GV_DIV	VARIABLE
MANAGE_EMPLOYEE	NEW_EMPLOYEE	PROCEDURE
MANAGE_EMPLOYEE	EXIT_EMPLOYEE	PROCEDURE
MANAGE_EMPLOYEE	SALARY_AUDIT	PROCEDURE
NORMALIZE_ADDRESS	NORMALIZE	FUNCTION

Figure 5-19 Output of the SYSTRIGDEP sample query

OBJECT_TYPE indicates the type of the object that the trigger depends on. The possible values are listed:

- ALIAS
- FUNCTION
- INDEX
- MATERIALIZED QUERY TABLE
- PACKAGE
- PROCEDURE
- SCHEMA
- SEQUENCE
- TABLE
- TYPE
- VARIABLE
- VIEW

SYSTRIGUPD

The SYSTRIGUPD view contains one row for each column that is identified in the UPDATE column list, if any.

Example 5-29 shows a sample query for the SYSTRIGUPD view.

Example 5-29 SYSTRIGUPD sample query

```
SELECT trigger_name, event_object_table, event_object_table, triggered_update_columns
   from SYSTRIGUPD WHERE trigger_schema = 'JIMD';
```

Figure 5-20 shows the output of the SYSTRIGUPD sample query.

TRIGGER_NAME	EVENT_OBJECT_TABLE	EVENT_OBJECT_TABLE	TRIGGERED_UPDATE_COLUMNS
MANAGE_EMPLOYEE	EMPLOYEE	EMPLOYEE	SALARY

Figure 5-20 Output of the SYSTRIGUPD sample query

6

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 similar to another system-supplied function that can be used in Structured Query Language (SQL) statements.

This chapter includes the following topics:

- Introduction
- Nature of user-defined functions
- Types of user-defined functions
- Structure of an SQL UDF
- CREATE FUNCTION syntax for SQL scalar and table functions
- Resolving a UDF
- System catalog tables and views
- UDF examples
- UDF inlining
- UDTF examples
- Pipelined table functions
- Coding considerations: UDF preferred practices
- SQL control statements
- Handling errors in SQL UDFs

Debugging techniques are common to all SQL routines. They are described in 7.2, “Debug SQL routines and triggers” on page 203.

6.1 Introduction

UDFs are host-language functions for performing customized, often-used tasks in applications. UDFs allow the programmers to modularize a database application to create 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 different requirements that are not covered by the built-in functions. UDFs play an important role by allowing users and programmers to enrich the database manager by providing their own functions.

UDFs offer the following advantages:

- Customization

  You can create functions that are required by your application but that do not exist in the set of DB2 built-in functions. 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.

- Performance

  A UDF can run in the database engine. It is useful for performing calculations in the database manager server. Another area where performance might increase is working with large objects (LOBs). UDFs can be used for extracting or modifying 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.

- Portability

  When you migrate data from other database managers, built-in functions might not be defined in DB2 for i. You can use UDFs to create those functions to port the data more easily.

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 platform-specific functions.

- Handle user-defined data types

  You can implement the behavior of a user-defined 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 you can have two or more functions with the same name in the same library. For example, you can have several instances of the SUBSTR function that accept different data types as input parameters. Function overloading is one the key features that are required by object-oriented programming.

- Allow code reuse and sharing

  A business logic rule that is implemented as a UDF becomes part of the database, and it can be accessed by any interface or application by using SQL.

6.2 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, concatenation) that is 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 two groups:

- User-defined scalar functions
- User-defined table functions

Functions are created with the CREATE <OR REPLACE> FUNCTION SQL statement.

6.2.1 User-defined scalar functions

User-defined scalar functions are UDFs that return a single scalar value. Scalar UDFs are called from almost any SQL statement, and they are used in a SELECT list or in a WHERE clause. They can be useful for encapsulating complex operations or calculations, making them available to all database users in any part of an application.

The statement in Example 6-1 uses a scalar function that returns the temperature in Celsius for a specific temperature in Fahrenheit.

Example 6-1 Scalar UDF that is used to convert Fahrenheit to Celsius

```
SELECT
   f_to_c(temperature)
   FROM climate_info
```

The scalar function in the example is F_TO_C.

6.2.2 User-defined table functions

User-defined table functions (UDTFs) are UDFs that can return a set of output values in a row and columnar format. This set of output values is known as a *table* or *result set*. Examples of this type of function are shown:

- 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

UDTFs are invoked from an SQL FROM clause, and they can be referred to in a view for ease of use. They are good for encapsulating non-traditional data or complex operations.

Note: One useful and important use of a table function is the ability to access data in non-relational objects with an SQL statement. 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 as though the data came from a table that was created by SQL.

For example, the system-provided XMLTABLE function returns a table from the evaluation of an XPath expression.

6.3 Types of user-defined functions

UDFS can be divided into three categories:

- Sourced UDFs
- SQL UDFs
- External UDFs

In this book, we describe only sourced and SQL UDFs. External UDFs are described in *External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i*, SG24-6503.

6.3.1 Sourced UDFs

Sourced UDFs are functions that are registered to the database that simply make a reference to another function. In fact, they map to the sourced function, which means that no coding is involved. Nothing more is required in implementing a sourced UDF than registering it to the database by using the CREATE OR REPLACE FUNCTION statement. Sourced UDFs are often used to implement the required behavior of UDTs.

You can define a sourced UDF over an arithmetic operator, such as +, -, *, /, or ||, which 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, based on the standard “+”(DECIMAL, DECIMAL) that returns DECIMAL. See Example 6-2.

Example 6-2 Sourced function + for MONEY UDT

```
CREATE OR REPLACE FUNCTION Library/“+”(MONEY, MONEY)
   returns MONEY
   specific plus00001
   source qsys2.“+”(decimal, decimal);
```

6.3.2 SQL UDFs

SQL UDFs are functions that are written entirely by using the SQL programming 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.
- The definition of the interface between the database and the function is by using SQL declarations. You do not need to worry about the details of the actual parameter passing.
- You can use an SQL UDF to pass LOBs and UDTs as parameters and manipulate them in the function itself.

For example, in your EMPLOYEE table, three columns contain the employee's first name, middle initial, and last name, but you need to present this information as a single string with last name and first name separated by commas. Use a SELECT, as shown in Example 6-3.

Example 6-3 Trimming and concatenating in a SELECT statement

```
SELECT
   RTRIM(lastname) CONCAT ', ' CONCAT RTRIM(firstnme) CONCAT ' '  CONCAT midinit
   FROM employee;
```

Alternatively, to avoid repeating this operation every time that you need to present this formatted information, define a UDF to perform the operation. Change the previous SELECT statement to the SELECT statement in Example 6-4.

Example 6-4 SELECT statement that uses a UDF to trim and concatenate data

```
SELECT
   format_employee_name(firstnme, midinit, lastname)
   FROM employee;
```

The SQL UDF for this operation is shown in Example 6-5.

Example 6-5 SQL Format_employee_name UDF

```
CREATE OR REPLACE FUNCTION Format_employee_name (
   firstnme VARCHAR(12),
   midinit CHAR(1),
   lastname VARCHAR(15)
   )
   RETURNS VARCHAR(128)
   NO EXTERNAL ACTION
   NOT FENCED
   DETERMINISTIC
   SPECIFIC simona.emp_form
      -- ##################################################################
      -- # Formats employee name as Lastname, Firstnme I
      -- ##################################################################
   RETURN RTRIM(lastname) CONCAT ', ' CONCAT RTRIM(firstnme) CONCAT
   ' '  CONCAT midinit;
```

SQL UDFs are also useful when you want to see the result of a query as a table. In 6.10, "UDTF examples" on page 177, we show a table with a group of employees who work on a project.

6.4 Structure of an SQL UDF

SQL functions are UDFs that you define, write, and register by using the CREATE FUNCTION statement. They are written by using the SQL language only. Their definition is completely contained within one CREATE FUNCTION statement. The creation of an SQL function registers the UDF, generates the executable code for the function, and defines the details of how parameters are passed to the UDF or UDTF in the database catalog. Therefore, writing these functions is clean.

An SQL UDF consists of these components:

- A function name.
- A sequence of parameter declarations.
- The option list or function properties.
- The set option statement, which specifies the parameters to use to create the function, for example, to create a function that can be debugged. This component is optional.
- The SQL routine body that specifies a single SQL statement, including a compound statement.

Example 6-6 shows the general structure of a UDF.

Example 6-6 CREATE FUNCTION prototype statement

```
CREATE FUNCTION name-of-udf                1
   (List of the input parameters)          2
Returns                                    3
Function properties                        4
Generation options                         5
Routine body                               6
```

Notes about Example 6-6:

1 Every UDF starts with CREATE FUNCTION and its name. The UDF signature consists of the fully qualified name in combination with the parameter data types. Two UDFs with the same signature cannot reside on the same schema.

2 Parameters of the UDF and the data type of each parameter. A maximum of 1,024 parameters are allowed in this list. Parameter defaults can be defined, or a function can have no parameters. In this case, a set of empty parentheses () is needed.

3 Data type and attributes of the output return. You can specify any built-in data type (except LONG VARCHAR or LONG VARGRAPHIC), distinct type, or an array. Certain restrictions apply if the return type of a function is an array type.

4 You can define certain properties of the UDF, such as the language that is used to code the UDF. LANGUAGE SQL is optional. You can specify other characteristics:

- Specific name
- Whether it is deterministic
- Debug enabling
- Fenced or unfenced
- Concurrent access behavior
- Parallelism
- External actions
- Whether the UDF is secured, wrapped, and level of data access (whether the UDF contains SQL, reads SQL data, or modifies SQL data)

5 You can affect the way that the function is generated by DB2 for i by using the SET OPTION clause. One of the most relevant options is DBGVIEW, which forces the system to generate debugging information. For more information, see the DB2 for i SQL reference at https://ibm.biz/Bd42dh.

6 The routine body of the function consists of a single SQL statement (RETURN, SELECT, UPDATE, INSERT, or DELETE) or an SQL compound statement (IF, WHEN, FOR, or CASE) that must contain at least one RETURN statement.

6.5 CREATE FUNCTION syntax for SQL scalar and table 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 the language of the function, and certain behavioral characteristics, such as whether the function is deterministic, if it can be used in parallel, and if it reads or modifies SQL data.

The following section describes the syntax for CREATE FUNCTION. The syntax is explained step-by-step with the description that follows each part of the syntax diagram. Development tools might provide wizards or templates to facilitate the process of creating functions. Figure 6-1 shows the first part of the CREATE FUNCTION syntax for an SQL scalar function (a function that returns a single value).

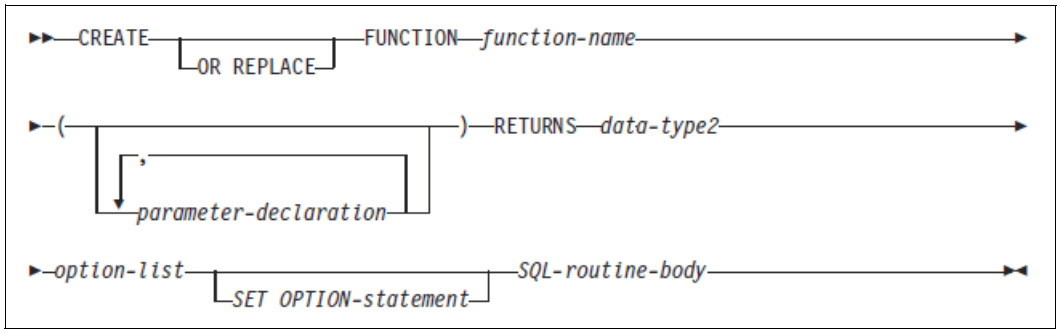

Figure 6-1 CREATE FUNCTION syntax for the SQL scalar function

Every function creation statement starts with CREATE FUNCTION and its name.

Function name

The function name is the UDF name. The combination of name, schema name, the number of parameters, and the data type of each parameter (without consideration for any length, precision, scale, or coded character set identifier (CCSID) attributes of the data type) must not identify a UDF that exists at the current server. For the naming behavior, see 3.2.1, “Routine and trigger creation process” on page 55.

The function name can be either qualified or unqualified. You can have two or more functions with the same name in the same schema, library, or collection, if they have a different *signature*. A signature is the combination of name, schema name, the number of parameters, and the data type of each parameter (without consideration for any length, precision, scale, or CCSID attributes of the data type). A signature must be unique. See 6.6.1, “UDF overloading and function signature” on page 166.

Parameter-declaration

The function name is followed by the input parameter declaration. See Figure 6-2.

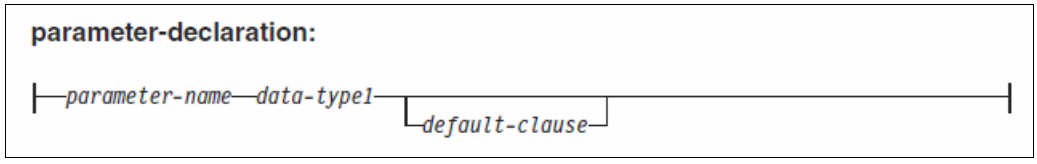

Figure 6-2 CREATE FUNCTION syntax for the SQL function’s parameter declaration

The parameter declaration specifies the number of input parameters of the function and the data type of each parameter. Although the parameter name is not required, you can give each parameter a name. The parameters can be specified in any order by specifying the parameter name on the invocation.

The maximum number of parameters that are allowed in CREATE FUNCTION is 1,024 for SQL scalar and SQL table functions (90 for all other functions). For UDTFs, the return columns are considered parameters and the maximum number is now 1025 - the number of parameters. A function can have no input parameters. In this case, an empty set of parentheses must be specified. The defaults for parameters can be set.

Tip: For the portability of functions across other DB2 products, do not use the following data types, which might have different representations on different platforms:

- FLOAT: Use DOUBLE or REAL instead.
- NUMERIC: Use DECIMAL instead.

When you choose the data types of the input and result parameters for a function, consider the rules of promotion that can affect the values of the parameters. For more information, see 6.6.2, “Parameter matching and promotion” on page 166.

Parameter defaults

With the DEFAULT clause, it is possible to specify a default value for the parameter. This support is similar to the support that is provided for SQL procedures as of IBM 7.1.

The default can be one of the following objects:

- A constant
- A special register
- A global variable
- An expression
- The keyword **NULL**

When you invoke a function, parameters can be omitted if the routine was defined with a default value. If a default value is not specified, the parameter has no default and it cannot be omitted on the invocation.

When you create a function, the default expression must be assignment-compatible with the parameter data type.

Note: Any comma in the default expression that is intended as a separator of numeric constants in a list must be followed by a space.

All objects that are referenced in a default expression must exist when the function is created. A default cannot be specified for a parameter of type array.

RETURNS

Figure 6-3 shows the next clause, RETURNS. It describes the output of the function. The syntax slightly differs in a SCALAR function and a TABLE function.

Figure 6-3 CREATE FUNCTION syntax for SQL function RETURNS

In a SCALAR function, RETURNS can be considered similar to an output parameter and the output type must also be defined.

In a TABLE function, the output is a virtual table and all result columns must be defined. The return columns are considered parameters and the maximum number of columns is 1,025 - the number of input parameters.

Option list

Use the option list to set options and environmental attributes to use when you execute the function. All options in the option list can be specified in any order but each clause can appear only one time. See Figure 6-4.

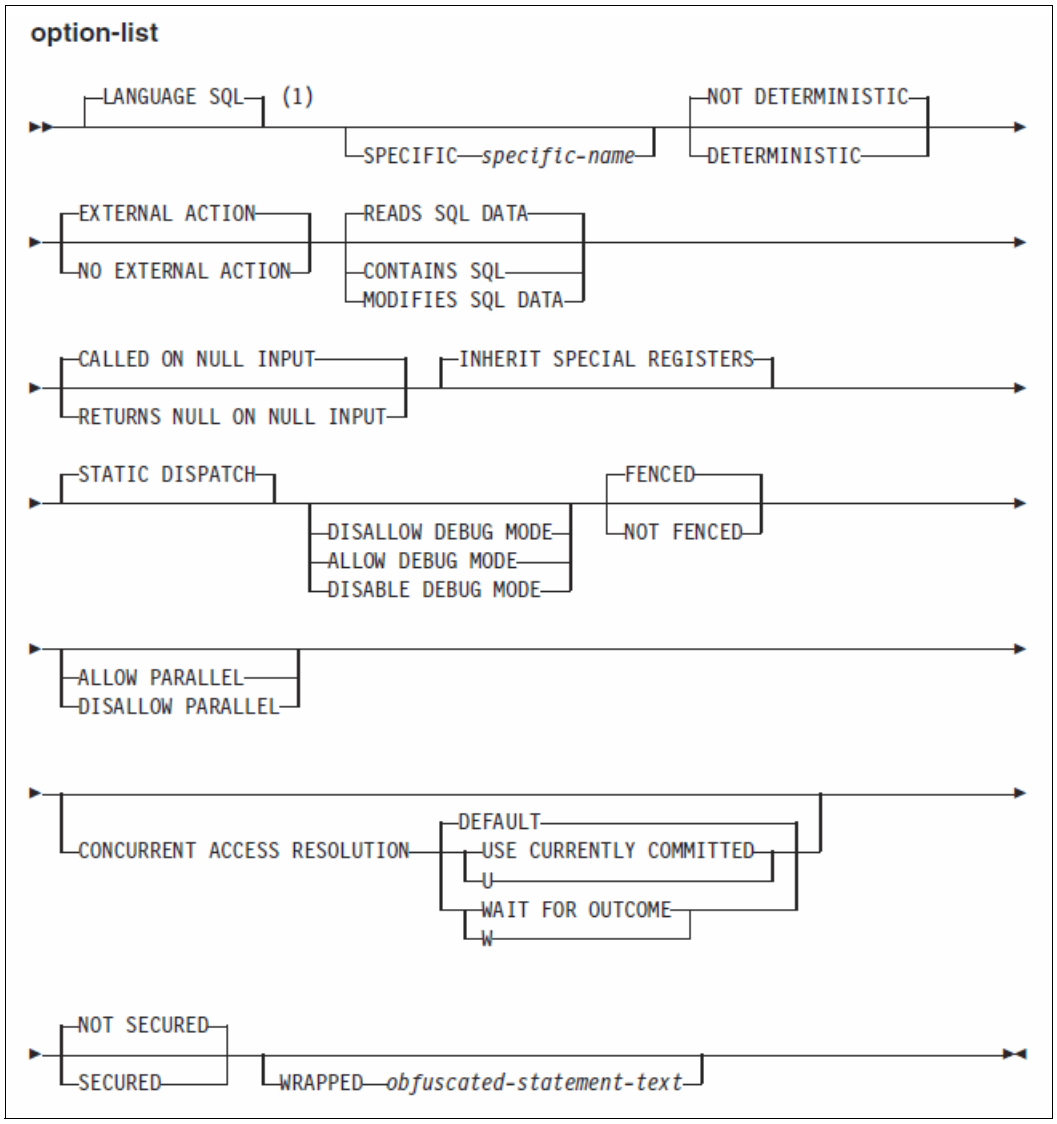

Figure 6-4 CREATE FUNCTION syntax for SQL function OPTION-LIST

The option list syntax in Figure 6-4 on page 160 is explained:

- LANGUAGE

  LANGUAGE specifies the language interface convention to which the function body is written. In this environment, SQL is specified.

- SPECIFIC *specific-name*

  SPECIFIC *specific-name* specifies a unique name for the function.

  When you define multiple functions with the same name and schema but different parameters (6.6, “Resolving a UDF” on page 166), we recommend that you also specify a specific name. The specific name can be used to uniquely identify the function, such as when sourcing on this function, dropping the function, or commenting on the function. However, the function cannot be invoked by its specific name.

  The specific name is implicitly or explicitly qualified with a schema name. If a schema name is not specified on CREATE FUNCTION, it is the same as the explicit or implicit schema name of the function name (*function-name*). If a schema name is specified, it must be the same as the explicit or implicit schema name of the function name. The name, including the schema name, must not identify the specific name of another function or procedure that exists at the current server.

  If the SPECIFIC clause is not specified, a specific name is generated.

  Important: 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.

- 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. By specifying DETERMINISTIC, you enable DB2 for i to use cached results instead of calling the function, which might provide a performance advantage.

  A UDF that returns the temperature in Celsius from a Fahrenheit temperature is deterministic. No matter under what circumstances it is called, it always will return 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.

- 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). Therefore, the function must be invoked with each successive function invocation. Specify EXTERNAL ACTION if the function contains a reference to another function that has an external action. An example of an external action can be to insert a row to a table or to place 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.

- READS SQL DATA, CONTAINS SQL, 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.

- CALLED ON NULL INPUT or RETURNS NULL ON NULL INPUT

  This clause specifies whether the function is called if any of the input arguments is null at execution time:

  - RETURNS NULL ON NULL INPUT specifies that the function is not invoked if any of the input arguments is null. The result is the null value.
  - CALLED ON NULL INPUT specifies that the function will be invoked if any or all argument values are null. This specification means that the function must be coded to test for null argument values. The function can return a null or nonnull value.

- INHERIT SPECIAL REGISTERS

  The existing values of special registers are inherited upon entry to the function.

- STATIC DISPATCH

  All functions are statically dispatched.

- DISALLOW DEBUG MODE, ALLOW DEBUG MODE, or DISABLE DEBUG MODE

  This clause indicates whether the function is created so that it can be debugged by the Unified Debugger. If DEBUG MODE is specified, a DBGVIEW option in the SET OPTION statement must not be specified.

  - DISALLOW DEBUG MODE

    The function cannot be debugged by the Unified Debugger. This option can be changed by using ALTER FUNCTION.

  - ALLOW DEBUG MODE

    The function can be debugged by the Unified Debugger. This option can be changed by using ALTER FUNCTION.

  - DISABLE DEBUG MODE

    The function cannot be debugged by the Unified Debugger. When the DEBUG MODE attribute of the function is DISABLE, the function cannot be altered later to change the debug mode attribute.

  If FENCED or ALLOW PARALLEL is specified for the function, the DEBUG MODE option will be ignored. DISALLOW DEBUG MODE will be used.

  If DEBUG MODE is not specified, but a DBGVIEW option in the SET OPTION statement is specified, the function cannot be debugged by the Unified Debugger but it might be debugged by the system debug facilities. If DEBUG MODE is not specified and a DBGVIEW option is not specified, the debug mode that is used is from the CURRENT DEBUG MODE special register.

- 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 can run in the same thread as the invoking SQL statement. NOT FENCED functions can keep SQL cursors open across individual calls to the function and typically provide better performance. 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 is running the SQL statement.

  By default, UDFs are created as FENCED. This option ensures that potential problems are avoided when the same cursor name is used repeatedly inside a complex UDF, which might cause cursor name conflicts when the procedure is executed.

  A UDF that was 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 NOT FENCED versus FENCED UDFs provides better performance because the original query and the UDF can run within the same thread.

  Coding preferred practice suggests that you give cursors meaningful names.

- PARALLEL

  The PARALLEL parameter indicates whether the function can be run in a parallel implementation of the query (if the optimizer chooses to do so). Therefore, it applies only when DB2 symmetric multiprocessing (SMP) is installed and activated. The same UDF program can be running in multiple threads at the same time. Therefore, if ALLOW PARALLEL is specified for the UDF, ensure that it is 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.

- CONCURRENT ACCESS RESOLUTION

  This clause specifies whether the database manager needs to wait for data while the data is updated. DEFAULT is the default.

  - DEFAULT

    DEFAULT specifies that the concurrent access resolution is not explicitly set for this function. The value that is in effect when the function is invoked will be used.

  - WAIT FOR OUTCOME

    WAIT FOR OUTCOME specifies that the database manager will wait for the commit or rollback of data while the data is updated.

  - USE CURRENTLY COMMITTED

    USE CURRENTLY COMMITTED specifies that the database manager will use the currently committed version of the data when it encounters data that is being updated. When the lock contention is between a read transaction and a delete or update transaction, the clause applies to scans with the cursor stability (CS) isolation level (but not for CS KEEP LOCKS).

- SECURE

  SECURE specifies whether the function is considered secure for row access control and column access control:

  - NOT SECURED

    NOT SECURED specifies that the function is considered not secure for row access control and column access control. This specification is the default. When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled when the table is using active column access control.

  - SECURED

    SECURED specifies that the function is considered secure for row access control and column access control. A function must be defined as secure when it is referenced in a row permission or a column mask.

    From the DB2 for i SQL reference manual:

    Invoking other user-defined functions in a secure function: When a secure user-defined function is referenced in an SQL data change statement that references a table that is using row access control or column access control, and if the secure user-defined function invokes other user-defined functions, the nested user-defined functions are not validated as secure. If those nested functions can access sensitive data, a user that is authorized to the Database Security Administrator function of IBM i needs to ensure that those functions are allowed to access that data and the user needs to ensure that a change control audit procedure was established for all changes to those functions.

- WRAPPED *obfuscated-statement-text*

  WRAPPED allows independent software vendors (ISVs) to prevent proprietary code source to be retrieved by using the QSYS2.GENERATE_SQL() system-provided procedure to generate an encoded definition of the function. A CREATE FUNCTION statement can be encoded by using the WRAP scalar function.

Set option

In Figure 6-5, you can see the final part of the CREATE FUNCTION statement, which is represented by the SET OPTION clause. This part is common to other components of SQL Persistent Stored Modules (PSM), and it is described in “SET OPTION” on page 57.

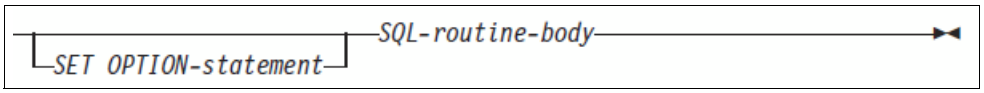

Figure 6-5 CREATE FUNCTION syntax for SQL function SET OPTION

SQL routine body

The SQL routine body includes the actual statements and action requests. The SQL routine body can consist of one or multiple SQL statements, including control statements.

6.5.1 Modifying or dropping a UDF

The ALTER FUNCTION statement can be used to alter or modify a UDF. Its syntax is similar to the CREATE FUNCTION statement, and it is documented in the DB2 for i SQL reference.

Alternatively, you can first DROP and then CREATE or use the CREATE OR REPLACE FUNCTION statement.

The following actions occur when a function is re-created by REPLACE:

- Any existing comment or label is discarded.
- Authorized users are maintained. The object owner can change.
- Current journal auditing is preserved.

To drop a UDF by using the SQL interface, use the DROP FUNCTION statement.

The DROP FUNCTION statement references the function by one of these variables:

- *Name*: An example is DROP FUNCTION myUDF. This name is only valid if exactly 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): An example is DROP FUNCTION myUDF(int). The data type of the parameters must match exactly the data type of the parameters of the found function. Also, if length, precision, and scale are specified, they must match exactly the function to be dropped. *SQLSTATE* `42883` is signaled if a match to an existing function is not found.
- *Specific name*: An example is DROP SPECIFIC FUNCTION myFun0001. Because the SPECIFIC name must be unique for each schema, this specific name will find, at most, one function. If the function is not found, *SQLSTATE* `42704` (`'Function not found'`) is signaled.

Similar considerations might apply to the ALTER FUNCTION statement.

Note: Also, you can use the new *IBM Navigator for i* web interface to perform database management tasks, such as dropping functions.

6.6 Resolving a UDF

Resolving to the correct function to use for an operation is more complicated than other resolution operations because DB2 for i 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 a user can define the user's 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.

6.6.1 UDF overloading and function signature

DB2 for i supports the concept of function overloading. Function overloading means that two or more functions with the same name in the same schema, library, or collection can exist, if their *signatures* differ. The *signature* of a function is 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 part of the signature. Only the data type of the input parameters is considered part of the signature. Therefore, if a function that is called DNAME in schema SAMPLEDB01 accepts an input parameter of type CHAR(10), another function that is called DNAME cannot exist in the same SAMPLEDB01 that accepts CHAR(12). However, it is possible for a function that is called DNAME in library SAMPLEDB01 that accepts an INTEGER value as an input parameter to exist with another function that is called DNAME in library SAMPLEDB01 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))
```

Note: Certain data types are considered equivalent for function signatures. For example, CHAR and GRAPHIC are treated as the same data type from the signature perspective.

The data type of the value that is returned by the function is *not* considered part of the function signature. Therefore, two functions that are called DNAME in library SAMPLEDB01 that accept input parameters of the same data type cannot both coexist, even if they return values of different data types.

6.6.2 Parameter matching and promotion

When an SQL Data Manipulation Language (DML) statement references a UDF, the system, at first, tries to find a match for the function by searching for functions with the same signature. If the system finds a function with input parameters that match the 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 matches the input parameters that are specified on the DML statement, the parameters on the function call in the DML statement are *promoted* to their next higher type. Then, another search is made for a function that accepts the promoted parameters as input.

During parameter promotion, a parameter is cast to its next higher data type. For example, a parameter of type CHAR is promoted to VARCHAR, and then to CLOB. 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 CUSTOMER table in library `LIB1`. This table has, among its other columns, a column that is named CUSTOMER_NUMBER, which is a CHAR(5). Assume that you wrote a function GetRegion that will perform processing 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. Now, if you execute the following query, you can see that the function GetRegion(CLOB(50K)) was executed:

```
select GetRegion( customer_number ) from customer
```

How was the function executed? 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 called GetRegion, which accepts an input parameter of type CHAR. However, this UDF was not 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 GetRegion UDF, which accepted an input parameter of type VARCHAR. Again, no 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 clearly demonstrated in the previous example. Table 6-1 indicates the data types and the data types to which they can be promoted.

Table 6-1 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, DOUBLE
INTEGER	INTEGER, DECIMAL or NUMERIC, REAL, DOUBLE
DECIMAL or NUMERIC	DECIMAL or NUMERIC, REAL, DOUBLE
REAL	REAL, 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 6-1.

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
```

6.6.3 Function path and the function selection algorithm

On IBM i, two types of naming conventions are available when you use SQL. The two naming conventions are described in 3.2, “Common information for SQL routines and triggers” on page 55. One naming convention is called the *system-naming convention*, and the other naming convention is called the *SQL naming convention*. The system-naming convention is native to the IBM i, and the SQL naming convention is specified by the American National Standards Institute (ANSI) SQL standard.

The function resolution process depends on the naming convention that you use when you execute the SQL statement that 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 to resolve 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", "SYSPROC", "SYSIBMADM", "<USER ID>"`

For the system-naming convention, the path is set initially to the following default value:

`*LIBL`

When you use the system-naming convention, the system uses the library list of the current job as the path, and it 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, SYSPROC, SYSIBMADM, MYUDFS, COMMONUDFS`

The libraries `QSYS`, `QSYS2`, `SYSPROC`, and `SYSIBMADM` are automatically positioned 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, we recommend that you use the SYSTEM PATH registry rather than the `QSYS` and `QSYS2` library names on the SET PATH statement.

Function selection algorithm

The function selection algorithm searches the library path for a UDF by using the following steps:

1. Find all of the functions from the catalog (SYSFUNCS) and built-in functions that match the name of the function. If a library was specified, the algorithm gets those functions from that library only. Otherwise, it gets all functions whose library is in the function path.
2. Eliminate those functions whose number of defined parameters does not match the invocation.
3. Eliminate 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. Parameters can be promoted and casted.

2. If more than one candidate function remains after the previous steps, all of the remaining candidate functions have identical signatures but they are in different schemas. The algorithm works in this manner. The algorithm chooses the function whose schema is earliest in the user's function path.
3. If more than one candidate function remains and if one candidate function has a number of parameters that is fewer than or equal to the number of parameters of the other candidate functions, those candidate functions with a greater number of parameters are eliminated, considering the number of arguments in the function invocation.
4. If no unique identification occurs and arguments that are untyped expressions exist, they are considered (from left to right) to eliminate candidate functions.
5. If multiple candidate functions still exist, an error is returned.

After the function is selected, reasons still exist why the use of the function might not be permitted. Each function is defined to return a result with a specific data type. If this result data type is incompatible within the context in which the function is invoked, an error occurs.

Note: Resolution rules are articulated. The steps are merely a summary overview. For more information, see DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

6.7 System catalog tables and views

The database manager provides many data dictionary facilities to track UDFs, as shown in Table 6-2.

Table 6-2 Functions: Catalog tables and views

Catalog names	Area
QSYS2.SYSFUNCS	IBM i catalog tables and views
QSYS2.SYSROUTINES	IBM i catalog tables and views
QSYS2.SYSROUTINEDEP	IBM i catalog tables and views
QSYS2.SYSCONTROLSDEP	IBM i catalog tables and views
QSYS2.SYSPARMS	IBM i catalog tables and views
SYSIBM.SQLFUNCTIONS	ODBC and JDBC catalog views
SYSIBM.SQLFUNCTIONCOLS	ODBC and JDBC catalog views
QSYS2.ROUTINES	ANSI and ISO catalog views

Note: The SYSROUTINES catalog contains details for both UDFs and stored procedures. If you want to see UDFs only, you can use the SYSFUNCS view, or you can select rows in the SYSROUTINES catalog where ROUTINE_TYPE is FUNCTION.

For detailed descriptions of the DB2 for i catalogs, see DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

We describe how to query UDF information by using the SYSFUNCS view and the SYSPARMS table.

6.7.1 SYSFUNCS catalog

UDF references are stored in various catalogs. A simple and efficient way to understand a database environment and to check whether specific parameters were set when a function was created is by querying the system catalogs. Different information can be gathered from the various catalog objects that are available. Example 6-7 shows querying specific columns in the SYSFUNCS catalog view. For example, you might want to check whether a function is INLINE-capable.

Example 6-7 SYSFUNCS catalog query

```
SELECT
   routine_name, specific_schema, specific_name,  is_deterministic, sql_data_access ,
   function_origin, function_type , parallelizable,  fenced,  inline, secure
   FROM qsys2.sysfuncs
   WHERE routine_schema  = 'SIMONA' ;
```

Figure 6-6 shows a sample of the content in the catalog. The INLINE column is highlighted.

ROUTINE_NAME	SPECIFIC_SCHEMA	SPECIFIC_NAME	IS_DETERMINISTIC	SQL_DATA_ACCESS	FUNCTION_ORIGIN	FUNCTION_TYPE	PARALLELIZABLE	FENCED	INLINE	SECURE
FORMAT_EMPLOYEE_NAME	SIMONA	EMP_FORM	YES	READS	E	S	YES	NO	YES	N
FORMAT_EMPLOYEE_NAME	SIMONA	EMP_SHORT	YES	READS	E	S	YES	NO	YES	N
FORMAT_EMPLOYEE_NAME_1	SIMONA	FORMAT_EMPLOYEE_NAME_1	NO	READS	E	S	NO	YES	NO	N

Figure 6-6 Sample content of SYSFUNCS catalog

6.7.2 SYSPARMS catalog

The SYSPARMS catalog contains one row for each parameter of a UDF that was created by the CREATE FUNCTION statement.

We want to retrieve the parameter details for all instances of the FORMAT_EMPLOYEE_NAME function, which is in our sample schema. We run the SQL statement in Example 6-8 to display this information.

Example 6-8 Sample query on SYSPARMS to show the parameters for the UDFs

```
SELECT b.routine_name,  a.specific_name,  parameter_name, parameter_mode, data_type
   FROM qsys2.sysparms a INNER JOIN  qsys2.sysfuncs b
   ON a.specific_schema=b.specific_schema and a.specific_name=b.specific_name
   WHERE b.specific_schema= 'SIMONA'
   and b.routine_name = 'FORMAT_EMPLOYEE_NAME'  ;
```

Due to function overloading, a schema might contain functions with the same routine name and a different specific name. Running this query produced the results that are shown in Figure 6-7, which show two instances of the FORMAT_EMPLOYEE_NAME function in the sample schema. Their signatures differ because one accepts three parameters 1 and the other function accepts only two parameters 1. On creation, you are required to specify a different SPECIFIC name in each case.

The result of the function is stored in the SYSPARMS catalog as an OUTPUT parameter 2.

ROUTINE_NAME	SPECIFIC_NAME	PARAMETER_NAME	PARAMETER_MO...	DATA_TYPE
FORMAT_EMPLOYEE_NAME	EMP_FORM	FIRSTNME	IN	CHARACTER VARYING
FORMAT_EMPLOYEE_NAME	EMP_FORM	MIDINIT	IN	CHARACTER
FORMAT_EMPLOYEE_NAME	EMP_FORM	LASTNAME	IN	CHARACTER VARYING
FORMAT_EMPLOYEE_NAME	EMP_FORM	-	OUT	CHARACTER VARYING
FORMAT_EMPLOYEE_NAME	EMP_SHORT	FIRSTNME	IN	CHARACTER VARYING
FORMAT_EMPLOYEE_NAME	EMP_SHORT	LASTNAME	IN	CHARACTER VARYING
FORMAT_EMPLOYEE_NAME	EMP_SHORT	-	OUT	CHARACTER VARYING

Figure 6-7 UDF parameter details in the SYSPARMS catalog

6.8 UDF examples

Simple examples of UDFs are described.

6.8.1 Simple scalar UDF

A simple scalar UDF is shown. This single SQL statement UDF returns a scalar value. Example 6-9 shows how the total salary is calculated by adding base salary, bonus, and commission.

Example 6-9 Simple scalar UDF

```
CREATE OR REPLACE FUNCTION total_salary (                          1
   salary decimal(9, 2) ,
   bonus decimal(9, 2) ,                                           2
   commission decimal(9, 2) )
   RETURNS decimal(9, 2)                                           3
   LANGUAGE SQL                                                    4
   SPECIFIC totsal01                                               5
   DETERMINISTIC
   CONTAINS SQL                                                    6
   NOT FENCED
   RETURN coalesce(SALARY, 0) + coalesce(BONUS, 0) + coalesce(COMMISION, 0); 7
```

Notes about Example 6-9 on page 172:

1. The name of the function in the example is TOTAL_SALARY.
2. A function might need input parameters. In this case, we defined three parameters: SALARY as the base salary, BONUS as the bonus, and COMMISION as the commission.
3. Every function returns a value. We specify the data type for the return to be DECIMAL(9,2).
4. The language that was used to code this UDF was SQL PSM.
5. In the *specific* clause, you define a unique name for the function. This option is useful when multiple functions with the same name and different signatures are on the same schema.
6. Additional options exist for the function, such as the options that are defined in 6.5, “CREATE FUNCTION syntax for SQL scalar and table functions” on page 157.
7. The last line is the SQL procedure body, which consists of a single add operation of the three values that were received as parameters.

This function has a single statement that performs an arithmetic operation on the three values that were received as parameters and returns the result. This UDF can be used in a SELECT, SET, or VALUES statement, or with clauses, such as the clauses that are shown in Example 6-10.

Example 6-10 Scalar UDF usage

```
SELECT firstnme, lastname, total_salary(salary, bonus, comm) as total  FROM employee;

-- give yourself and your team a wonderful salary increase! --
UPDATE employee SET salary = salary + total_salary(salary, bonus, comm) *0.5
   WHERE workdept = 'A01';


INSERT INTO ctrltbl (empno, workdept, total_comp)  VALUES (:EMPNO, :WORKDEPT,
   total_salary(:SALARY, :BONUS, :COMM));
```

6.8.2 More complex SQL statement UDF

More complex UDFs can help you with various situations:

- More than one SQL statement can be included in a single UDF. The SQL statements are enclosed between a BEGIN and END pair.
- Various operations can be implemented. SELECT, DELETE, INSERT, and UPDATE statements are supported.

In our first example in Example 6-10, we added three values. In reality, calculations are more complex. They cannot be solved with a single SQL statement.

In Example 6-11, we add complexity to Example 6-10 on page 173 by adding a business rule to enforce the payment of the commission, depending on an employee's project assignments.

Example 6-11 Compound SQL statement UDF

```
CREATE OR REPLACE FUNCTION totalsal (                                    1
   p_empno CHAR(6))                                                      2
   RETURNS DECIMAL(9, 2)                                                 3
   LANGUAGE SQL
   SPECIFIC totalsal2
   NOT DETERMINISTIC
   NOT FENCED
   READS SQL DATA
   RETURNS NULL ON NULL INPUT
   BEGIN                                                                 4
      DECLARE total DECIMAL (9, 2) ;
      DECLARE utlztn DECIMAL (5, 2);
      SELECT SUM ( emptime ) INTO utlztn FROM empprojact
         WHERE empno = p_empno;
      SELECT COALESCE(salary, 0) + COALESCE(bonus, 0) + COALESCE(utlztn * comm, 0)
         INTO total
         FROM employee WHERE empno = p_empno;
      RETURN TOTAL ;
   END;                                                                  5

COMMENT ON SPECIFIC FUNCTION totalsal2
   IS 'Total Salary' ;
```

Notes about Example 6-11:

1 The function name is TOTALSAL.

2 The function has one input parameter, which is employee number.

3 Similar to the previous TOTALSAL function, this function returns the total salary as a decimal.

4 In this example, the SQL UDF body consists of a compound SQL statement. Every compound statement starts with a BEGIN clause.

5 Every compound statement requires an END clause.

The use of this UDF is illustrated in the following statement in Example 6-12.

Example 6-12 Using the TOTALSAL UDF

```
SELECT firstnme, lastname, totalsal(empno) AS total FROM employee;

SET V_SALARY = totalsal(empno);
```

Note: When you write a UDF, consider that it is invoked for each row that matches the selection criteria for the SELECT or SET statement where it is used.

6.9 UDF inlining

UDFs can be *inlined*, which means that on the first call the expression in the RETURN statement can be “embedded” in the caller query and can be reexecuted without accessing external objects again. Inlining can provide a performance advantage.

This behavior relates to program temporary fix (PTF) SF99701 Level 6 in 7.2 Base.

UDFs can be inlined in the following specific circumstances:

- The function must contain only a RETURN statement.
- The RETURN statement cannot contain a scalar subselect or fullselect.
- The function must be DETERMINISTIC.
- The function cannot specify ATOMIC.
- The function must be re-created or altered after the installation of the PTFs.

An inline function is only copied (inlined) into a query in the following situations:

- The function references an object. The authority attributes of the function and the query must be compatible.
- The function is defined to run under the user’s authority (*USER).
- Both the function and query run under the owner’s authority (*OWNER), and the owner for both the function and query is the same owner.

The following attributes of the function must match the attributes of the query:

- DATFMT, DATSEP, TIMFMT, and TIMSEP
- SRTSEQ and LANGID (if SRTSEQ is *LANGIDSHR or *LANGIDUNQ)
- DECFLTRND (if a DECFLOAT field is used in the function)
- DECMPT and DECRESULT
- SQLPATH (if an object reference is in the function)
- The CCSID of constants (the source CCSID) in the function and query

If a function is inlined and it contains a reference to a special register, the value of the special register is the same as other references to the same special register in the query.

If a function is inlined and it is defined with MODIFIES SQL DATA, the MODIFIES SQL DATA attribute is ignored.

6.9.1 Examples of INLINE and NON INLINE UDFs

Example 6-13 shows a function that is NON INLINE-capable because it contains a SELECT statement.

Example 6-13 NON-INLINE FUNCTION with SELECT

```
(SELECT state20 FROM states WHERE state2=statecode);
```

In Example 6-14, you can see a function that is INLINE-capable. It contains a RETURN and no SELECT.

Example 6-14 INLINE function with RETURN

```
CREATE FUNCTION Discount(totalSales DECIMAL(11,2))
   RETURNS DECIMAL(11,2)
   LANGUAGE SQL
   DETERMINISTIC
   NOT FENCED
```

```
   RETURN
   (CASE WHEN totalSales>2000 THEN totalSales*.95 ELSE totalSales END);
```

In Example 6-15, SET is used, which prevents the function from being INLINE-capable.

Example 6-15 NON INLINE Function with SET

```
CREATE OR REPLACE FUNCTION Format_employee_name_1 (
   firstnme VARCHAR(12), midinit CHAR(1), lastname VARCHAR(15))
   RETURNS VARCHAR(128)
   NO EXTERNAL ACTION
   NOT FENCED
   F1: BEGIN ATOMIC
   DECLARE employee_name VARCHAR(128);
   SET employee_name = RTRIM(lastname) ||
   ', ' || RTRIM(firstnme) ||
   ' '  || midinit;
   RETURN employee_name;
   END ;
```

Example 6-16 shows you that the code in Example 6-15 can be modified to become INLINE-capable.

Example 6-16 INLINE FUNCTION

```
CREATE OR REPLACE FUNCTION Format_employee_name (
   firstnme VARCHAR(12), midinit CHAR (1), lastname VARCHAR(15))
   RETURNS VARCHAR(128)
   NO EXTERNAL ACTION
   NOT FENCED
   DETERMINISTIC
   SPECIFIC simona.emp_form
   RETURN
   RTRIM(lastname) CONCAT ', ' CONCAT RTRIM(firstnme) CONCAT ' '  CONCAT midinit;
```

The SYSFUNCS catalog contains a column that is named INLINE, which provides a YES/NO value that indicates whether the function can be inlined. Use the sample query in Example 6-17 to check whether this behavior is enabled.

Example 6-17 Sample statement for checking INLINE-capable functions

```
SELECT
   routine_name, specific_schema, specific_name,  is_deterministic, fenced,  inline, secure
   FROM qsys2.sysfuncs
   WHERE routine_schema  = 'SIMONA' ;
```

Figure 6-8 highlights that two of our functions are INLINE-capable.

ROUTINE_NAME	SPECIFIC_SCHEMA	SPECIFIC_NAME	IS_DETERMINISTIC	FENCED	INLINE	SECURE
FORMAT_EMPLOYEE_NAME	SIMONA	EMP_FORM	YES	NO	YES	N
FORMAT_EMPLOYEE_NAME	SIMONA	EMP_SHORT	YES	NO	YES	N
FORMAT_EMPLOYEE_NAME_1	SIMONA	FORMAT_EMPLOYEE_NAME_1	NO	YES	NO	N

Figure 6-8 Checking INLINE-capable functions

6.10 UDTF examples

Several examples are described of user-defined table functions (UDTFs). A UDTF results in a table.

6.10.1 Single SQL statement UDTF

Example 6-18 shows a single statement UDTF that returns a table of the employees that are involved in a specific project.

Example 6-18 Single SQL statement UDTF

```
CREATE FUNCTION empbyprj (
   prjnbr VARCHAR(6) )
   RETURNS TABLE (
      empno CHAR(6) ,
      firstnme CHAR(20) ,                                    1
      lastname CHAR(20) ,
      birthdate DATE)
   LANGUAGE SQL
   NOT FENCED
   DETERMINISTIC
   SPECIFIC empbyprj
      RETURN
         SELECT empno, firstnme, lastname, birthdate
         FROM employee WHERE empno  IN (                     2
            SELECT empno  FROM empprojact
            WHERE projno = prjnbr ) ;
```

Note: UDTFs differ from UDFs in how the return clause is specified. Instead of declaring a scalar value type to be returned, it declares columns for the temporary table that will be the function result, as shown in 1. In this example, the function returns a table, whose content will be provided by the SELECT function in 2.

UDTFs can be used in SELECT statements, as shown in Example 6-19.

Example 6-19 UDTF in a SELECT statement

```
SELECT * FROM TABLE(empbyprj('OP1010')) AS x;

SELECT a.col1, a.col2, a.col2, b.col1, b.col2, b.col3
   FROM tbl1 AS a JOIN TABLE(udtf01(parm1, parm2)) AS b ON a.keycol = b.keycol;
```

6.10.2 More complex SQL statement UDTFs

Example 6-20 shows how to use a UDTF to select a specific portion of a character large object (CLOB) column that is stored in a table and to redefine this portion as a VARCHAR in the generated table.

Example 6-20 UDTF that extracts a CLOB portion

```
CREATE OR REPLACE FUNCTION Resume_Pos ()
   RETURNS TABLE (name VARCHAR(128), current_position VARCHAR(130), empno CHAR(6))
   LANGUAGE SQL
   NO EXTERNAL ACTION
   F1: BEGIN ATOMIC
   RETURN
```

```
    SELECT SUBSTR(resume,15,POSITION('Personal Infor' IN  resume) -15),      1
    substr(resume,POSITION('present' IN  resume) +12 , 95), empno            2
    FROM emp_resume
    WHERE resume_format = 'ascii';                                           3
END;
```

Notes about Example 6-20 on page 177:

1 The NAME result column is populated. Inside a CLOB column, use POSITION to find the string 'Personal Infor' and start to read a predefined number of characters with function SUBSTR.

2 The CURRENT_POSITION result column is populated. Inside a CLOB column, use POSITION to find string 'present' and start to read a predefined number of characters with function SUBSTR.

3 Only rows with a specific resume format are selected.

Example 6-21 shows a simple usage of the UDTF that was created.

Example 6-21 Usage of a UDTF

```
SELECT  x.*, job FROM employee a INNER JOIN TABLE(Resume_Pos()) as x ON a.empno = x.empno
   WHERE job= 'ANALYST' ;
```

This UDTF is used to retrieve the employee full name, current position description, and number for those employees that are defined as an ANALYST in a joined table, as shown in Figure 6-9. The content of the CURRENT_POSITION column is retrieved by reading from a CLOB column and positioning correctly in its content.

NAME	CURRENT_POSITION		EMPNO	JOB
Delores M. Quintana	Advisory Systems Analyst	Producing documentation tools for engineering	000130	ANALYST
Heather A. Nicholls	Architect, OCR Development	Designing the architecture of OCR products.	000140	ANALYST

Figure 6-9 Data from the UDTF that extracts the CLOB portion

6.10.3 External action UDTF

So far, we described UDTFs that read and present data, but more function is possible. A UDTF can modify data and read and present data. This UDTF is described.

Assume that you need to track the users that access sensitive data. In your data model, you can define a “logging table” where access is registered. You can enforce the use of a function to access sensitive data, instead of exposing the original tables for direct access.

A simple UDTF that can perform this operation is presented in Example 6-22.

Important: As a prerequisite for this example UDTF, create a logging table:

```
CREATE OR REPLACE TABLE AUDIT_EMPLOYEE_ACCESS FOR SYSTEM NAME AUDT_EMP (
USR_ACS CHAR(10) CCSID 37 DEFAULT NULL , ACC_TIME TIMESTAMP DEFAULT NULL )
RCDFMT AUDT_EMP   ;

LABEL ON COLUMN  AUDIT_EMPLOYEE_ACCESS ( USR_ACS TEXT IS 'User accessing
EMPLOYEE table' , ACC_TIME TEXT IS 'time EMPLOYEE table was accessed' ) ;
```

Example 6-22 UDTF with INSERT and SELECT

```
CREATE OR REPLACE FUNCTION adv_deptemployees (i_deptno VARCHAR(3))
   RETURNS TABLE (emp_id CHAR(6), emp_name VARCHAR(35), educ_level SMALLINT, dept_name
   VARCHAR(36))
   LANGUAGE SQL
   DETERMINISTIC
   NOT FENCED
   EXTERNAL ACTION
   MODIFIES SQL DATA                                                              1
   DISALLOW PARALLEL
   CARDINALITY 10                                                                 2
   BEGIN
      DECLARE allcaps_deptnum VARCHAR(3);
      SET allcaps_deptnum=UPPER(i_deptno);                                        3
      INSERT INTO audit_employee_access VALUES(current user, current timestamp);  4
      RETURN
         SELECT empno, lastname CONCAT ', ' CONCAT firstnme, edlevel, deptname    5
         FROM employee e INNER JOIN dept d ON e.workdept=allcaps_deptnum;
   END;
```

Notes about Example 6-22 on page 179:

1 This UDTF can modify data (--> INSERT).

2 CARDINALITY 10 means to expect a result set of more or less 10 rows.

3 The input parameter is converted in uppercase to circumvent user error.

4 The logging table is inserted:

- The special register CURRENT USER retrieves information from the job attribute to record the current session user name.
- The special register CURRENT TIMESTAMP retrieves information from the system.

5 The actual selection is performed to feed the resulting table.

The previous example can be used with the statement in Example 6-23.

Example 6-23 Using a multiple action UDTF and checking the log table

```
SELECT * FROM  TABLE(adv_deptemployees('a00')) x;

select * from audit_employee_access;
```

6.10.4 UDTF for ranking

The SQL language is growing more powerful. You can take advantage of this new power in many ways. Example 6-24 shows how to use online analytical processing (OLAP) specifications to produce a numbered list of the top 10 employees by salary.

Example 6-24 UDTF that uses OLAP specifications

```
CREATE OR REPLACE FUNCTION salary_rank()
   RETURNS TABLE(position INTEGER, empno CHAR(6), firstnme CHAR(20), lastname CHAR(20),
   salary DECIMAL(13,2), rank INTEGER, dense_rank INTEGER)
   SPECIFIC SAL_RANK
   LANGUAGE SQL
   READS SQL DATA
   NOT FENCED
   DETERMINISTIC
   BEGIN
      RETURN
         SELECT
         ROW_NUMBER() OVER(ORDER BY salary DESC) as position  ,                1
         empno, firstnme, lastname, salary,
         RANK() OVER(ORDER BY salary DESC) as rank,                            2
         DENSE_RANK() OVER(ORDER BY salary DESC) as dense_rank                 3
       FROM employee
         ORDER BY salary DESC
         FETCH FIRST 10 ROWS ONLY;
   END;
```

Notes about Example 6-24:

1 The ROW_NUMBER is used to number the result list.

2 RANK provides the relative rank of each salary. Rows with the same salary are assigned the same rank. RANK returns a value for a row that is one more than the total number of rows that precede that row. Gaps occur in the numbering sequence whenever duplicates exist.

3 In contrast to RANK, DENSE_RANK prevents gaps in the numbering sequence. It returns a value for a row that is one more than the number of distinct row values that precede it.

For more information about OLAP specifications in SELECT statements, see the IBM Knowledge Center or DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

Example 6-25 shows this UDTF.

Example 6-25 UDTF that uses OLAP

```
SELECT * FROM TABLE (salary_rank()) R;
```

Figure 6-10 shows the result set that was obtained by referencing our new UDTF.

POSITION	EMPNO	FIRSTNME	LASTNAME	SALARY	RANK	DENSE_RANK
1	000010	CHRISTINE	HAAS	52750.00	1	1
2	000110	VINCENZO ...	LUCCHESSI	46500.00	2	2
3	200010	DIAN	HEMMINGER	46500.00	2	2
4	000020	MICHAEL	THOMPSON	41250.00	4	3
5	000050	JOHN	GEYER	40175.00	5	4
6	000030	SALLY	KWAN	38250.00	6	5
7	000070	EVA	PULASKI	36170.00	7	6
8	000060	IRVING	STERN	32250.00	8	7
9	000220	JENNIFER	LUTZ	29840.00	9	8
10	200220	REBA	JOHN	29840.00	9	8

Figure 6-10 UDTF that uses the OLAP result set

This simple SQL statement is powerful. The SQL programming language offers many advantages and useful capabilities.

6.11 Pipelined table functions

A pipelined function uses the PIPE SQL statement to return UDTF results, row by row. A new form of the RETURN statement indicates the end of file (EOF) condition.

A pipelined table function is a pure SQL alternative to an external UDTF.

Pipelined table functions offer these advantages:

- Ease of implementation

  Pipelined table functions are easy to implement. The alternative to a pipelined function is typically a complex query, which might include advanced joins, selective aggregation, or CASE logic statements. Many programmers choose to perform data conditioning inside table functions because they are more comfortable with the procedure constructs.

- Additional support

  Pipelined table functions allow the flexibility to programmatically create “virtual” tables with greater control than SELECT or CREATE VIEW can provide. The following examples show this capability:

  - Pass function parameters to provide behavior customization
  - Perform data validation, correction, and conversion
  - Log bad values
  - Return lines of text from a backup log file or other operating system file
  - Return multiple rows from a single input row
  - Reference multiple databases within a single statement
  - Provide customized join behavior
  - Perform outer joins to more than one table at one time
  - Perform outer joins to a list of values or a subquery

- Big data, analytics, and performance

  Pipelined table functions can provide more efficient processing of large amounts of data that is returned by an SQL table function. Rather than building a temporary table (either on disk or in memory) and then returning the entire table to the data processor, the data can be returned to the processor one row at a time. A temporary table is not necessary, and the expense of creating a temporary table can be avoided. This capability can help save memory and improve performance for an SQL table function.

  In certain cases, only a subset of the data from an SQL table function is needed. Pipelined table functions can be used to avoid the creation of data that is not required. Because only the rows that are required are produced, this capability can help improve performance for an SQL table function.

- Interoperability/portability

  This enhancement enables the portability of pipelined table functions from other platforms.

Note: The preferred practice of embracing “set at a time” processing (going to the query engine one time instead of many times) remains the best SQL programming practice. Use the power and flexibility of a pipelined function in cases where a traditional SQL UDTF is not a viable choice.

6.11.1 PIPE syntax

The PIPE statement, which is used within the SQL-routine body, returns one row from a table function. Its syntax is briefly described. For more information, see DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

Figure 6-11 shows the PIPE statement syntax.

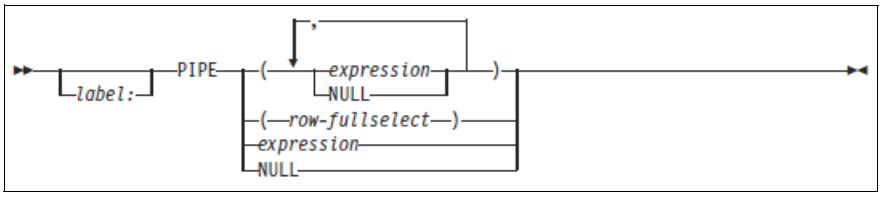

Figure 6-11 PIPE statement syntax

The following parts of the PIPE statement syntax are described:

- `label`

  The label specifies the label for the PIPE statement. It cannot be the same as the routine name or another label within the same scope.

- `(expression, ...)`

  The expression specifies that a row value is returned from the function. The number of expressions or NULL keywords in the list must match the number of columns in the function result. The data types of each expression must be assignable to the data type of the corresponding column that is defined for the function result. Use the common promotion rules. (See 6.6.2, “Parameter matching and promotion” on page 166.)

- `row-fullselect`

  The row-fullselect specifies a fullselect that returns a single row. The number of columns in the fullselect must match the number of columns in the function result. The data types of the result table columns of the fullselect must be assignable to the data types of the columns that are defined for the function result. Use the promotion rules. (See 6.6.2, “Parameter matching and promotion” on page 166). If the result of the row fullselect is no rows, a null value is returned for each column. An error is returned if more than one row is in the result.

- `expression`

  The expression specifies that a scalar value is returned from the function. The table function must return a single column, and the value of the expression must be assignable to that column.

- `NULL`

  NULL specifies that a null value is returned from the function. If more than one result column exists, a null value is returned for each column.

6.11.2 Pipelined function examples

Example 6-26 shows how the PIPE statement is used to provide a result set in the form of a virtual table whose results can be consumed one row at a time.

Example 6-26 Simple PIPELINED UDTF example

```
CREATE OR REPLACE FUNCTION transform()
   RETURNS TABLE ( employee_name CHAR(20), unique_nbr INT )                    1
   BEGIN
      DECLARE empname VARCHAR(15);
      DECLARE myrecnum INTEGER DEFAULT 1;                                      2
      DECLARE at_end INTEGER DEFAULT 0;
      DECLARE emp_cursor CURSOR FOR SELECT lastname FROM employee ORDER BY 1;  3
      DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'                            4
         SET at_end = 1;
   OPEN emp_cursor;
      myloop: LOOP                                                             5
      FETCH emp_cursor INTO empname;
      IF at_end = 1 THEN                                                       6
         LEAVE myloop;
      END IF;
      PIPE (empname, myrecnum);  -- return single row                          7
      SET myrecnum = myrecnum + 1;                                             8
   END LOOP;                                                                   9
      CLOSE emp_cursor ;                                                       10
      RETURN;
   END;
```

Notes about Example 6-26 on page 183:

1 This definition describes the table that is returned by this function.
2 The list of parameters is shown.
3 The CURSOR definition is matched to the SELECT statement.
4 A handler is declared to handle the LAST ROW condition.
5 A LOOP goes through the employee table until the last row.
6 When the LOOP is at the last row, the LOOP is closed.
7 PIPE is used to outfeed results one row at the time.
8 The myrecnum counter is incremented.
9 The LOOP is ended.
10 The cursor is closed.

You can check the virtual table content by using a simple SELECT statement, as shown in Example 6-27.

Example 6-27 Simple PIPELINED UDTF usage

```
SELECT * FROM TABLE(transform()) x;
```

Note: If more that one employee has the same lastname, the results from the previous example do not indicate the specific employee. You can use a different statement in your TRANSFORM function to identify the individual. Do not use this statement from the previous example:

```
SELECT lastname FROM employee
```

Use this statement instead:

```
SELECT RTRIM(lastname) CONCAT ', ' CONCAT SUBSTR(firstnme,1,1) CONCAT '. '
CONCAT (case when midinit = ' ' then ' ' else midinit CONCAT '. ' end) FROM
employee order by 1
```

Pipelined SQL table functions can contain multiple PIPE statements. Example 6-28 shows how to use two PIPE statements to provide a different result based on a previously specified condition. If multiple PIPE statements are included, each PIPE statement must return a value for every result column. A RETURN statement must be executed at the end of processing.

Example 6-28 Multiple PIPE statement UDF example

```
CREATE OR REPLACE FUNCTION projfunc (indate DATE)
   RETURNS TABLE (projno CHAR(6), actno SMALLINT, acstaff DECIMAL(5,2), acstdate DATE,
      acendate DATE, carryover CHAR(1))
   BEGIN
      FOR  projcur CURSOR FOR SELECT * FROM projact                              1
            WHERE acstdate<= indate   DO
      IF YEAR(acstdate) < YEAR(indate) THEN
      PIPE (projno , actno , acstaff , acstdate , acendate ,  'Y');              2
      ELSE
         PIPE (projno , actno , acstaff , acstdate , acendate , 'N');            3
      END IF;
      END FOR;
      RETURN;
   END;
```

Notes about Example 6-28:

1 This statement is the selection statement with a WHERE clause.
2 PIPE feeds out the result set with CARRYOVER = 'Y'.
3 PIPE feeds out the result set with CARRYOVER = 'N'.

The function can be invoked the same way as a non-pipelined function, as shown in Example 6-29.

Example 6-29 Non-pipelined function

```
SELECT * FROM TABLE(projfunc(:datehv)) X

SELECT projno, sum(actno) as tot_no, carryover, acstdate
   FROM TABLE(PROJFUNC(:datehv)) X
   GROUP BY projno,  acstdate, carryover ;
```

The body of a pipelined function is not limited to querying tables. It can call another program, get information from an API, query data from another system, combine the results, and use one or more PIPE statements to determine the row values to return as the result table.

6.12 Coding considerations: UDF preferred practices

Consider the following UDF preferred practices when you code UDFs:

- If you can choose between built-in SQL functions and UDFs, choose built-in SQL functions.
- UDFs generally bring the overhead of external calls and running in a separate system thread. The NOT FENCED clause can reduce the overhead by using a function call in the same thread.
- Consider the use of the DETERMINISTIC clause on the UDF definition to allow the SQL Query Engine on DB2 (SQE) optimizer to cache the results of previous function calls.
- If the function does not invoke high-level language (HLL) code, remember to specify NO EXTERNAL ACTION.
- SMP behavior:
  - DISALLOW PARALLEL is the default if one or more of the following clauses are specified: NOT DETERMINISTIC, EXTERNAL ACTION, or MODIFIES SQL DATA, or if this function is a pipelined table function.
  - In all other cases, ALLOW PARALLEL is the default and the database manager can consider parallelism for the function if SMP is enabled.

Often the same result can be obtained with different techniques. The use of different techniques can result in different response times (a performance consideration). For example, a commonly used join can be placed in a VIEW or implemented in a UDF, but the view will provide better performance for the following reasons:

- No function invocation costs are involved.
- The optimizer can rewrite the query that references the view more easily in the context of the statement where it was referenced.

A FENCED UDF executes 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 now allowed for UDFs. However, this choice might affect performance negatively.
- Fenced UDFs do not inherit program-adopted authority that was active. Authority comes from the UDF program or from the user who is running the SQL.

6.13 SQL control statements

DB2 for i provides a set of *programming constructs* (syntactic structures that are used to write procedural code) to help programmers to write SQL procedures. In DB2 for i, these programming constructs are called *control statements*. A control statement is one of the statements that can be placed in the routine body of an SQL UDF. Several control statements are listed:

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

These control statements are described in Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5, or in the DB2 for SQL reference.

6.14 Handling errors in SQL UDFs

Working with errors can be seen from two perspectives. A UDF must address error and warning conditions that occur inside its scope. A mechanism is necessary to communicate error and warning conditions to its caller. The other perspective involves the programs that invoke UDFs. These programs need to catch the warning or error conditions that might be fired from the UDF.

Handling errors in UDFs is similar to handling errors in stored procedures. For a general description of error handling, see 2.6, “Error handling” on page 29.

The differences in handling errors in UDFs are described. Assume that many tables represent dates as DEC(8), but our database manager does not have a built-in function that converts these dates to DATE format. We decided to expand the database management system (DBMS) functionality by creating a DEC2DATE function that receives a DEC(8) and returns a DATE. In this case, the designer of the function wants to receive an *SQLSTATE* 93001 when the function is provided with an invalid date. See Example 6-30.

Example 6-30 DEC2DATE UDF that signals a warning message for invalid dates

```
CREATE OR REPLACE FUNCTION dec2date (datedec DECIMAL(8, 0) )
   RETURNS DATE
   LANGUAGE SQL
   DETERMINISTIC
   CONTAINS SQL
   RETURNS NULL ON NULL INPUT
   NO EXTERNAL ACTION
   NOT FENCED
```

```
BEGIN
DECLARE newdate DATE ;
DECLARE InvalidDate CONDITION FOR '22007';
DECLARE EXIT HANDLER FOR InvalidDate
SIGNAL SQLSTATE '93001' SET MESSAGE_TEXT='DEC2DATE - Invalid date was provided';
SET newdate = DATE(SUBSTRING(DIGITS(datedec),1,4) || '-' ||
SUBSTRING(DIGITS(datedec),5,2) || '-' ||
SUBSTRING(DIGITS(datedec),7,2));
RETURN newdate;
END;
```

Note: The SIGNAL statement signals the error or warning condition explicitly. For a description of the SIGNAL statement, see 2.6.4, “SIGNAL and RESIGNAL” on page 35.

When you call this function with an incorrect value, you receive a customized error message, as shown in Example 6-31.

Example 6-31 Customized error message

```
SELECT dec2date(20113131) FROM sysibm.sysdummy1;

         SQL State: 93001
         Vendor Code: -438
         Message: [SQL0438] DEC2DATE - Invalid date was provided
```

Many ways are possible to resolve this date formatting issue. Certain methods might be more efficient than other methods. We do not specifically encourage you to use a function. In the “Using a date conversion table to convert to dates” section in Appendix A, “Date and time functionality” of the *IBM DB2 Web Query for i Version 2.1 Implementation Guide*, SG24-8063, you can see an example of how to use a join into a conversion table to solve this business problem.

7

Development and deployment

This chapter explains the tools that are available to develop and deploy the Structured Query Language (SQL) routines that are described in this publication.

This chapter includes the following topics:

- Tools for developing SQL routines and triggers
- Debug SQL routines and triggers
- Reverse engineering of SQL routines and triggers
- Ownership and authorities of SQL routines and triggers
- Deployment of SQL routines and triggers

7.1 Tools for developing SQL routines and triggers

Many tools are available from various vendors to help you develop SQL routines and triggers. The following tools that are provided by IBM are described:

- System i Navigator
- IBM Data Studio
- IBM i Access Client Solutions
- DB2 Express-C

7.1.1 System i Navigator

System i Navigator provides a graphical interface so that you can perform typical database development tasks. A pop-up menu is provided for each database object type so that you can create objects, alter existing objects, or drop existing objects.

Figure 7-1 contains a sample of the System i Navigator expanded database view.

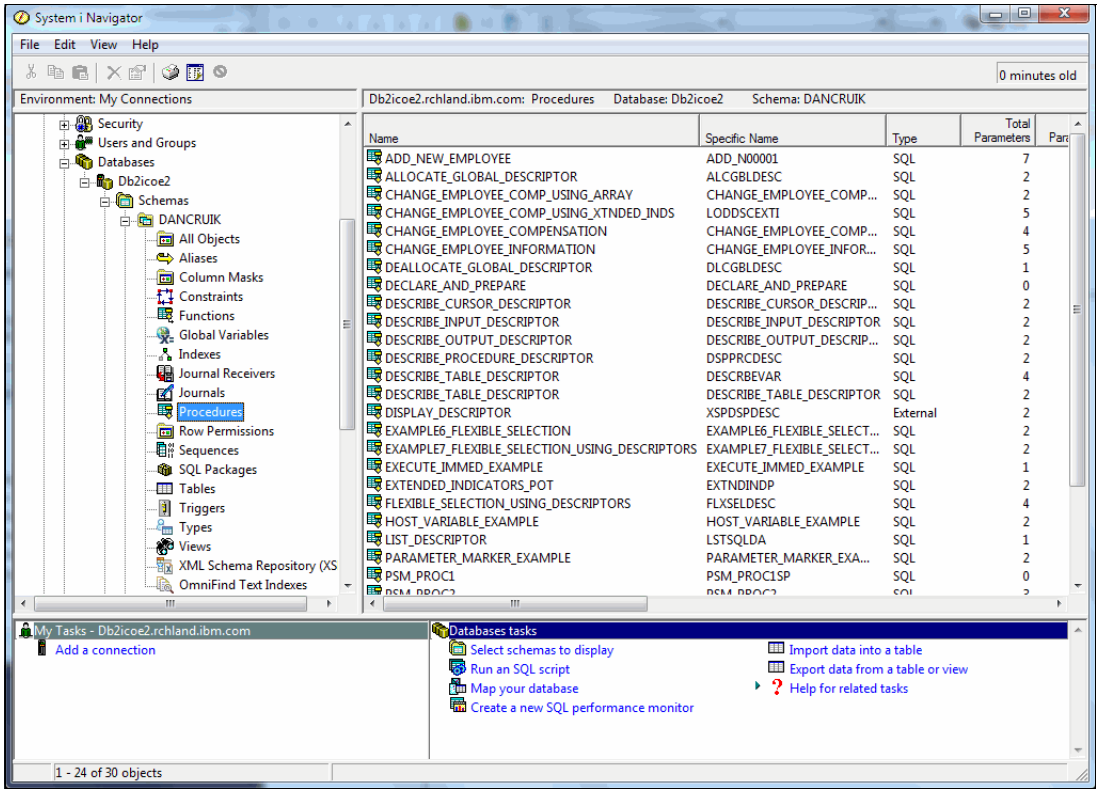

Figure 7-1 System i Navigator database view

You can use the Run SQL Scripts tool, which is shown in Figure 7-2, for easy testing and debugging of SQL statements or procedures.

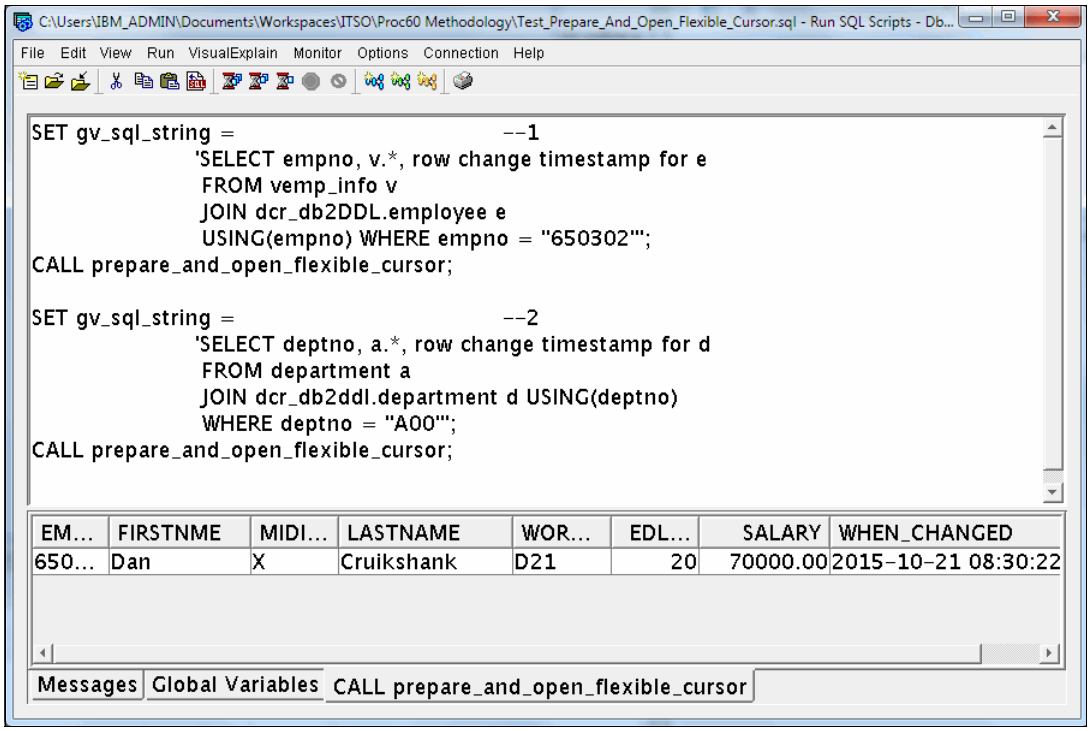

Figure 7-2 System i Navigator Run SQL Scripts tool

You can access the IBM DB2 for i Visual Explain tool through Run SQL Scripts. The Visual Explain tool supports three options: Explain only, Run and Explain, or Explain While Running. You access Visual Explain through the menu bar, as shown in Figure 7-3, or by selecting the corresponding icon from the toolbar.

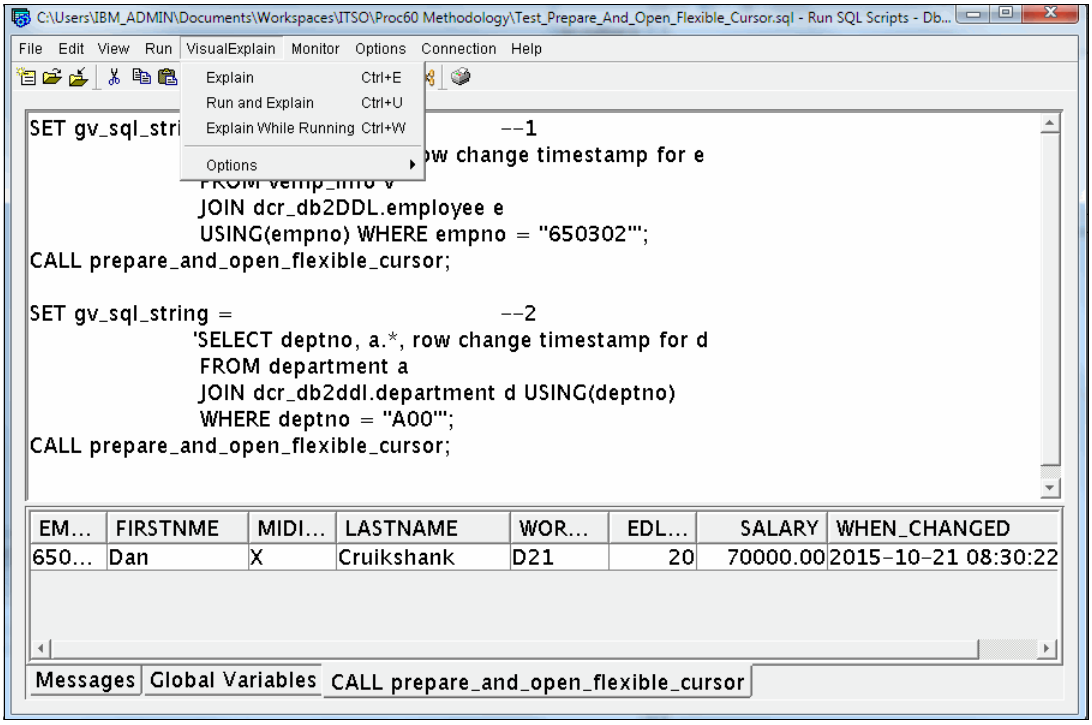

Figure 7-3 Run SQL Scripts: Accessing the Visual Explain tool

Figure 7-4 show an example of Visual Explain.

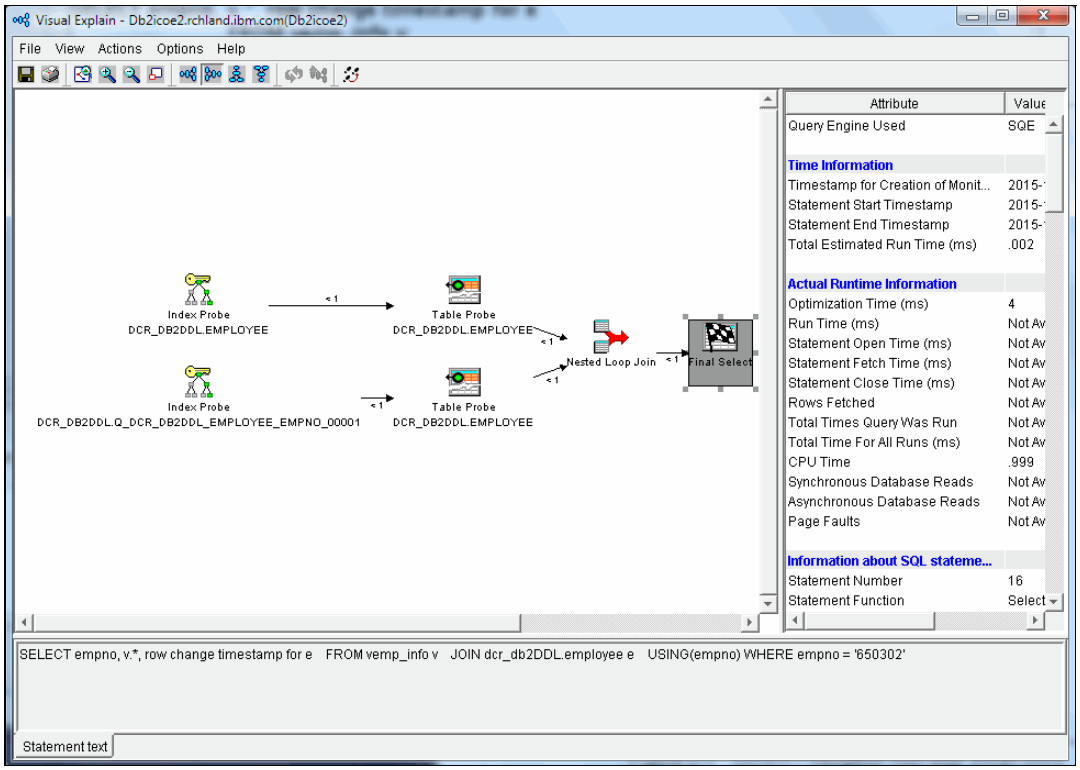

Figure 7-4 Visual Explain example

7.1.2 IBM Data Studio

The Data Studio client is built on Eclipse technology. It provides an integrated environment for database. Data Studio helps developers develop, debug, and deploy database routines and triggers.

Many developers who use IBM Rational® Lifecycle Package tools find that Data Studio fits easily into their integrated development environment (IDE). Data Studio takes advantage of the following development aids:

- A starter set of customizable SQL routine and trigger templates
- Snippets for the quick insertion of parameterized common code blocks
- Parameter markers in SQL scripts
- Automatic prompting when you run procedures and functions from Data Studio

Figure 7-5 shows the Data Studio Data perspective.

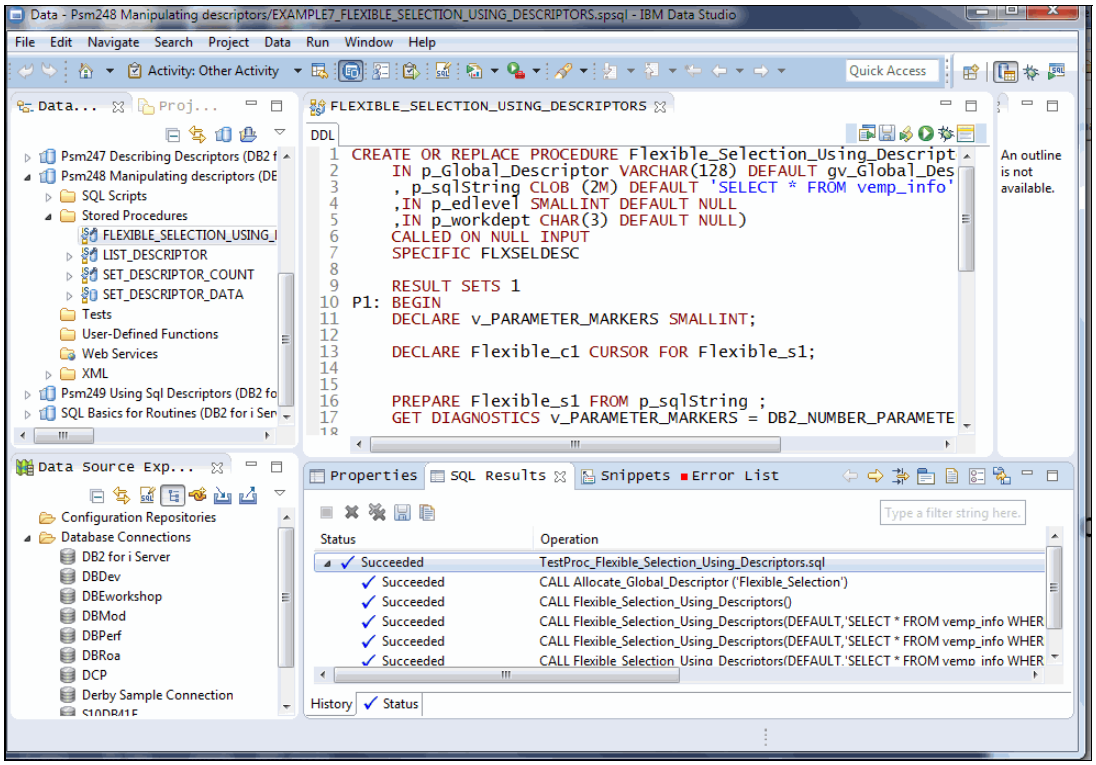

Figure 7-5 Data Studio Data perspective

The Data Studio Data perspective contains four primary views, which provide tools for the development of database artifacts:

- Data Project Explorer
- Data Source Explorer to manage database connections
- SQL Results to review the output from executing SQL scripts or routines
- The editor to create and maintain SQL scripts, routines, and triggers

For more information about IBM Data Studio, see this website:

https://ibm.biz/Bd4qbU

Accessing System i Navigator Run SQL Scripts from within Data Studio

The IBM Data Studio product provides a Visual Explain tool for IBM DB2 for Linux, UNIX, and Windows (LUW) and IBM DB2 for z/OS. It does not provide a Visual Explain tool for DB2 for i. However, Data Studio provides a method to launch the System i Navigator Run SQL Scripts tool from within Data Studio.

Use the following steps to enable the capability to launch the Run SQL Scripts tool from within Data Studio:

1. In Data Studio, open the **Preferences** panel from the Window drop-down menu, as highlighted in Figure 7-6.

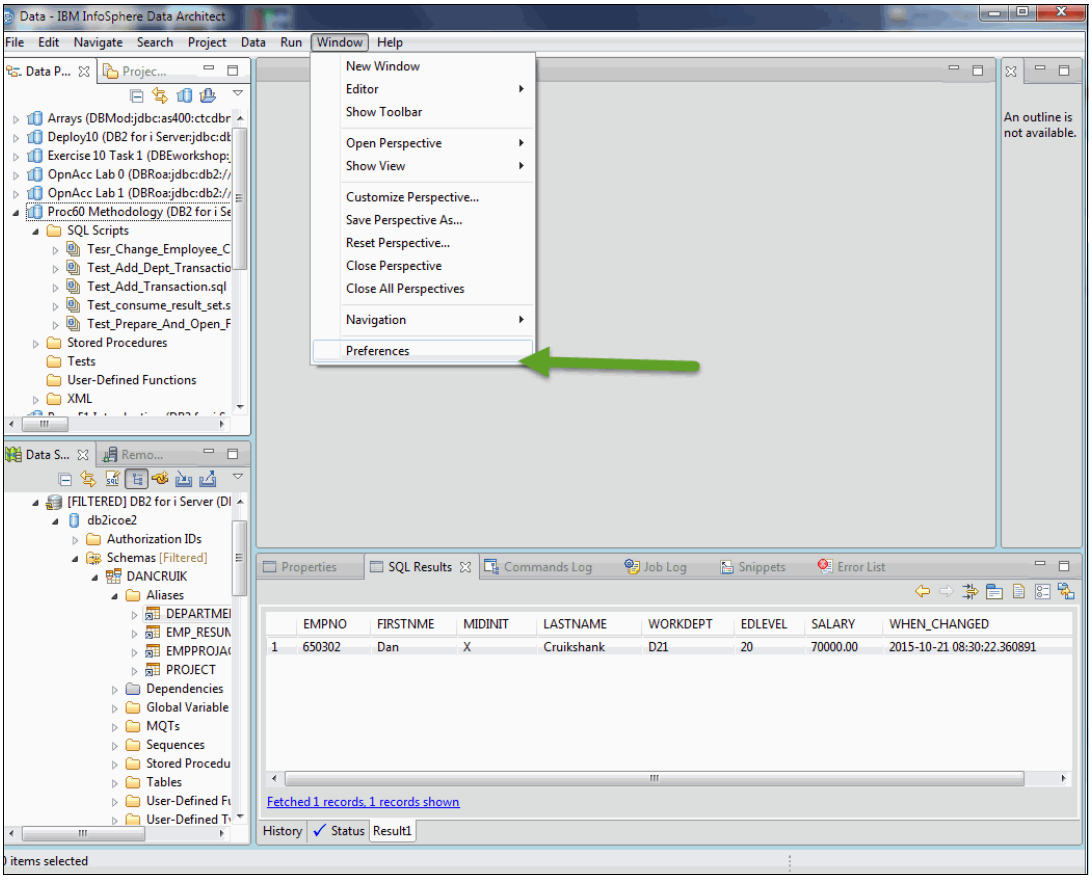

Figure 7-6 Open Preferences

2. A window is shown with all of the preferences. Expand the **General** options in this window and click **File Associations**, as shown in Figure 7-7.

   In the File types section, click **.sql** and click **Add** to display the Associated editors list.

   In the Associated editors list, click **SQL File**. Click **Add** (A in Figure 7-8 on page 197) and click **OK**.

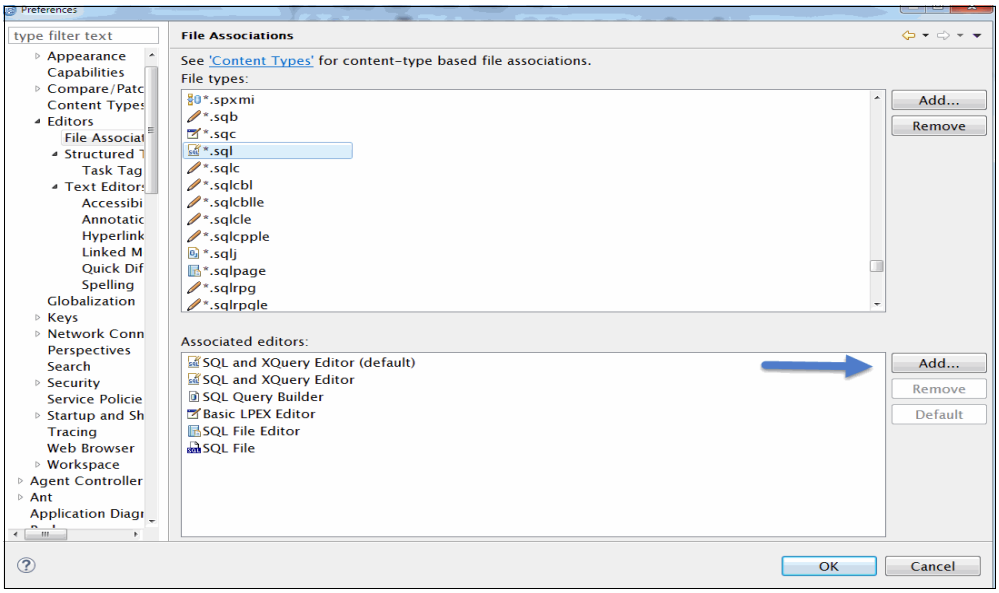

Figure 7-7 Associated editors

3. The Editor Selection window opens. Click **External programs**. Scroll down the list until you see SQL File. Click **SQL File** (B). Click **OK** (C) to associate this editor with file type `.sql`. Figure 7-8 shows these actions.

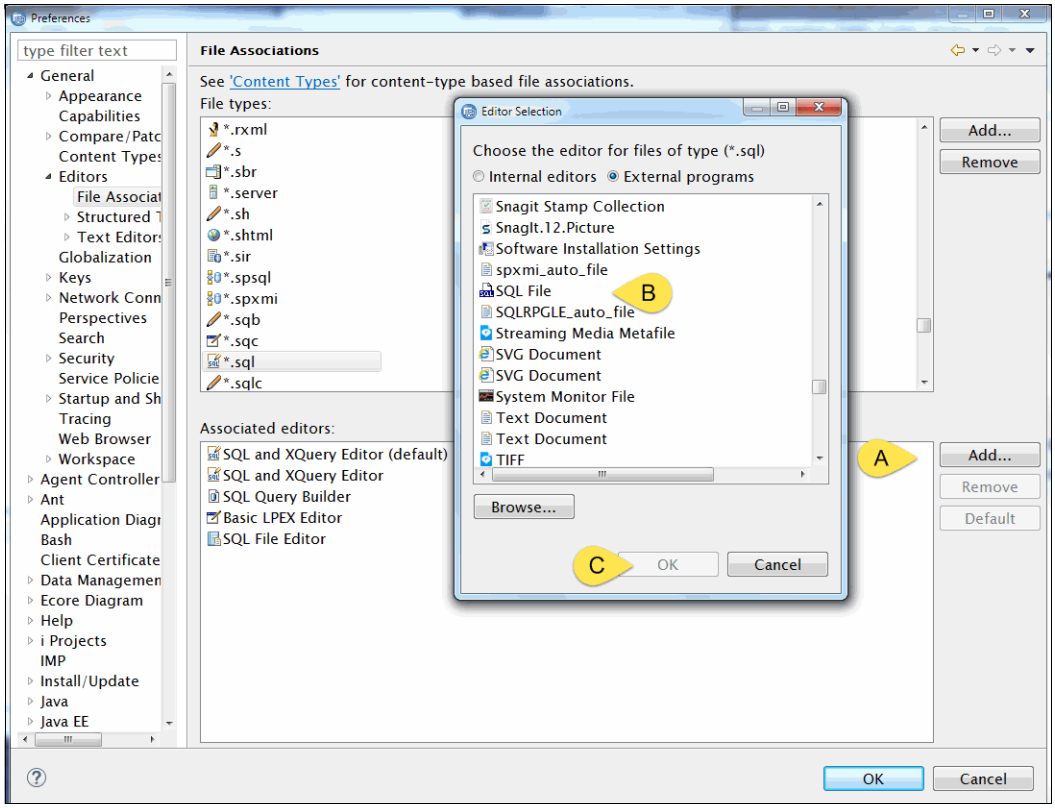

Figure 7-8 Add the Run SQL Scripts editor

4. After you create the association, the Run SQL Scripts tool appears as an alternative editor for SQL scripts. Right-click an existing SQL script in Data Studio, choose the **Open With** option from the pop-up menu, and then choose **SQL File**. Figure 7-9 shows these steps.

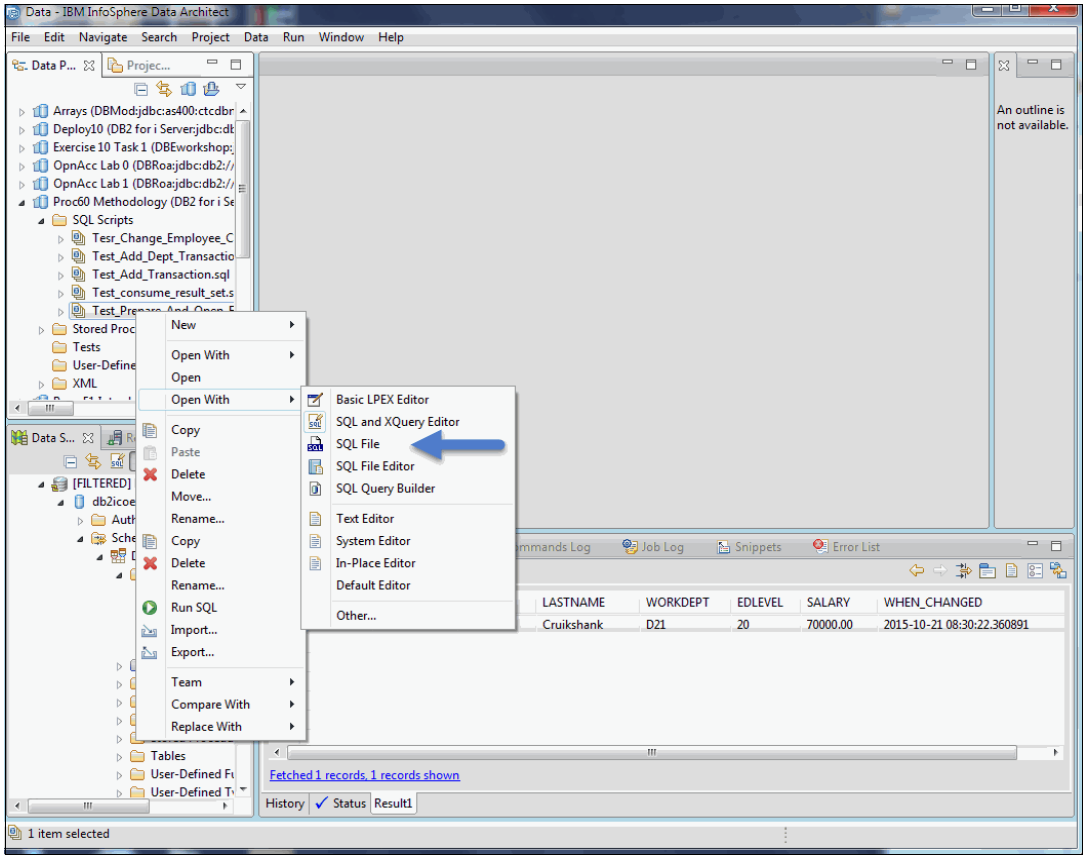

Figure 7-9 Launching the Run SQL Scripts tool from within Data Studio

Figure 7-10 shows the Run SQL Scripts tool after you launch it from within Data Studio.

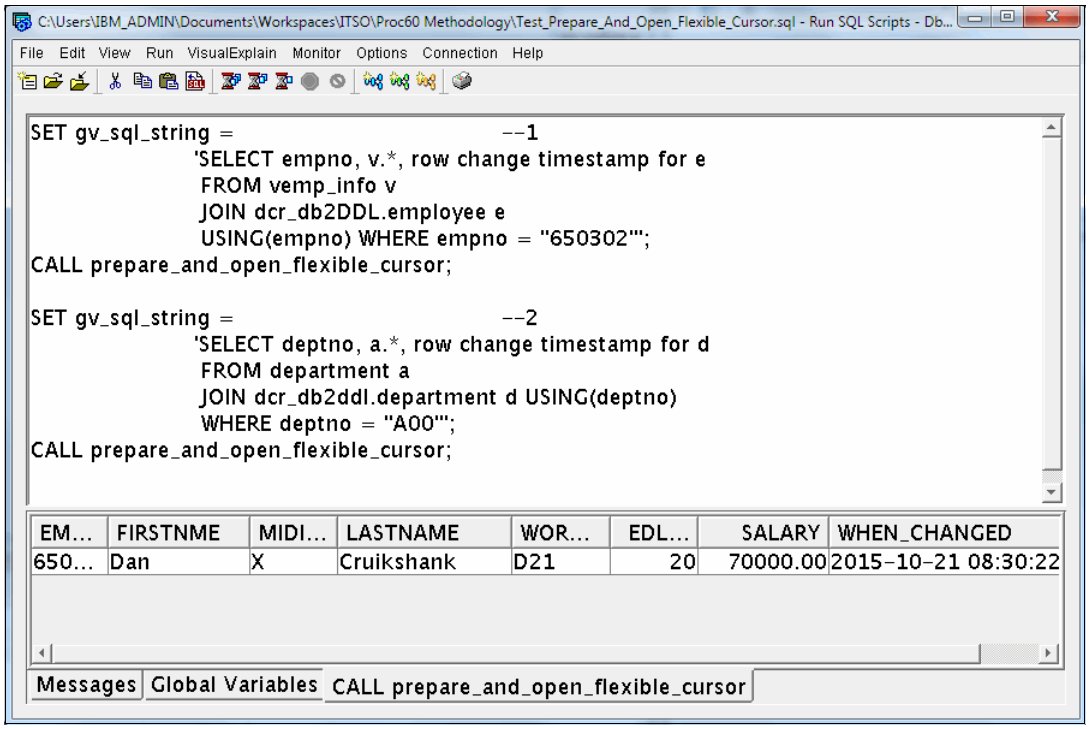

Figure 7-10 Run SQL Scripts tool that was launched from within Data Studio

7.1.3 IBM i Access Client Solutions

IBM i Access Client Solutions (ACS) provides a platform-independent alternative to IBM i Access for Windows that is based on Java. It consolidates many common tasks for managing your IBM i environment. You can run it on Windows, Mac, Linux, and other platforms that support a full Java virtual machine (JVM). IBM i Access Client Solutions was enhanced to include database functionality as part of IBM i 7.1 Technology Refresh 11 and IBM i 7.2 Technology Refresh 3.

Figure 7-11 shows the Welcome window for IBM i Access Client Solutions.

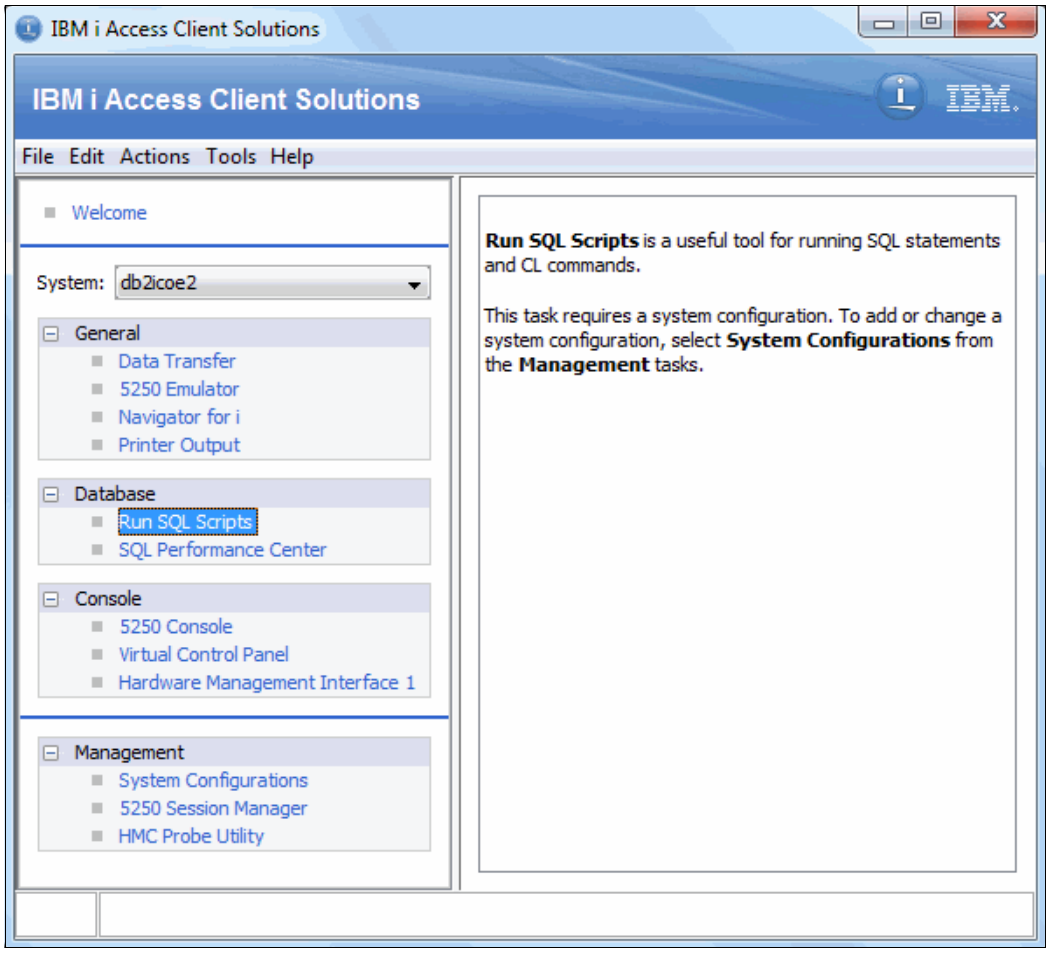

Figure 7-11 IBM i Access Client Solutions Welcome window

The Database options include the SQL Performance Center and a new implementation of Run SQL Scripts. The SQL Performance Center provides interfaces for working with Performance Monitors, Plan Cache Snapshots, and Plan Cache Event Monitors.

Figure 7-12 shows the IBM i Access Client Solutions version of Run SQL Scripts.

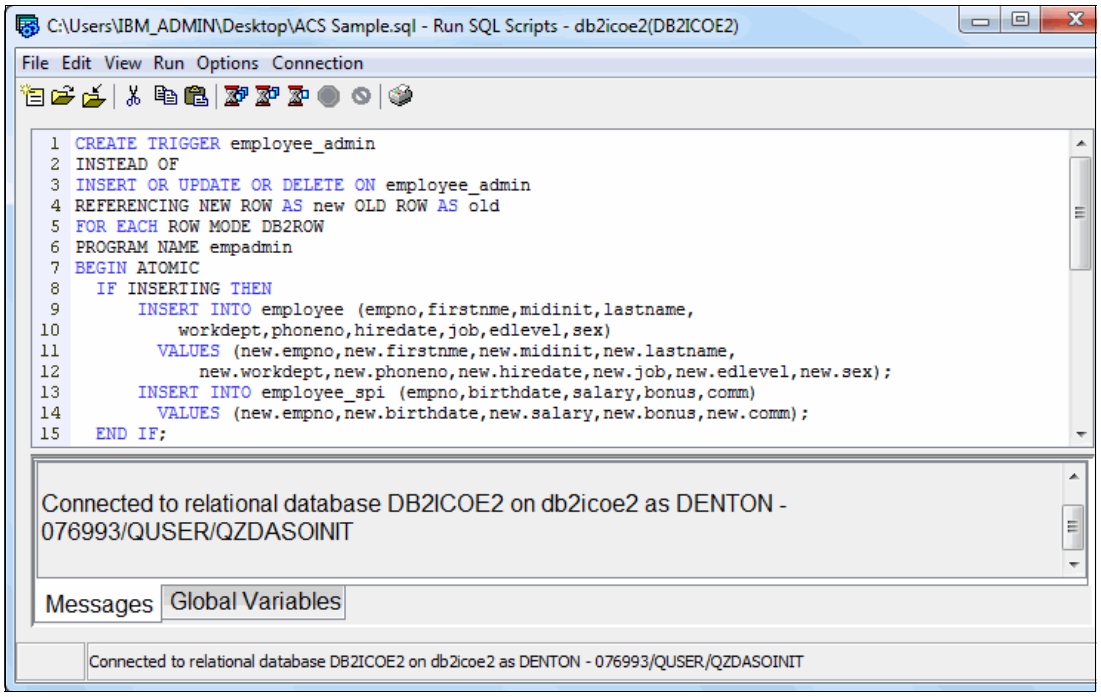

Figure 7-12 IBM i Access Client Solutions Run SQL Scripts

The IBM i Access Client Solutions version of Run SQL Scripts is patterned after the support in IBM System i Navigator. This version provides many of the same functions. It also provides the following benefits:

- Faster launch
- Syntax formatting (SQL keywords are shown in blue)
- Line numbers
- Status bar
- Reconnect and disconnect functions

Development intends to offer the following functionality in later versions:

- Visual Explain
- Integrated SQL Performance Monitor (cross-launch)
- Control language (CL) prompting

For additional information about IBM i Access Client Solutions, see this website:

https://ibm.biz/Bd4q8R

7.1.4 Comparison

Table 7-1 provides a comparison of several major features of the tools that were described.

Table 7-1 Tool comparison

Feature	IBM Data Studio	System i Navigator	IBM i Access Client Solutions
Routine/trigger development	Yes	Yes	Yes
Run SQL Scripts	Yes	Yes	Yes
Visual Explain	No[a]	Yes	Planned
Debugger	Yes	Yes	Yes
Deployment	Yes	Yes	Yes
Source change management	Yes[b]	No	No
Multiple database support	Yes	No	No

a. You can launch the System i Navigator Run SQL Scripts tool and then use System i Navigator Visual Explain.
b. IBM Data Studio integrates with any Eclipse source change management (SCM) product.

7.1.5 DB2 Express-C

DB2 Express-C is a no-charge edition of DB2 for Linux, UNIX, and Windows. It is easy to install. It is available for download, deployment, and redistribution at *no charge*.

Although it is installed on Linux, UNIX, and Windows servers, it can be used for the offline development and testing of applications that are intended for deployment on much larger enterprise database servers, such as DB2 for i or DB2 for z.

You can download DB2 Express-C from the following website:

`https://ibm.biz/Bd4q8q`

The download includes IBM Data Studio, which is an integrated tools environment that helps you manage database administration and development with ease.

Figure 7-13 shows an example of IBM Data Studio that is connected to DB2 for Linux, UNIX, and Windows.

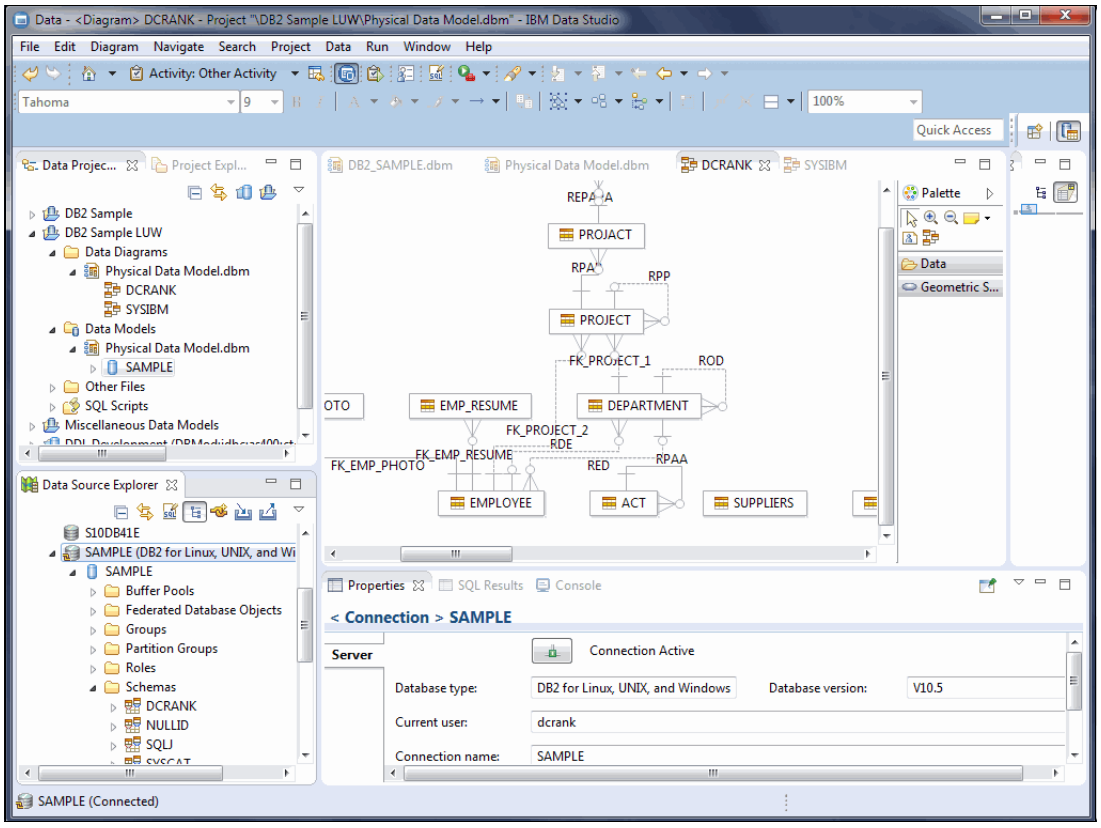

Figure 7-13 IBM Data Studio Data perspective for DB2 for Linux, UNIX, and Windows

IBM Data Studio supports multiple connections so that a database engineer can switch from unit testing that uses DB2 for Linux, UNIX, and Windows to deployment on DB2 for i quickly.

7.2 Debug SQL routines and triggers

When you develop any kind of software, it is important to use a debugging tool. Use debugging to detect, diagnose, and eliminate runtime errors in a program.

The following tools provide graphical debugging of SQL routines and triggers:

- IBM Data Studio
- IBM Run SQL Scripts

7.2.1 Debug SQL routines and triggers by using IBM Data Studio

The graphical debugger that comes with IBM Data Studio simplifies debugging SQL routines and triggers. To get an overview of debugging an SQL procedure by using the IBM Data Studio debugger, see *IBM Data Studio debugger and IBM DB2 for i*, by Kent Milligan, at the following website:

`https://ibm.biz/Bd4q8V`

The following section refers to content in *IBM Data Studio debugger and IBM DB2 for i.*

Enabling IBM Data Studio debugger

To enable Data Studio debugger, your SQL routine must be created with the ALLOW DEBUG MODE special attributes to make the procedure eligible to debug. You enable the debug mode by selecting **Enable Debugging** on the Deploy Routines interface.

When you use Data Studio to create a routine, click **Next** (1) to open the Routine Options window. Click **Enable debugging** (2) and select **Finish** (3) to deploy the return. These steps are shown in Figure 7-14.

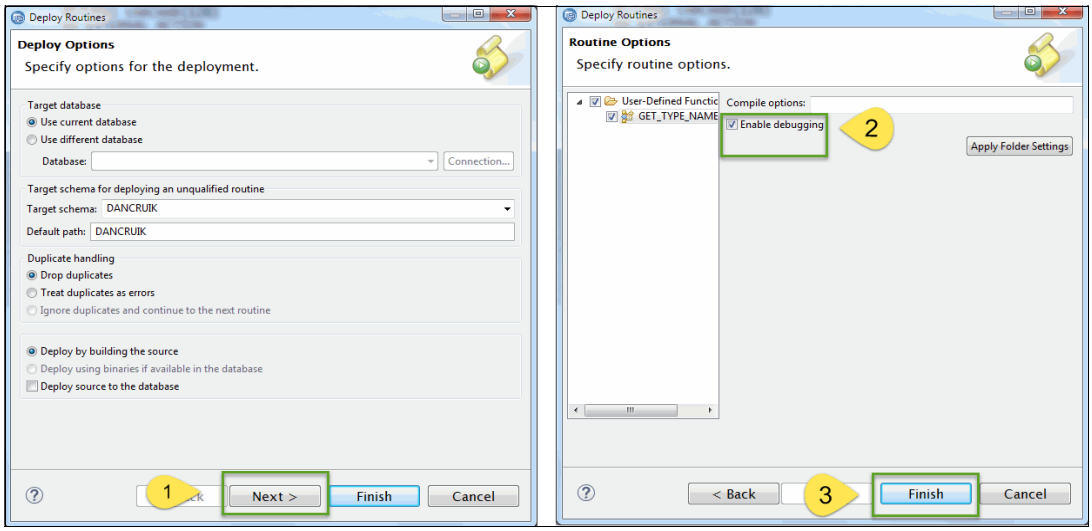

Figure 7-14 Enabling IBM Data Studio debugger

Also, you can add these attributes to the routine by specifying the ALLOW DEBUG MODE attribute or by using the CURRENT DEBUG MODE special register.

Debug an SQL routine

Locate the routine that you want to debug by using the Data Source Explorer (database connections) view, as shown in Figure 7-15.

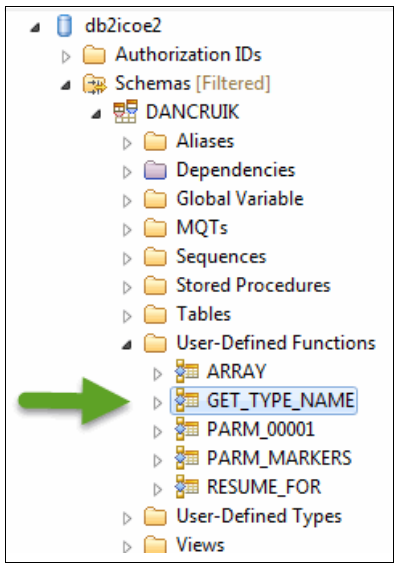

Figure 7-15 Data Source Explorer view

Run Settings option

Before you debug your SQL routine or trigger, you might need to prepare the environment by executing a SET PATH statement or populating a global variable. In addition, you might want to display the results of debugging your SQL routine or trigger by running an SQL query. You can display the results of debugging your SQL routine or trigger by right-clicking the SQL routine or trigger and selecting **Run Settings**, as shown in Figure 7-16.

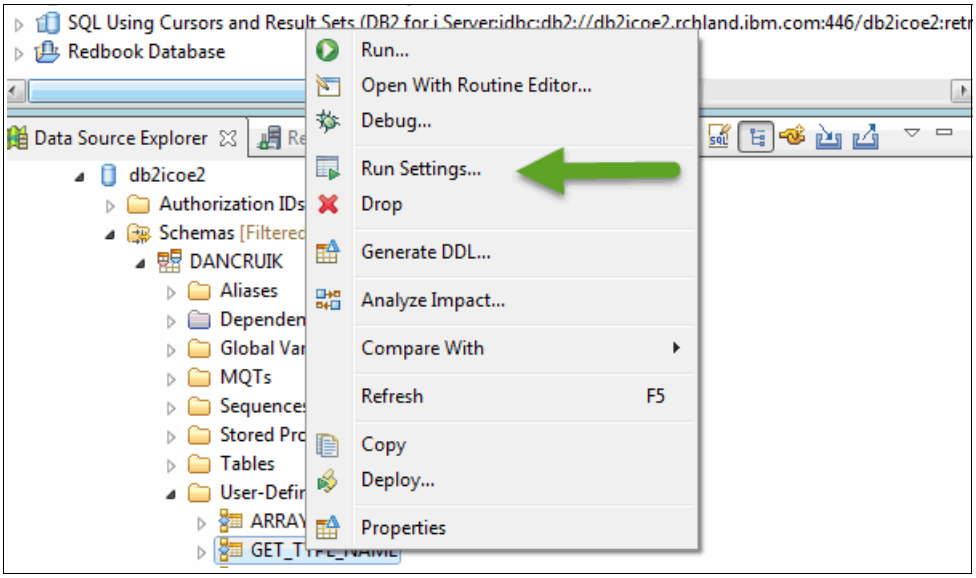

Figure 7-16 Run Settings option

Use the Before Run tab to enter SQL statements that need to execute before you debug your routine or trigger. Use the After Run tab to enter SQL statements that need to execute after you debug your routine or trigger. Figure 7-17 shows both options.

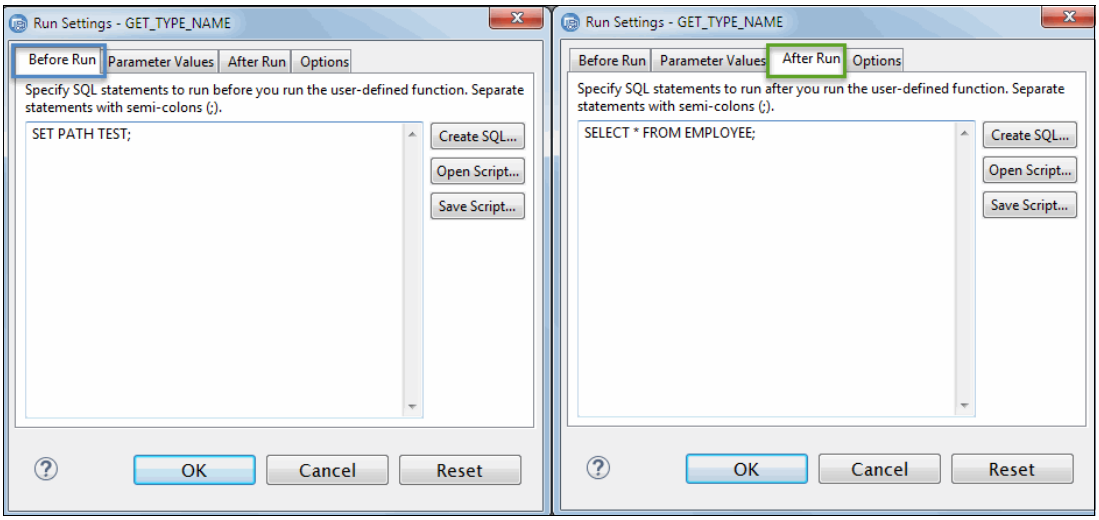

Figure 7-17 Before Run and After Run settings

Debug

When you are ready to debug your routine or trigger, right-click it and choose **Debug** (1 in Figure 7-18). You are prompted to enter values for any input parameters. Enter an appropriate value (2) and then click **Debug** (3).

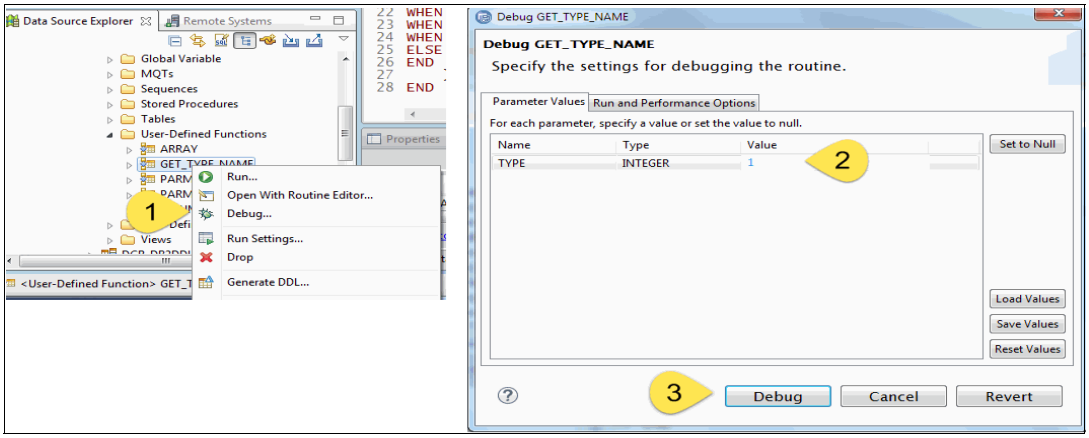

Figure 7-18 The Debug prompt

The Debug perspective is presented after you click Debug, as shown in Figure 7-19. You can now step through the routine or trigger and set breakpoints. The top line in the Debug view is where you can control the execution of the routine or trigger. Clicking the green arrow with the yellow bar icon causes the stored procedure to resume execution after a breakpoint is reached. The icons with yellow arrows provide various step executions (for example, Step Into and Step Over).

The Variables view in the upper-right corner (on the Variables tab) in Figure 7-19 is where you can access the value of the procedure's variables. Initially, the Variables view contains only two variables, *SQLCODE* and *SQLSTATE*, which are initialized to 0 by DB2. While the execution of the routine or trigger progresses, the Data Studio debugger automatically adds variables to this view.

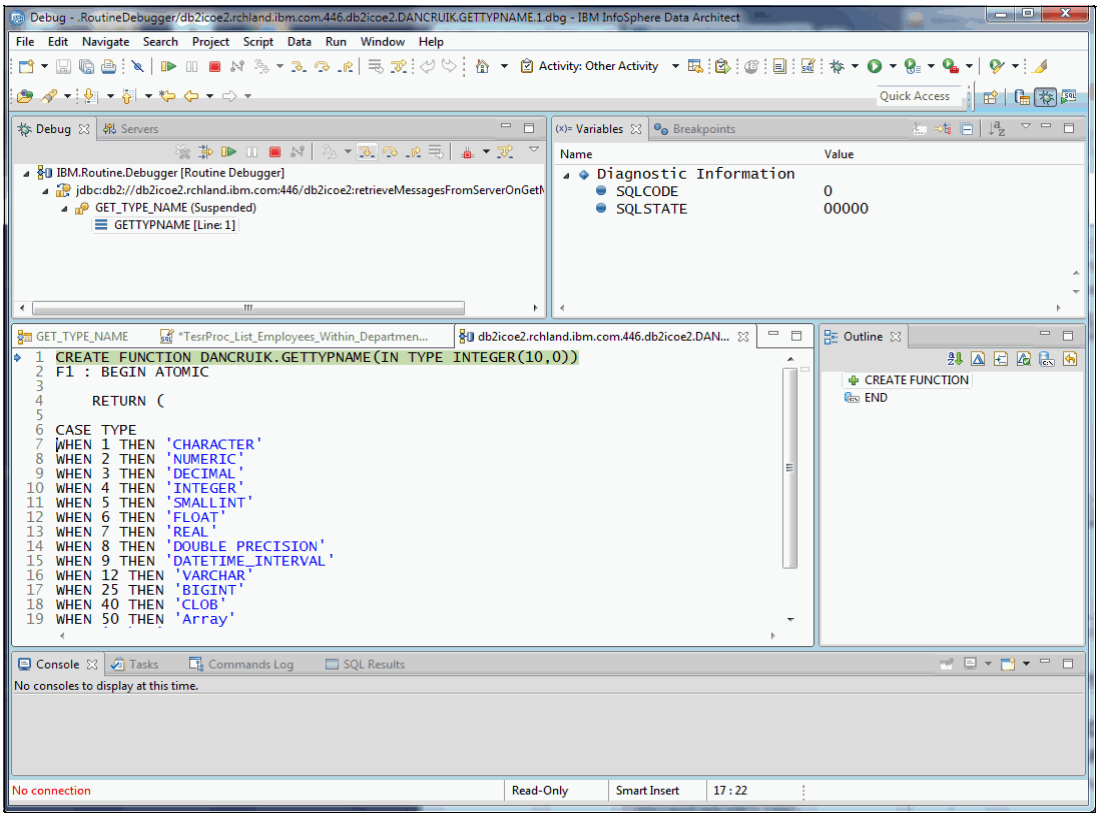

Figure 7-19 Debug perspective

Considerations

When you debug SQL functions, you must specify the NOT FENCED and DISALLOW PARALLEL attributes. Otherwise, DISALLOW DEBUG MODE is used.

7.2.2 Debug by using IBM Run SQL Scripts

The IBM i Run SQL Scripts tool provides many debug options to help you diagnose and resolve runtime errors in SQL routines and triggers. The following list shows the available debug tools:

- IBM i graphical debugger
- Debug messages within the job log
- SQL Performance Monitor for tracing a procedure

IBM i graphical debugger

Debugging SQL routines or triggers on IBM i is simplified with the SQL *SOURCE Debug View. To use this capability, the DBGVIEW option must be included in the routine or trigger source.

Example 7-1 shows a procedure that contains the code for setting the DBGVIEW option.

Example 7-1 Preparing a procedure for graphical debugging

```
CREATE OR REPLACE PROCEDURE List_Employees_Within_Department (
   p_workdept CHAR(3))
   RESULT SETS 1
   LANGUAGE SQL
   SPECIFIC lstempidpt                                              1
-- PROGRAM TYPE SUB                                                 2
   SET OPTION DBGVIEW = *SOURCE                                     3

P1: BEGIN
   -- Declare cursor
   DECLARE lstempidpt_c1 CURSOR WITH RETURN TO CLIENT FOR
   SELECT empno, FIRSTNME, MIDINIT, LASTNAME, workdept
   FROM vemp
   WHERE WORKDEPT = p_WORKDEPT
   ORDER BY LASTNAME;

   -- Cursor left open for client application
   OPEN lstempidpt_c1;
END P1
```

Notes about Example 7-1:

1 The SPECIFIC option is used to provide a 10-character system name for the routine. The IBM i graphical debugger is limited to system names only.

2 By default, SQL procedures are deployed as C program object types. The PROGRAM TYPE SUB option is used to deploy a C service program object type. It is important to know the program object type when you add a routine or trigger to the IBM i graphical debugger.

3 Use the SET OPTION DBGVIEW = *SOURCE to debug the procedure at the SQL statement level. The default option is *NONE, which provides no benefit when you debug.

To use the IBM i graphical debugger from Run SQL Scripts, click **Debugger** from the Run menu drop-down list (1 on Figure 7-20) or press Ctrl+D. This action launches the Start Debug window. Enter the system program (or service program) name (2) for the procedure. Click **Recent** to display a list of previously debugged programs or service programs. Click **OK** (3) after you enter the program information.

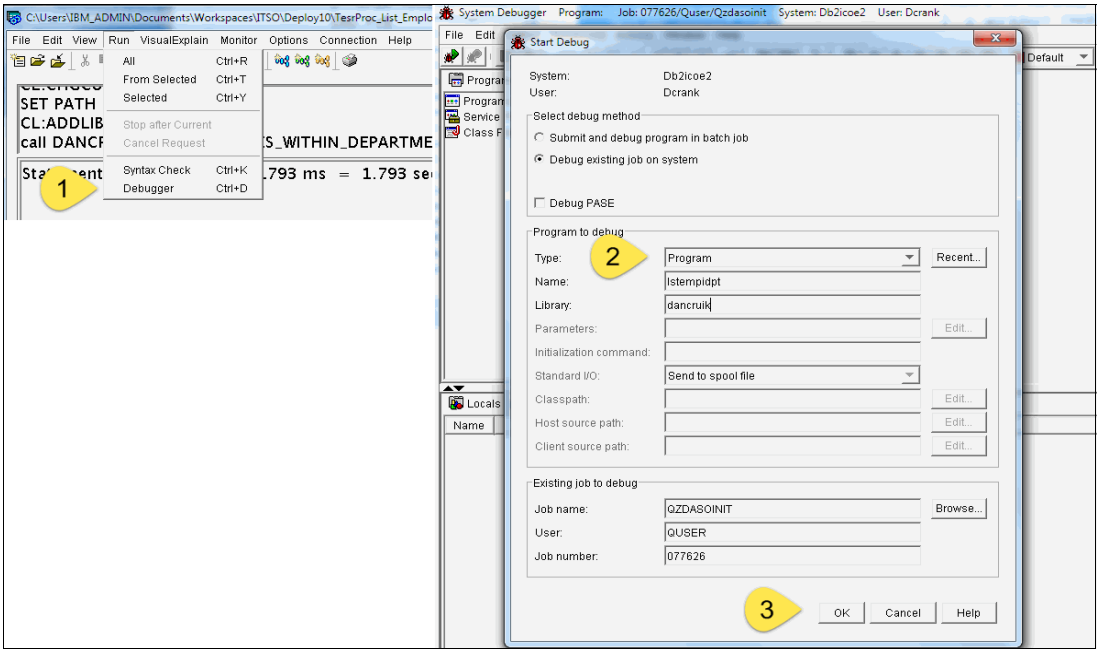

Figure 7-20 Start the debugger from IBM i Run SQL Scripts

Figure 7-21 contains the image of the IBM i Run SQL Scripts System Debugger after you click OK on the Start Debug panel.

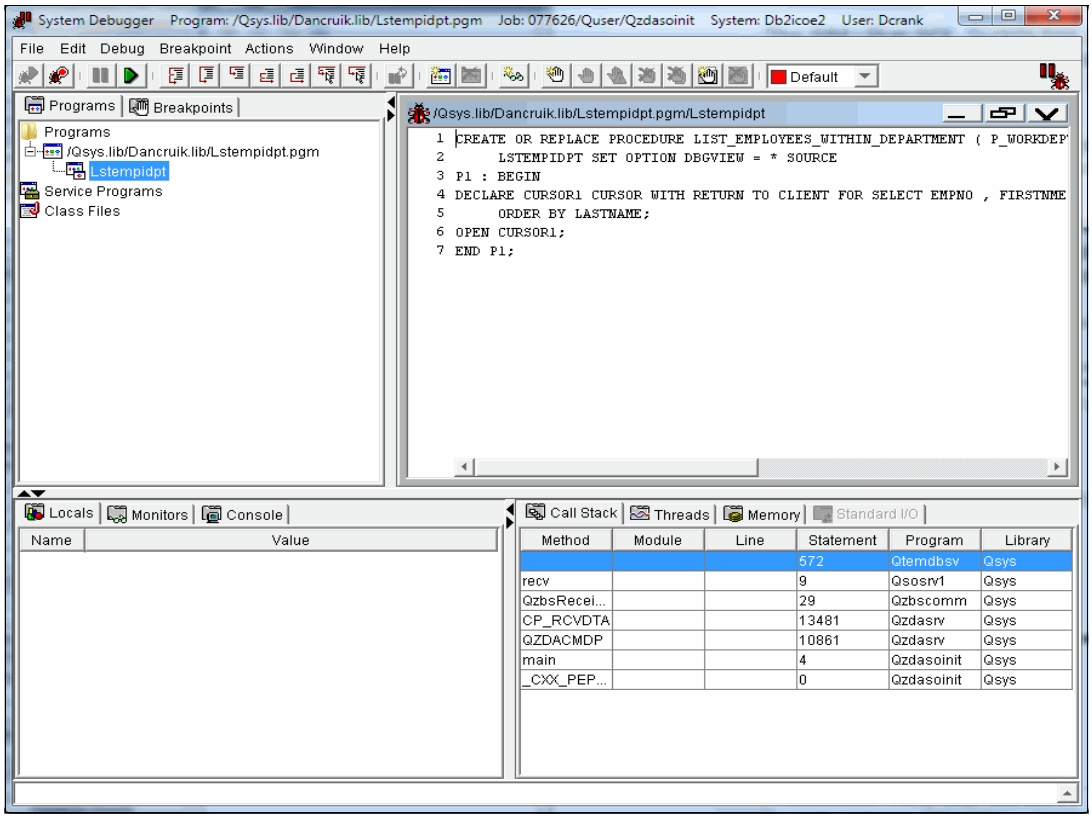

Figure 7-21 IBM i Run SQL Scripts System Debugger

For the latest technical information about the IBM i Run SQL Scripts graphical debugger, see the following links:

- https://ibm.biz/Bd4qhL
- https://ibm.biz/Bd4qh3

Debug messages within the job log

Every Run SQL Scripts session has an associated job log that contains informational and exception messages about the SQL statements that are processed. In addition, after the job log is enabled, additional information about the execution of SQL statements is also logged.

Although the IBM i graphical debugger is the best tool to diagnose runtime issues, sometimes the information in the job log can provide enough information to resolve a problem or provide information for setting breakpoints in the procedure that you need to debug.

To enable debug messages, click **Include Debug Messages in Job Log** from the Options drop-down menu, as shown in Figure 7-22.

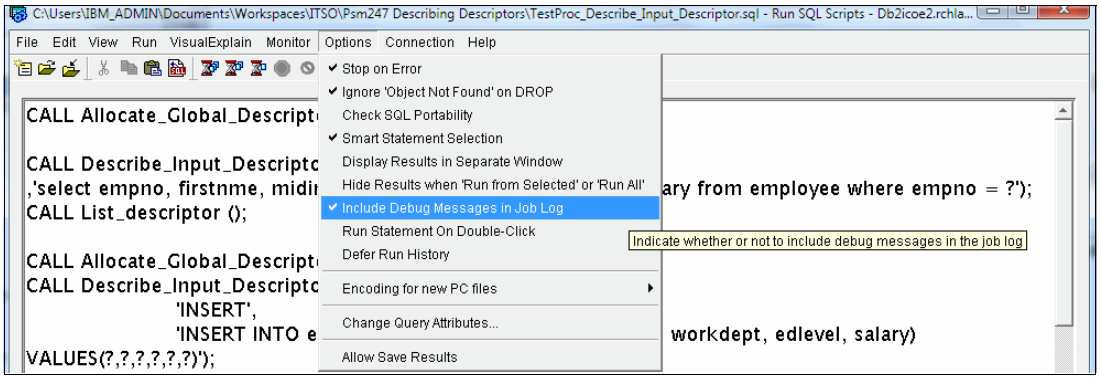

Figure 7-22 Enabling debug messages

Figure 7-23 shows the job log for the previous session with debug messages.

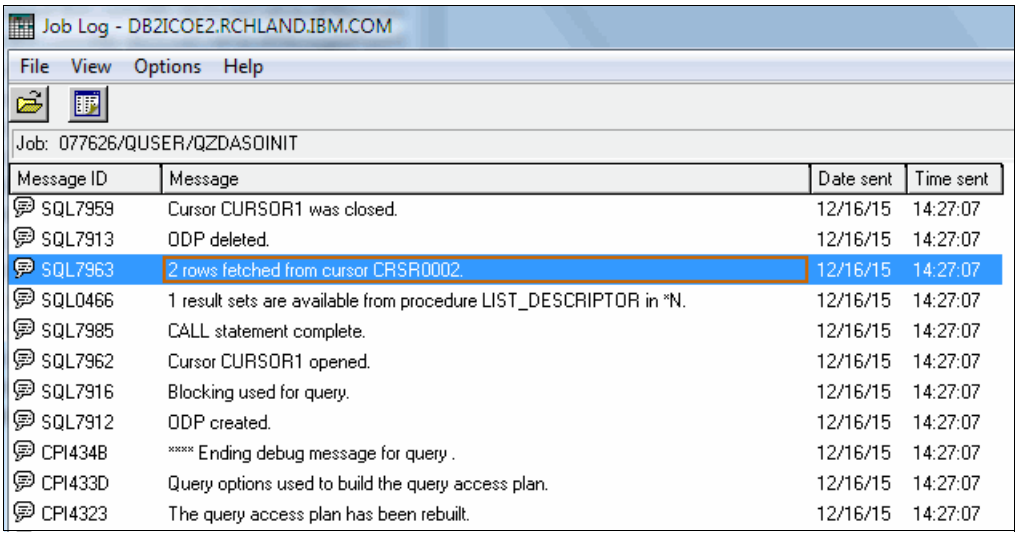

Figure 7-23 Job log with debug messages

SQL Performance Monitor for tracing a procedure

Sometimes, a procedure contains many SQL statements. Stepping through this procedure, and the possible nested procedures or functions that it calls, can be time-consuming, especially if the problem is not error-related. In this case, it might be more beneficial to capture the SQL statements by using an SQL monitor.

To start a trace, click **Start SQL Performance Monitor** from the Monitor menu options, as shown in Figure 7-24.

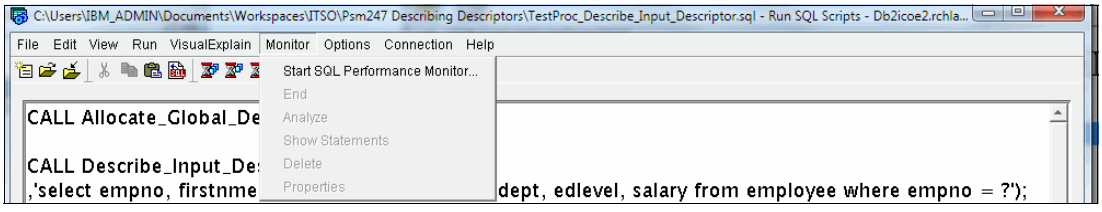

Figure 7-24 Starting an SQL Performance Monitor

This option launches the SQL Performance Monitor Wizard. Type a name for your trace (1 in Figure 7-25) and the schema that will contain the trace data (2). Click **Next** to continue (3).

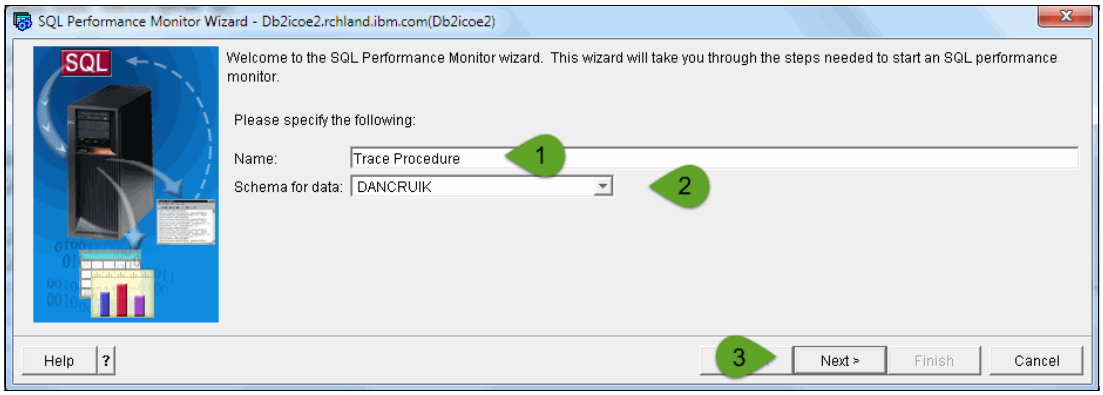

Figure 7-25 SQL Performance Monitor Wizard initial window

The SQL Performance Monitor Wizard filtering options window opens. Click **Next** (1 in Figure 7-26) to continue.

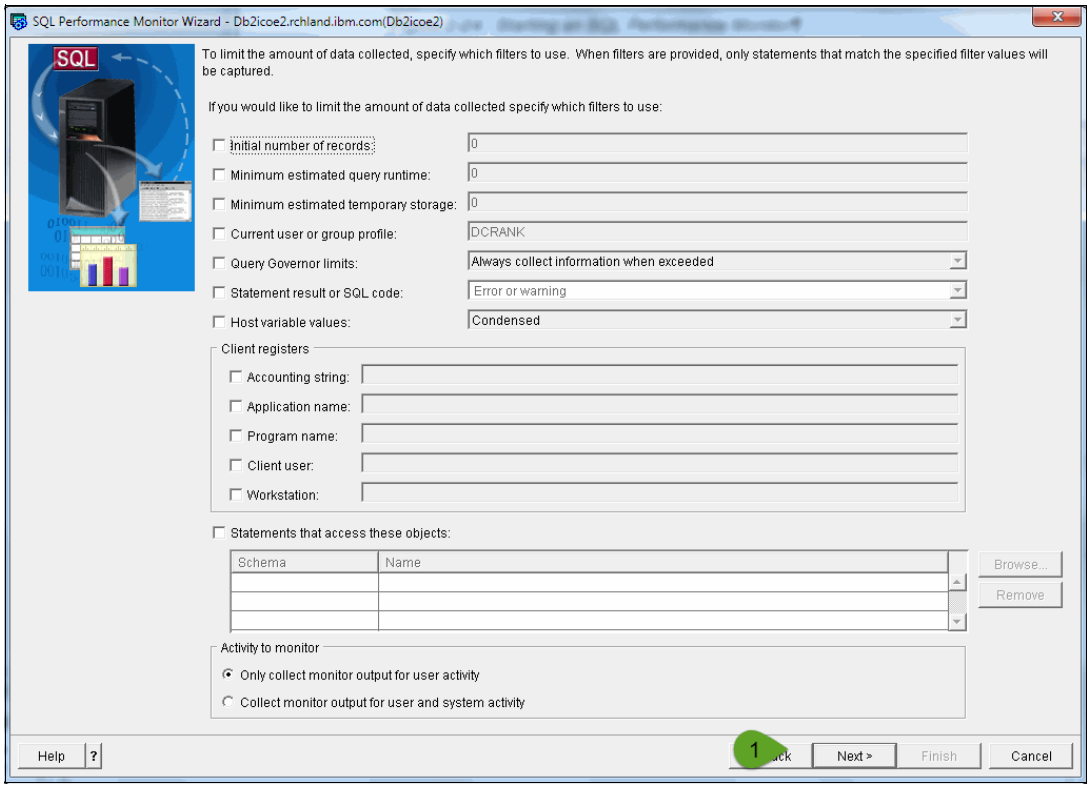

Figure 7-26 SQL Performance Monitor filtering options window

The SQL Performance Monitor Wizard Details window opens. Review the details and click **Finish** (1 in Figure 7-27 on page 213) to start the monitor.

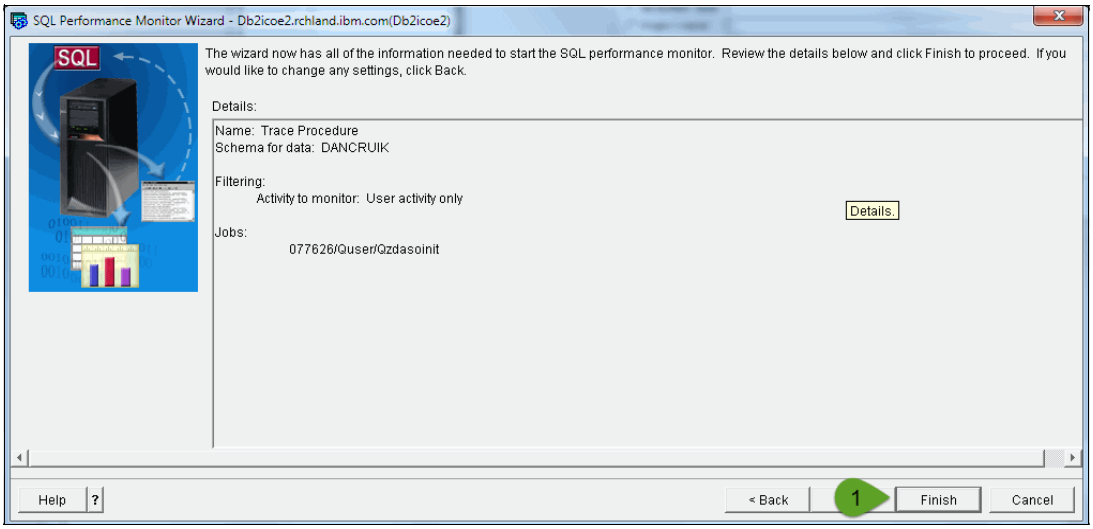

Figure 7-27 SQL Performance Monitor Wizard Details window

After you run your procedure, or procedures, click **Analyze Trace Procedure** from the Monitor menu to analyze the trace data, as shown in Figure 7-28.

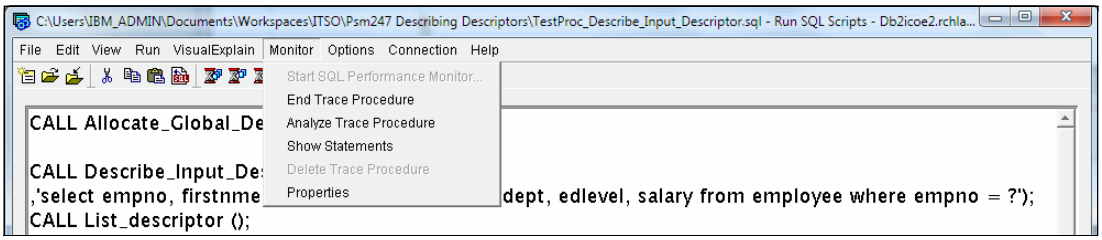

Figure 7-28 Analyzing SQL Performance Monitor trace data

For more information about analyzing data that was generated by the SQL Performance Monitor, see *OnDemand SQL Performance Analysis Simplified on DB2 for i5/OS in V5R4*, SG24-7326.

7.3 Reverse engineering of SQL routines and triggers

Reverse engineering is the process of generating the Data Definition Language (DDL) that is necessary to create an SQL object from the object itself. For example, you can generate the CREATE PROCEDURE statement for a procedure by targeting the procedure itself. Also, you can use this support to build SQL source for several non-SQL objects. For example, you can target a data description specifications (DDS) physical file and generate the corresponding CREATE TABLE statement. It can also be useful to generate the SQL source DDL when the object was created by using a guided or wizard interface that does not directly expose the source.

Three interfaces support reverse engineering:

- System i Navigator
- QSYS2.GENERATE_SQL procedure
- QSQGNDDL application programming interface (API)

All three interfaces support the same object types and generate the corresponding DDL, including warnings about any differences or incompatibilities. This section includes examples from the System i Navigator interface.

7.3.1 System i Navigator

From System i Navigator, use the following steps to list all of the objects in a certain schema:

1. Open your server, in this case, Db2icoe2.
2. Open the databases.
3. Open the schemas. (If your schema is not displayed, right-click **Schemas** and follow the prompts for **Select Schemas to Display**.)
4. Open your schema, in this case, the schema is named SIMONA.

Figure 7-29 shows the type of window that you see on System i Navigator.

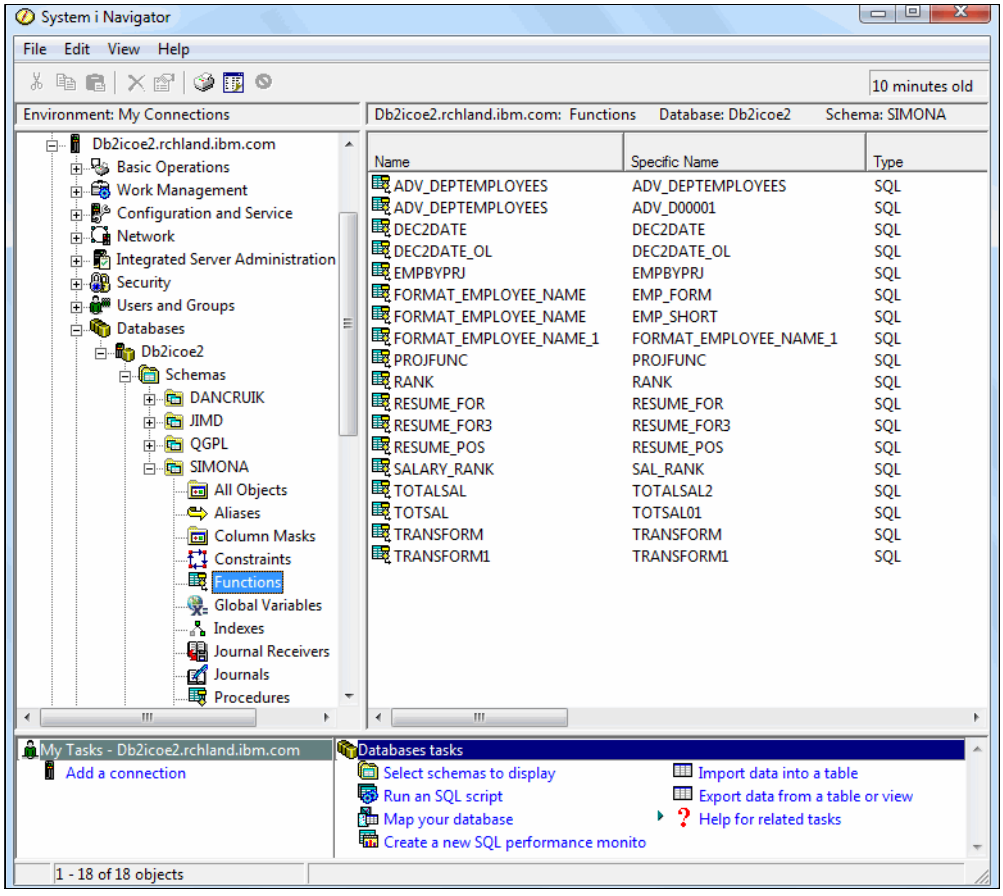

Figure 7-29 System i Navigator objects in schema

After you reach this window, you can select one or more objects by using the Shift key to select adjacent objects or the Ctrl key to select individual objects.

After you select the objects, right-click an object to see the option for Generate SQL, as shown in Figure 7-30.

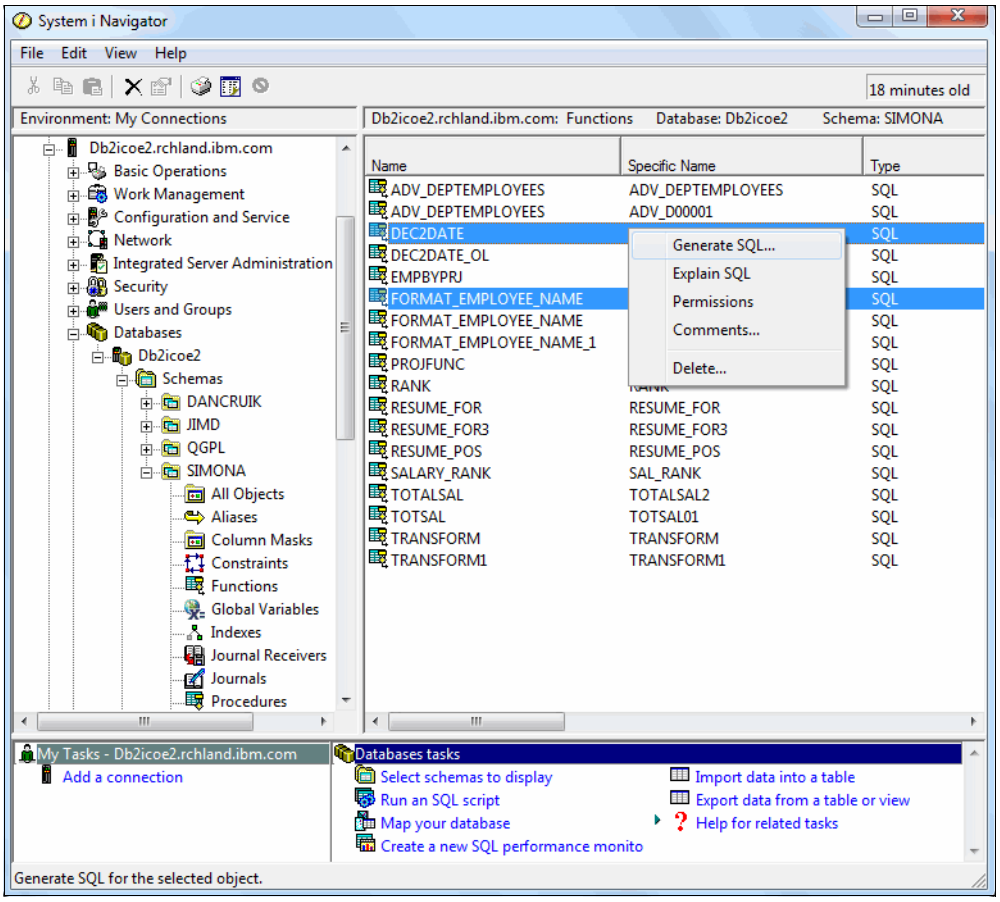

Figure 7-30 System i Navigator - Generate SQL for selected objects

The window that is shown in Figure 7-31 provides the opportunity for you to add and remove items from the list of objects and to specify different options for the generation of the SQL. This window defaults to displaying the resulting DDL in a Run SQL Scripts window.

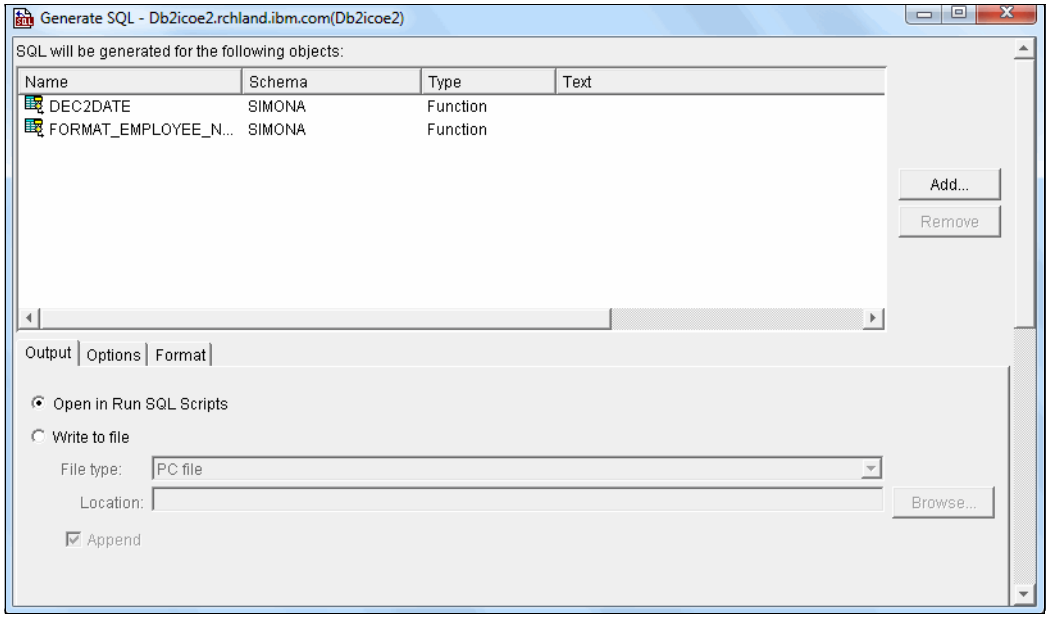

Figure 7-31 System i Navigator Generate SQL output options

Click **Options** on Figure 7-31.

Figure 7-32 shows the Options tab, which provides many choices about how to generate the SQL.

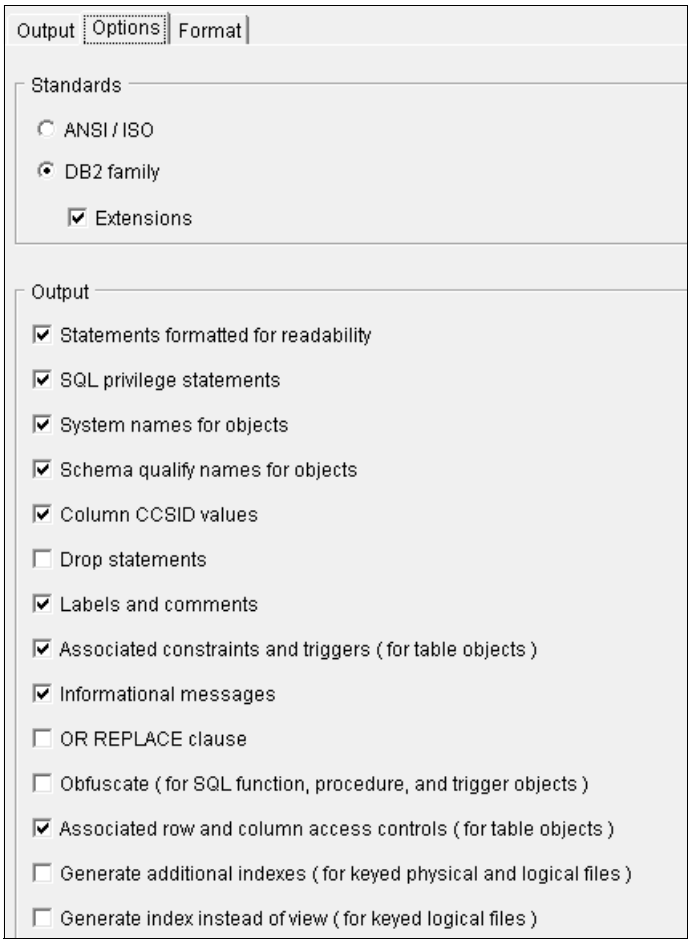

Figure 7-32 System i Navigator Generate SQL options

Click **Generate** at the bottom of the same window to generate the SQL. This part of the window is shown in Figure 7-33.

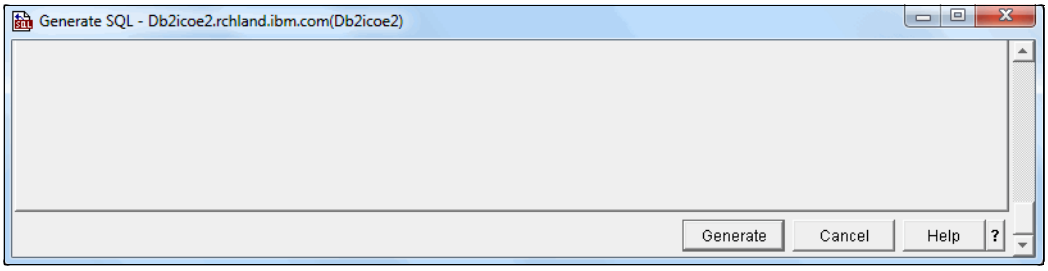

Figure 7-33 System i Navigator Generate SQL

You might see a progress window while the SQL is generated.

Figure 7-34 shows the Run SQL Scripts window that contains the generated SQL.

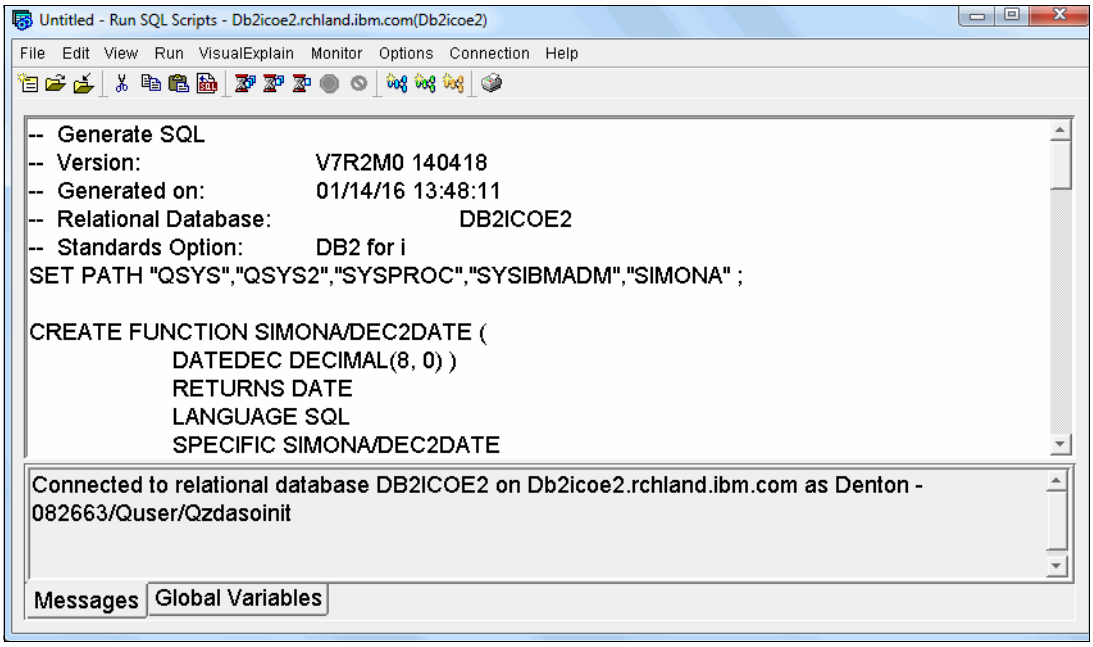

Figure 7-34 System i Navigator generated SQL

7.3.2 GENERATE_SQL

The GENERATE_SQL SQL procedure provides a powerful and programmatic solution for reverse engineering database objects. The following parameters are required:

- DATABASE_OBJECT_NAME: The name of the database object for which DDL is generated. It supports a percent sign '%' wildcard character to allow for all objects or for a generic name, such as 'SALES%'.
- DATABASE_OBJECT_LIBRARY_NAME: The name of the library/schema that contains the database objects. This parameter also supports wildcards as described for the object name.
- DATABASE_OBJECT_TYPE: A character value that specifies the type of objects for which the DDL is generated. It supports all of the SQL object types, such as TABLE, INDEX, VIEW, and ALIAS.

The next three optional parameters specify where to place the generated DDL. If these values are specified, the specified object must exist:

- DATABASE_SOURCE_FILE_NAME: Target source file to receive the generated DDL. If it is not specified, this value defaults to Q_GENSQL.
- DATABASE_SOURCE_FILE_LIBRARY_NAME: Library to contain the target source file. If it is not specified, this value defaults to QTEMP.
- DATABASE_SOURCE_FILE_MEMBER: Member to receive the generated DDL. If it is not specified, this value defaults to Q_GENSQL.

The procedure also returns the generated DDL as a result set that can be processed by its caller.

GENERATE_SQL provides many options for naming, date separators, whether to generate DROP statements for the objects, obfuscation, standards compliance, and many more. One common option is REPLACE_OPTION, which specifies whether the generated DDL will be appended to ('0') or will replace ('1') the contents of the source file.

The following examples show simple calls to the GENERATE_SQL procedure.

This example generates DDL for all tables in a schema and returns the source as a result set:

```
CALL QSYS2.GENERATE_SQL('%', 'JIMD', 'TABLE', REPLACE_OPTION => '0');
```

This example generates DDL for all indexes that start with 'X' within the SIMONA schema and places the output in a file that is named DDLSOURCE/GENFILE member INDEXSRC:

```
CALL QSYS2.GENERATE_SQL('X%', 'SIMONA', 'INDEX',
'GENFILE', 'DDLSOURCE', 'INDEXSRC', REPLACE_OPTION => '0');
```

For additional information, see GENERATE_SQL in the IBM Knowledge Center:

https://ibm.biz/Bd4fRc

7.4 Ownership and authorities of SQL routines and triggers

The ownership and authorities for SQL procedures, triggers, and functions are described.

7.4.1 Ownership

SQL procedures, triggers, and functions all follow the same rules for the ownership of the SQL object and the generated program or service program. These rules depend on the naming convention that is used when they are created. If SQL naming is used, the following rules apply:

- If a user profile with the same name as the schema into which the SQL objects that are created exists, the owner of the SQL object is that user profile.
- Otherwise, the owner of the function is the user profile or group user profile of the thread that executes the CREATE statement.

If system naming is used, the owner of the function is the user profile or group user profile of the thread that executes the CREATE statement.

7.4.2 Authorities

Authorities of SQL routines (procedures and functions) follow these rules:

- If SQL naming is in use at CREATE time, routines are created with the system authority of *EXCLUDE on *PUBLIC.
- If system naming is used at CREATE time, routines are created with the authority to *PUBLIC that is specified by the create authority (**CRTAUT**) parameter of the schema.
- If the owner of the function is a member of a group profile (GRPPRF keyword) and group authority is specified (**GRPAUT** keyword), that group profile also has authority to the routine.

SQL triggers are always created with the system authority of *EXCLUDE on *PUBLIC.

An SQL routine can execute under either the authority of its owner or the user that called it. The default rules depend on the naming convention that was used when the SQL routine was created:

- If SQL naming is used, the SQL routine runs under its owner's authority.
- If system naming is used, the SQL routine runs under the authority of the user that calls or invokes it.

You can also use the SET OPTION clause to explicitly specify the USRPRF and DYNUSRPRF values, which control the authority that is used for static and dynamic SQL statements. For more information, see "SET OPTION" on page 57.

Note: Procedures and functions can run under the authority of either their owner or the user that runs them.

An SQL trigger always executes under the authority of its owner. The users that execute a triggering event do not need authority to the trigger.

Note: An SQL trigger always runs under the authority of its owner.

7.5 Deployment of SQL routines and triggers

SQL routines and triggers are similar to high-level language (HLL) programs because they are often developed in one environment and then deployed to another environment. Two basic approaches are used for deployment:

- Run the SQL CREATE statement on the target environment. The following considerations apply:
  - The source specifications are required on the target environment, which might expose the intellectual property that the routine or trigger contains unless the routine or trigger CREATE statement is obfuscated, as described in "WRAPPED" on page 62.
  - This approach is easier because it avoids any issues that relate to saving the routine or trigger from one library and restoring it to another library.
  - DB2 Query Manager and SQL Development Kit for IBM i is not required to create SQL routines or triggers so you do not need to install this software on the target environment.
  - This approach is often slower than the other method.
- Use Save/Restore commands to save and restore the corresponding programs and service programs. This approach has the following considerations:
  - As explained in 3.2.3, "CREATE OR REPLACE" on page 56, DB2 for i generates programs or service programs to implement the functions that are specified in the CREATE statements for procedures, triggers, and functions.

    These programs and service programs are flagged as an SQL-generated routine or trigger. The program or service program contains the attributes that are needed to instantiate the routine or trigger on the target environment by updating the catalogs.
  - This approach has the same exposure of intellectual property unless the routine or trigger was obfuscated because the catalogs contain the body of the routine or trigger.
  - This approach has limitations that are specific to SQL triggers.
  - This approach is generally faster.

7.5.1 Deploying procedures and user-defined functions

Procedures and UDFs behave in a similar manner when they are deployed.

CL and API support for procedures and functions

The programs and service programs that are generated to implement SQL procedures and functions contain all of the information that is required to instantiate that routine or trigger on another environment.

The program and service program names can be determined by using the EXTERNAL_NAME column in the SYSROUTINES catalog view for procedures and functions.

The following control language (CL) commands (and their API counterparts) will update the SYSROUTINES, SYSPARMS, and SYSROUTINEDEP catalogs to instantiate the underlying program or service program as a procedure or function.

For the commands that create a new instance of a routine, a row is inserted in the SYSROUTINES catalog for the routine. The column EXTERNAL_NAME will contain the name of the newly restored or duplicated object. These commands are listed:

- Save/Restore (**SAVOBJ**/**RSTOBJ** and **SAVLIB**/**RSTLIB**)
- Create Duplicate Object (**CRTDUPOBJ**)
- Copy Library (**CPYLIB**)

For the commands that change an existing instance of a routine, the existing rows in the SYSROUTINES catalog are modified. The column EXTERNAL_NAME will contain the name of the renamed or moved object. These commands are listed:

- Rename Object (**RNMOBJ**)
- Move Object (**MOVOBJ**)

For the commands that delete an existing instance of a routine, the existing rows in the SYSROUTINES catalog will be deleted by using the specific name columns. These commands are listed:

- Clear Library (**CLRLIB**)

  All rows in SYSROUTINES are deleted where the SPECIFIC_SCHEMA column matches the name of the library that is cleared.

- Delete Program (**DLTPGM**) and Delete Service Program (**DLTSRVPGM**)

  The corresponding rows in SYSROUTINES are deleted by using the SPECIFIC_NAME and SPECIFIC_SCHEMA columns.

If a *PGM or *SRVPGM is operated on by using a system command and if the catalog entries for the associated routines cannot be updated, message SQL9015 is issued.

Restore considerations

Several specific behaviors relate to restore processing.

Restoring a program or service program that is associated with a procedure or function does not guarantee that the catalogs will be updated, which is a required step in registering the routine for use in SQL statements.

The following rules for DB2 for i apply:

- If no matching routine is in the catalogs, the object is restored and registered as an SQL routine. This behavior corresponds to the initial deployment of the routine.
- If a routine with the same name and the same signature exists, the behavior depends on the specific name:
  - If the specific name is the same, the catalogs will be updated and the routine is registered as an SQL routine. This case corresponds to the creation of a new version of the routine either to fix a problem or to add function.
  - If the specific name differs, the program or service program will be restored, but it will not be registered as an SQL routine in the catalogs. This situation is reported by an SQL9015 message.
- If a routine exists with the same specific name but the routine has a different name, the program or service program will be restored but it will not be registered as an SQL routine in the catalogs. This situation is reported by an SQL9015 message.
- If a matching routine exists but it has a different signature, the object is restored and registered as an SQL routine.

Note: The program or service program object will be restored in the restore library (`RSTLIB`) that is specified without considering the schema that was specified on its original CREATE statement.

Important: When the signature of a procedure or function changes in a newer version, we recommend that you drop the existing routine before you deploy the new version. If not, you might accidentally overload versions of the procedure or function.

Both procedures and functions support drop by specifying the signature:

```
DROP PROCEDURE proc1 (INT);
```

Additional deployment tools for routines

Many clients use a high-availability configuration where a backup machine needs to match the production environment. Because DB2 for i catalogs are not replicated objects, the entries for procedures and UDFs functions might not match between servers.

To address the complexity, DB2 for i provides a catalog assessment utility that will compare a target system's SYSROUTINE catalog with the version on the local system. This utility is a procedure that is named CHECK_SYSROUTINE. Example 7-2 shows the parameters for this procedure.

Example 7-2 CHECK_SYSROUTINE procedure parameters

```
CALL SYSTOOLS.CHECK_SYSROUTINE(
                <target-database-name>,
                <schema-to-compare>,
                <optional-result-set-parameter>);
```

The named parameters are described:

- **P_REMOTE_DB_NAME** is the name of the remote database.
- **P_SCHEMA_NAME** is the name of the schema to use as the comparison point.
- **P_AVOID_RESULT_SET** is an optional parameter:
  - If the **P_AVOID_RESULT_SET** parameter is not passed, the result set is returned to the caller.
  - If the **P_AVOID_RESULT_SET** parameter is specified and it is *not* set to zero, no result set is returned and the caller can query the SESSION.SYSRTNDIFF table.

The output result set (or the table in the SESSION schema) reports the differences between the two systems based on their respective SYSROUTINES catalogs.

7.5.2 Deploying triggers

Special considerations apply to triggers for two reasons:

- All unqualified references to columns, tables, and other objects are qualified when the trigger is created, which was described in 5.6.2, “Qualifying references” on page 142. This qualification is performed for integrity and auditing purposes because it increases the difficulty for even a knowledgeable database engineer to bypass the business rules that triggers contain.
- Triggers are closely associated with their subject table or view, which increases the likeliness that they will be propagated through other interfaces, such as the Copy File (**CPYF**) and Create Duplicate Objects (**CRTDUPOBJ**) CL commands.

Note: Dropping a table will also drop any triggers that are defined on that table.

Considerations for fully qualified references

Generally, the save/restore method of deploying SQL triggers works only when the referenced objects are always in the same schema. Even when a program that is generated for a trigger is restored to a different library, it still contains references to the objects in the original schemas from its creation. Example 7-3 shows the original CREATE TRIGGER statement.

Note: The reference to the table audit is “bound” at CREATE time to insert rows into the table audit in schema testlib. If the generated program for the trigger is restored into schema prod, it will still update the table audit in schema testlib.

Example 7-3 Original CREATE TRIGGER statement

```
CREATE TRIGGER mytrig
AFTER INSERT ON tab1
REFERENCING NEW AS n
FOR EACH ROW
BEGIN
   INSERT INTO audit
     VALUES(USER, n.c1,n.c2);
END;
```

Example 7-4 shows the fully qualified CREATE TRIGGER statement.

Example 7-4 Fully qualified CREATE TRIGGER statement

```
CREATE TRIGGER mytrig
 AFTER INSERT ON testlib.tab1
 REFERENCING NEW AS n
 FOR EACH ROW
 BEGIN
   INSERT INTO testlib.audit ( A_USER , A_C1 , A_C2 )
     VALUES(USER, N.C1, N.C2);
 END;
```

This consideration also applies to global variables that include defaults where an expression references SQL objects. Those references are qualified when the variable is created.

Enabling and disabling triggers

Triggers have a setting that controls whether they are activated when a change is made to the subject table or view. A trigger that is enabled will be activated during changes. A trigger that is disabled will not be activated during changes. The SQL syntax to manage this setting is available as part of ALTER TRIGGER.

Sometimes, it is beneficial to disable triggers during a mass update or change to the database, such as archival or database modernization. Example 7-5 shows several sample statements to disable triggers before an archival procedure is called.

Example 7-5 Using ALTER TRIGGER to disable and enable a trigger

```
ALTER TRIGGER master_trigger DISABLE;
CALL archival('master');
ALTER TRIGGER master_trigger ENABLE;
```

For more information about the ALTER TRIGGER syntax, see Figure 5-9 on page 121. Other tools, such as IBM i Navigator, provide graphical interfaces for these functions.

Concurrency considerations

Both CREATE TRIGGER and ALTER TRIGGER operations require an exclusive lock on the subject table or view. This lock usually makes it necessary to quiesce the application environment, which can be bothersome to the users.

One alternative is to create triggers that call procedures, as shown in 5.5.5, “Calling a procedure from a trigger” on page 127. This alternative allows the business logic in the procedure to be changed without requiring an exclusive lock on the subject table or view.

Renaming tables

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 then re-create the trigger so that it refers to the renamed table.

Inoperative triggers

An *inoperative trigger* is a trigger that is no longer available to be activated. If a trigger becomes invalid, no INSERT, UPDATE, or DELETE operation will be allowed on the subject table or view.

A trigger is invalid in the following situations:

- The SQL statements in the triggered-action reference the subject table or view; the trigger is a self-referencing trigger; and the table or view is duplicated by using the system **CRTDUPOBJ** CL command.
- The SQL statements in the triggered-action reference tables or views in the from library and the objects are not found in the new library when the table or view is duplicated by using the system **CRTDUPOBJ** CL command.
- The table or view is restored to a new library by using the system **RSTOBJ** or **RSTLIB** CL command, the triggered-action references the subject table or subject view, and the trigger is a self-referencing trigger.

An invalid trigger must first be dropped before it can be re-created by issuing a CREATE TRIGGER statement.

Note: Dropping and re-creating a trigger affects the activation order of a trigger if multiple triggers for the same triggering operation and activation time are defined for the subject table. CREATE OR REPLACE TRIGGER is both a DROP and CREATE of the trigger and also affects the activation order.

8

Creating flexible and reusable procedures

A *reusable procedure* can be called from multiple procedures. A *flexible procedure* can be reused and it accepts different inputs and produces output based on that input. Dynamic Structured Query Language (SQL) falls within this definition. But what about the procedures that take advantage of dynamic SQL? How can they be written so that they easily adapt to the ever-changing needs of the business?

This chapter provides the answers to those questions and more. It includes the following topics:

- Introduction to reusable SQL procedures
- A modular approach to SQL procedure development
- Summary

In this chapter, we develop flexible and reusable procedures that take advantage of the following features:

- Global SQL descriptors
- DB2 global variables
- Procedure parameters with default values
- Importing and exporting data between procedures
- Single procedures that perform all add, update, delete, and read operations
- Implicitly hidden DB2 database columns
- DB2 auto-generated columns, such as row change timestamp

8.1 Introduction to reusable SQL procedures

SQL is the programming language of choice by database engineers (DBEs) to develop data access modules that can be called from any application language that uses SQL. A *data access module* is a procedure that performs any of the following database functions:

- Create one or more table rows
- Update one or more table rows
- Delete one or more table rows
- Retrieve (or provide accessibility to the calling application to retrieve) one or more table rows

These functions are commonly referred to as create, retrieve, update, and delete (CRUD). They represent the four major operations that are used to manipulate data within a relational database.

8.2 A modular approach to SQL procedure development

Many methods are available to write procedures. Most development tools provide a means to create reusable templates. By using templates, you can create a standardized set of CRUD procedures that are copied and then modified to the specifics of each table. These procedures will be executable but they need to be edited before they are used with production tables. This approach is often easier to understand but it also results in many copies of the same standardized logic.

Another approach is to use a flexible procedure that handles any SQL statement that you provide as input. These procedures are production ready, capable of adapting to changes in the data model or business requests for new, specialized ways of filtering, ordering, or displaying the data. This approach can be harder to understand, but it reduces the number of copies of the same logic.

Finally, you can use a hybrid approach that takes advantage of the strengths and weaknesses of both methods. For example, a single module that returns a result set for every and all query requests might not be practical. A new set of modules can be quickly copied, modified, and used to handle all requests by a single application, or for a set of related tables.

You can take advantage of several enhancements to SQL to write modular, reusable, and flexible SQL procedures. The following list of SQL capabilities makes modular development attractive:

- Global variables as default parameters
- Simplified SQL descriptor usage
- Result set consumption
- Extended indicators and indicator arrays

Much of the SQL code that is used to create typical CRUD modules is highly reusable. Many SQL statements support the use of variables, allowing these statements to be coded as common, reusable SQL services. In addition, the use of dynamic SQL minimizes the need to write multiple insert, update, or delete procedures for every table that requires maintenance.

Figure 8-1 contains an overview of the modular approach to SQL procedure development.

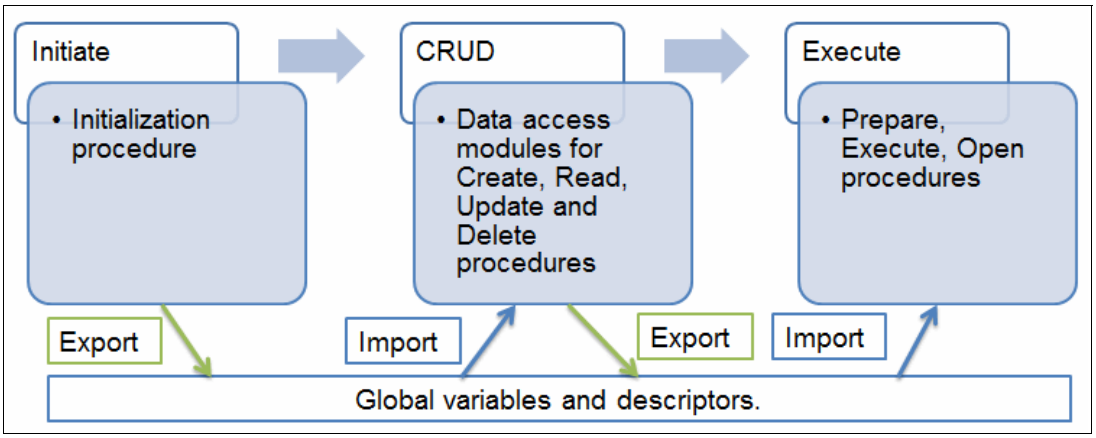

Figure 8-1 A modular approach to SQL development

The four major areas that are represented in Figure 8-1are described:

- Initiate represents the start of a transaction, which is typically the first procedure that is called within a session.[1] Global variables and descriptors are instantiated for reuse within this procedure.
- CRUD represents the data manipulation (INSERT, SELECT, UPDATE, and DELETE) that might occur during the life of the transaction.
- Execute represents the procedures that run the dynamic SQL statements.
- Global variables and descriptors make up the infrastructure between the procedures:
  - The term export is used to describe the assigning of a value to a global variable.
  - The term import is used to describe the assigning of a global variable value to a local variable.

Example 8-1 contains the code for a procedure that uses dynamic SQL and a global descriptor to add a row to a database table.

Example 8-1 Add_Transaction procedure

```
CREATE OR REPLACE PROCEDURE Add_Transaction (
-- Required parameters
   IN P_EMPNO CHAR(6),
   IN P_FIRSTNME VARCHAR(12),
   IN P_MIDINIT CHAR(1),
   IN P_LASTNAME VARCHAR(15),
   IN P_WORKDEPT CHAR(3),
   IN P_EDLEVEL SMALLINT,
   IN P_SALARY DECIMAL(9,2))

   SPECIFIC ADDTRNSACT                                              --1
   COMMIT ON RETURN YES                                             --2

P1 : BEGIN ATOMIC
   -- Local variables
    DECLARE v_sql_string CLOB (2M) DEFAULT                          --3
      'INSERT INTO vemp_inf--
          (empno, firstnme, midinit, lastname, workdept, edlevel, salary   )-4
```

[1] Disconnection from the server or ending a session will destroy all global variables and descriptors for that session.

```
       VALUES (?, ?, ?, ?, ?, ?, ?)';
  DECLARE v_Global_Descriptor VARCHAR(128) DEFAULT 'ADDNEWEMPD';
  DECLARE v_parms , v_type , v_len , v_precision , v_scale , v_indicator, --5
          v_item INTEGER ;
  --Local Error Handling
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE EXIT HANDLER FOR SQLSTATE '23505'
  RESIGNAL SQLSTATE '70001'
  SET MESSAGE_TEXT = 'Duplicate employee number' ;
  CALL allocate_global_descriptor(v_Global_Descriptor);                    --6

  -- Export SQL statement before describing input                          --3
  SET gv_sql_string = v_sql_string;
  CALL describe_input_descriptor();                                        --6

  -- Update input descriptor with values from parameters before calling
  --    execution procedure.                                              --7
  SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 1
     TYPE = 1, LENGTH = 6, DATA = P_EMPNO, INDICATOR = 0 ;
  SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 2
      TYPE = 12, LENGTH = 12, DATA = P_FIRSTNME , INDICATOR = 0 ;
  SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 3
     TYPE = 1, LENGTH = 1, DATA = P_MIDINIT , INDICATOR = 0 ;
  SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 4
     TYPE = 12, LENGTH = 15, DATA = P_LASTNAME , INDICATOR = 0  ;
  SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 5
     TYPE = 1, LENGTH = 3, DATA = P_WORKDEPT , INDICATOR = 0 ;
  SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 6
      TYPE = 5, DATA = P_EDLEVEL , INDICATOR = 0 ;
  SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 7
     TYPE = 3, PRECISION = 9, SCALE = 2, DATA = P_SALARY, INDICATOR = 0  ;

  CALL execute_transaction();                                              --6
END P1
```

Notes about Example 8-1 on page 229:

1 Each procedure that is called within the transaction has a 10-character specific name.[a]

2 This main procedure is responsible for transaction flow so it contains the COMMIT ON RETURN YES clause.

3 The SQL statement string is defined in a local variable and then exported to a global variable for later use.[b]

4 This INSERT statement string is prepared and described. The VALUES clause contains one parameter marker for each input value that matches the column list.

5 These integer local variables are used to update the global input descriptor.

6 These procedures are called as part of this transaction. Only the CALL allocate_global_descriptor statement has an explicit parameter that is defined. All other procedure calls use default parameters and global descriptors.

7 The global descriptor needs to be updated with the data values that are contained in the seven input parameters. This task requires one SET statement for each parameter.

a. A specific name can be up to 128 characters. See 6.5, “CREATE FUNCTION syntax for SQL scalar and table functions” on page 157 for specific name considerations.

b. The global variable is created by the DBE before the creation of this procedure.

Table 8-1 contains a side-by-side comparison of two procedures that use modular techniques. The procedure on the left is from the example that is shown in Example 8-1 on page 229. This procedure controls the process of adding a row to the employee table. The procedure on the right controls the process of adding a row to the department table. The highlighted sections of code indicate where reusable procedures are used.

Table 8-1 Two procedures that use modular techniques

Add_Employee_Transaction procedure	Add_Department_Transaction procedure
CREATE OR REPLACE PROCEDURE Add_Employee_Transaction (-- Required input parameters...) SPECIFIC ADDTRNSACT **COMMIT ON RETURN YES** P1 : BEGIN ATOMIC -- Local variables **DECLARE v_sql_string CLOB (2M)** DEFAULT 'INSERT INTO vemp_info (empno, firstnme, midinit, lastname, workdept, edlevel, salary) VALUES (?, ?, ?, ?, ?, ?, ?)'; DECLARE v_global_DA VARCHAR(128) DEFAULT 'ADDTRNSACT'; **--Procedure logic begins here** **CALL allocate_global_descriptor(v_global_DA);** **-- Export SQL statement** **SET gv_sql_string = v_sql_string;** **CALL describe_input_descriptor();** SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 1 TYPE = 1, LENGTH = 6, DATA = p_empno, INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 2 TYPE = 12, LENGTH = 12, DATA = p_firstnme , INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 3 TYPE = 1, LENGTH = 1, DATA = p_midinit , INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 4 TYPE = 12, LENGTH = 15, DATA = p_lastname , INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 5 TYPE = 1, LENGTH = 3, DATA = p_workdept , INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 6 TYPE = 5, DATA = p_edlevel , INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 7 TYPE =3, PRECISION =9, SCALE =2, DATA =p_salary, INDICATOR = 0 ; **CALL execute_transaction();** END P1	CREATE OR REPLACE PROCEDURE Add_Department_transaction (-- Required input parameters ...) SPECIFIC ADDDPTTRNS **COMMIT ON RETURN YES** P1: BEGIN ATOMIC -- Local variables **DECLARE v_sql_string CLOB (2M)** DEFAULT 'INSERT INTO department (deptno, deptname, mgrno,location, admrdept) VALUES (?, ?, ?, ?, ?)'; DECLARE v_global_DA VARCHAR(128) DEFAULT 'ADDDPTTRNS'; **--Procedure logic begins here** **CALL allocate_global_descriptor(v_global_DA);** **-- Export SQL statement** **SET gv_sql_string = v_sql_string;** **CALL describe_input_descriptor();** SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 1 TYPE = 1, LENGTH = 3, DATA = p_deptno, INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 2 TYPE = 12, LENGTH =36, DATA =p_deptname , INDICATOR =0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 3 TYPE = 1, LENGTH = 6, DATA = p_mgrno , INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 4 TYPE = 1, LENGTH = 16, DATA = p_location , INDICATOR = 0 ; SET SQL DESCRIPTOR GLOBAL v_global_DA VALUE 5 TYPE = 1, LENGTH = 3, DATA = p_admrdept, INDICATOR = 0 ; **CALL execute_transaction();** END P1

8.2.1 Global variables as default parameters

Global variables as default parameters provide you with another valuable tool to create flexible procedures. As shown in Figure 8-1 on page 229, a procedure can *export* a value to a global variable. Then, this value can be *imported* by another procedure that is running within the same session.

By using global variables as parameter defaults, you can hide a column (such as a row change timestamp value) that is required to successfully execute a transaction. This capability can prevent accidental data changes that might affect data integrity.

The use of global variables and default parameters requires a multiple step approach to successfully execute CRUD transactions. The first step is to initialize the global variables that will be used in subsequent steps. The next step is to construct the dynamic SQL CRUD statement that will be executed. The last step is to prepare and execute the dynamic SQL statement.

By executing CRUD transactions in this manner, you can take advantage of modular techniques, creating smaller, easier to maintain reusable procedures. For example, for non-select statements, such as INSERT, UPDATE, and DELETE, only one execute module is required.

Table 8-2 contains information about the global variables that are used in the modular procedures that are shown in this chapter.

Table 8-2 Global variables that are used in modular procedures

Global variable	Data type	Default value	Purpose
gv_global_descriptor	VARCHAR(128)	`NULL`	Stores the name of a global descriptor for later use within a transaction
gv_sql_string	CLOB(2M)	`NULL`	Contains the constructed SQL statement string that is prepared and executed within a transaction
gv_sql_cursor_returnability_attr	VARCHAR(1024)	`WITH RETURN TO CLIENT`	Used to override the cursor return ability attribute from CALLER to CLIENT or vice versa
gv_row_change_timestamp	TIMESTAMP	`NULL`	Used in flexible views for optimistic locking

8.2.2 Simplified SQL descriptor usage

External procedures that use embedded SQL can access a flexible development tool that is known as the *SQL Descriptor Area* (SQLDA). Each host language comes with an include that defines the format of the SQLDA for that language. For more information about using an SQLDA in external procedures, search on `SQLDA` in the IBM Knowledge Center:

`https://ibm.biz/Bd42dh`

The modern SQL programming language can also be used to access SQL descriptors that provide a powerful tool for data-centric database development. Several SQL statements exist that simplify the creation and manipulation of descriptors. A descriptor that is allocated and accessed through SQL is referred to as *an SQL descriptor* as opposed to the SQLDA. Defining and accessing descriptors is an excellent way to use multistep reusable procedures.

As the DBE, the use of SQL descriptors is your only choice in Persistent Stored Modules (PSM) because the language does not support data structures. Keep in mind it is not your only choice. For example, external procedures can choose between SQLDA and descriptors. PSM can only use descriptors. Whatever language you choose, we recommend that you use the SQL descriptor support because it is more flexible and easier to use than the traditional SQLDA support. For more information about using SQL descriptor statements, see Appendix A, “Allocating, describing, and manipulating descriptors” on page 285.

Global descriptors are valuable data-centric programming aids when you develop dynamic, reusable procedures. A global descriptor is similar to a global variable because both global descriptors and global variables enable simpler sharing of values between SQL statements and procedures. In addition, both global variables and descriptors can be shared only across the life of a job. Unlike a global variable, a global descriptor is created by using the ALLOCATE DESCRIPTOR SQL statement and by using the GLOBAL clause.

For more information about global variables and SQL descriptors, see Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5 and Chapter 3, “SQL fundamentals” on page 47.

Example A-3 on page 289 contains code for allocating a global descriptor and then exporting the name of the global descriptor through a global variable (described in Table 8-2 on page 233.

Example 8-2 contains the code for a flexible procedure that imports the global descriptor name through a parameter default and imports the SQL statement string global variables, which are shown in Table 8-2 on page 233, as parameter defaults.

Example 8-2 Flexible procedure

```
CREATE OR REPLACE PROCEDURE Change_Employee_Data (
   IN p_empno CHAR(6),
   -- Optional parameters                                                  --1
   IN p_phoneno CHAR(4) DEFAULT NULL,
   IN p_job CHAR(8) DEFAULT NULL,
   IN p_edlevel SMALLINT DEFAULT NULL,
   IN p_lastname VARCHAR(15) DEFAULT NULL,
   -- Dynamic SQL optional parameters                                      --2
   IN v_sql_string CLOB (2--
   DEFAULT 'UPDATE employee SET (phoneno, job, edlevel,lastname) = (?,?,?,?)
            WHERE empno = ?',
   IN v_global_Descriptor VARCHAR(128)DEFAULT gv_Global_Descriptor)

P1: BEGIN
   -- Program logic starts here
   -- Export SQL statement                                                 --3
    SET gv_sql_string = v_sql_string;
   CALL describe_input_descriptor();

   --Populate Descriptor                                                  --4

   CALL execute_transaction();                                           --5

END P1
```

Notes about Example 8-2 on page 234:

1 These optional parameters represent the changed data. They provide flexibility because they can be omitted or passed in any order on the CALL statement.

2 The dynamic SQL optional parameters provide flexibility:

- If an SQL statement string is not passed, all data parameters are passed and the default SQL string is used. In addition, an SQL statement that references a view that is built over the employee table can be passed as an UPDATE statement.
- If the global descriptor name is not passed, the name from the global variable is used. Otherwise, the name that is passed is used. This approach adds additional security because only an application programmer understands the requirement for a global descriptor.

3 The SQL statement string is exported to the *gv_SQL_string* global variable before the reusable describe_input_descriptor procedure is called.

4 The code that is required to update the input descriptor with the data parameters appears here. For more information, see Example 8-1 on page 229.

5 A call is made to the reusable execute_transaction procedure that is shown in Example 8-3.

The execute_transaction procedure is a reusable service that is called by all types of transactions that use dynamic non-select SQL statements to maintain the database. Example 8-3 contains the code for the execute_transaction procedure.

Example 8-3 The execute_transaction procedure

```
CREATE OR REPLACE PROCEDURE execute_transaction ()                          --1
   SPECIFIC EXCDSCRIPT
   PROGRAM TYPE SUB )                                                       --2
P1: BEGIN
   -- Declare local variables                                               --3
    DECLARE v_sql_string CLOB (2M);
    DECLARE v_Global_Descriptor VARCHAR(128);
   -- Declare  error handling variables
   DECLARE SQLCODE INTEGER DEFAULT 0;
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
         RETURN -1;
   -- Program logic starts here
   -- Import global variables                                               --4
   SET v_Global_Descriptor = gv_Global_Descriptor;
   SET v_sql_string = gv_sql_string;

   -- Validate CRUD statement
    PREPARE EXCDSCRIPT_S1 FROM v_sql_string ;                               --5
   -- Execute CRUD statement
    EXECUTE EXCDSCRIPT_S1  USING SQL DESCRIPTOR GLOBAL v_Global_Descriptor;  --6

 END P1
```

Notes about Example 8-3 on page 235:

1. No required or optional parameters exist for this procedure.
2. The PROGRAM TYPE SUB option is used to create a service program object.
3. The SQL statement string and global descriptor name are defined as local variables.
4. The values for the local variables are imported from the corresponding global variables.
5. The SQL statement that is named EXCDSCRIPT_S1 is assigned to the statement string that is prepared.
6. The EXECUTE statement is used to run the prepared SQL statement EXCDSCRIPT_S1. Unfortunately, no way exists to use a variable statement name. The use of a single module to execute many distinct non-select statements, within a single session, might affect performance. You might want to create more than one execute_transaction module, perhaps one for each application or set of tables.

8.2.3 Result set consumption

The ability to consume a result set in external applications is a staple of most web-based or client-based applications that use SQL to access a database. The ability to consume a result set that is created in a called procedure can also be used in SQL procedures. For more information, see 4.7, “Producing and consuming result sets” on page 91.

Modular procedure example

Assume that a company reengineered its tables by following the IBM Database Reengineering strategy that is explained in *Modernizing IBM i Applications from the Database up to the User Interface and Everything in Between*, SG24-8185. As part of the effort, the company added new ROW CHANGE TIMESTAMP columns to all of its core business tables. These new columns take advantage of the implicitly hidden attribute in order not to expose these columns to applications and user queries. The IT organization now wants to use those ROW CHANGE TIMESTAMP columns for optimistic-locking purposes. (That is, use the row change timestamp column with the unique key on update and delete operations.)

The DBE decided to use a modular approach to minimize the impact to existing applications. Figure 8-2 shows the conceptual use of modular result set consumption.

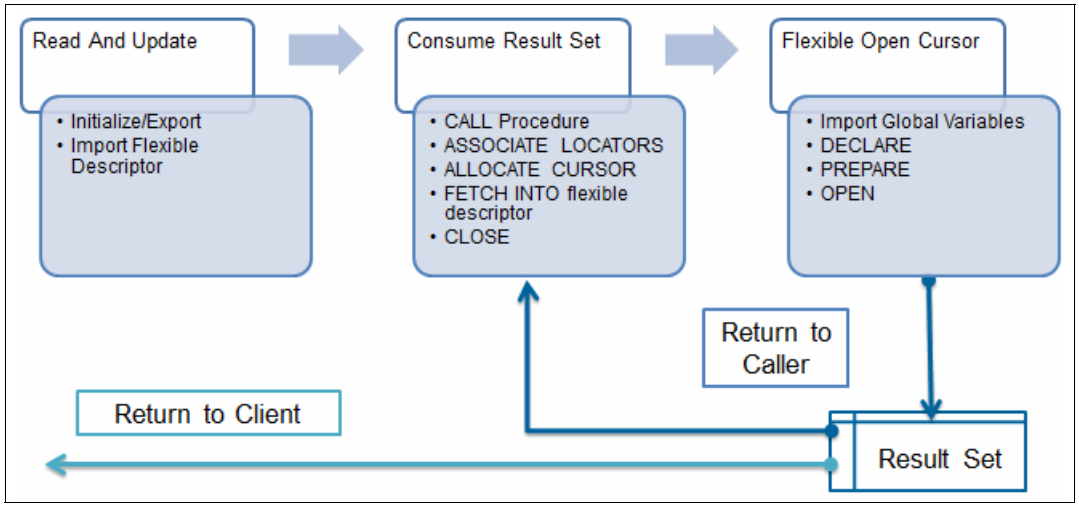

Figure 8-2 Modular approach to result set consumption

In Figure 8-2 on page 236, the procedure that is represented by the box that is titled Flexible Open Cursor is a flexible module with no parameters. Its purpose is to import an SQL statement string, prepare it, and then return the result set based on the returnability attribute (caller or client).

Example 8-4 shows the code for the Flexible Open Cursor module.

Example 8-4 Preparing and opening a flexible cursor

```
CREATE OR REPLACE PROCEDURE prepare_and_open_flexible_cursor ()                --1
   RESULT SETS 1
   LANGUAGE SQL
   SPECIFIC flexcursor
   PROGRAM TYPE SUB
P1: BEGIN
   -- Declare local variables
   DECLARE v_parms SMALLINT;
   DECLARE v_sql_string CLOB (2M);
   DECLARE v_attribute_string VARCHAR(1024);
   DECLARE v_Global_Descriptor VARCHAR(128);
   DECLARE flexcursor_c1 CURSOR FOR flexcursor_S1;
   -- Program logic starts here
   -- Import global variables                                                  --1
   SET v_sql_string = gv_sql_string;
   SET v_Global_Descriptor = gv_Global_Descriptor;
   SET v_attribute_string = gv_sql_cursor_returnability_attr;                   --2

   -- Prepare statement from variable statement string
   PREPARE flexcursor_S1 ATTRIBUTES v_attribute_string FROM v_sql_string ;      --3
   -- Check the global descriptor to see if there are any parameter variables --4
   GET SQL DESCRIPTOR GLOBAL v_GLOBAL_DESCRIPTOR v_parms = COUNT ;
   -- If no parms then open cursor, else open using the descriptor             --5
   IF v_parms = 0 THEN
      OPEN flexcursor_c1;
   ELSE
      OPEN flexcursor_c1 USING SQL DESCRIPTOR GLOBAL v_global_descriptor;       --6
   END IF;
END P1
```

Notes about Example 8-4 on page 237:

1 The SQL statement string and input descriptor are imported from the corresponding global variables.

2 The returnability attribute is imported from a global variable. This attribute is set to return the result set from this procedure to the client.

3 The variable that contains the returnability attribute is referenced on the ATTRIBUTES clause of the PREPARE statement.

4 The GET DESCRIPTOR SQL statement that uses the GLOBAL clause is used to assign the count of the number of variables that are contained in the descriptor that match any parameter markers that might be present in the SQL statement string to a local variable.

5 If no parameter markers exist, open the cursor. Otherwise, open the cursor by using the global input descriptor.

6 The example uses a cursor that is named flexcursor_c1. Unfortunately, no way exists to use a variable cursor name. The use of a single cursor to open many distinct select statements, within a single session, might affect performance. You might want to create more than one flexible cursor module, perhaps one for each application or set of tables.

Example 8-5 contains the SQL statements that are used to test the procedure that is shown in Example 8-4 on page 237.

Example 8-5 Returning the ROW CHANGE TIMESTAMP

```
SET gv_sql_string =                                    --1
   'SELECT v.*, row change timestamp for e
    FROM vemp_info v
    JOIN dcr_db2DDL.employee e
    ON e.empno = v.empno
    WHERE empno = ''650302''';
CALL prepare_and_open_flexible_cursor;                 --3

SET gv_sql_string =                                    --2
   'SELECT a.*, row change timestamp for d
    FROM department a
    JOIN dcr_db2ddl.department d
      ON a.deptno = d.deptno
    WHERE deptno = ''COE''';
CALL prepare_and_open_flexible_cursor;                 --3
```

Notes about Example 8-5:

1 The SET statement is used to populate a global variable with the SQL statement string to return all of the columns from an SQL VIEW. The view was not changed to include the row change timestamp to avoid application changes at the time that the database was reengineered. The core business table is joined by using the table unique key to explicitly include the row change timestamp column.

2 The statement is similar to the first statement except that this statement references an SQL ALIAS instead of a view, and the alias is built over a different table. The same join technique is used. However, the join columns are different.

3 In both cases, the same flexible open cursor procedure is called to produce the result sets.

Figure 8-3 shows the result sets that are created by calling the flexible open cursor procedure.

```
--Result set 1
EMPNO  FIRSTNME MIDINIT LASTNAME   WORKDEPT EDLEVEL SALARY   WHEN_CHANGED
------ -------- ------- ---------- -------- ------- -------- --------------------------
650302 Dan      X       Cruikshank D21           20 50000.00 2015-12-17 12:41:48.874058

--Result set 2
DEPTNO DEPTNAME                   MGRNO  ADMRDEPT LOCATION       WHEN_CHANGED
------ ------------------------------ ------ -------- ---------------
-------------------------
COE    DB2 for i Center of Excellence 650311 D21      Rochester        2015-11-17
11:55:40.561173
```

Figure 8-3 Result sets that are created by calling the flexible open cursor procedure

The procedure that is represented by the box that is titled Consume Result Set in Figure 8-2 on page 236 is a flexible module that has no parameters. Its purpose is to call and associate a cursor for the result set that is produced by the Flexible Open Cursor module. Then, it allocates the cursor, fetches rows from that cursor into the imported descriptor, and then closes the cursor. It returns to the calling procedure. Example 8-6 shows the code for the Consume_Flexible_Result_Set procedure.

Example 8-6 Consume_Flexible_Result_Set procedure

```
CREATE PROCEDURE Consume_Flexible_Result_Set ()
   LANGUAGE SQL
   SPECIFIC COMRESULTS

P1: BEGIN
   --Local variables
   DECLARE V_global_descriptor VARCHAR(128) ;
   DECLARE COMRESULTS_L1 RESULT_SET_LOCATOR VARYING;                          --1

   SET gv_sql_cursor_returnability_attr = 'WITH RETURN TO CALLER';            --2
   CALL prepare_and_open_flexible_cursor();                                    --2

   ASSOCIATE LOCATOR(COMRESULTS_L1) WITH PROCEDURE
      PREPARE_AND_OPEN_FLEXIBLE_CURSOR;                                        --3
   ALLOCATE COMRESULTS_C1 CURSOR FOR RESULT SET COMRESULTS_L1;                --4
   --Import global variables
   SET v_Global_Descriptor = gv_global_descriptor;
   DESCRIBE CURSOR COMRESULTS_C1
      USING SQL DESCRIPTOR GLOBAL v_Global_Descriptor;                        --5

   FETCH COMRESULTS_C1 INTO SQL DESCRIPTOR GLOBAL v_Global_Descriptor;        --6

W1:WHILE (SQLSTATE = '00000') DO
      FETCH COMRESULTS_C1 INTO SQL DESCRIPTOR GLOBAL  v_Global_Descriptor ; --7
   END WHILE;

   CLOSE COMRESULTS_C1;                                                      ---8

END P1
```

Notes about Example 8-6 on page 239:

1 The result locator is defined here. A locator must be defined for each result set that is returned.

2 The returnability attribute global variable is set to caller before the procedure prepare_and_open_flexible_cursor is called, which ensures that the result can be consumed by the calling procedure only. If this attribute is not provided, the called procedure will return the result set to the top of the call stack, which is another example of the flexibility of dynamic SQL.[a]

3 The ASSOCIATE LOCATOR statement is used to link the result set that was created in the called procedure with this procedure.

4 The ALLOCATE CURSOR statement is used to open the linked result set so that it can be consumed in this procedure.

5 The DESCRIBE CURSOR statement is used to describe the format of the result table by using the global descriptor, which is what makes this procedure flexible. Each call to the flexible open cursor procedure can potentially create a different result set each time.

6 The FETCH statement is used to retrieve the first row from the result table that is referenced by the allocated cursor. The data is assigned to the variables in the flexible global descriptor.

7 The FETCH statement is executed within a WHILE loop until the end of the result table is reached, which is signaled by *SQLSTATE* '02000'. In this example, if the result set contains more than one row, the data that is returned in the descriptor will be for the last row that was processed.

8 The procedure closes the allocated cursor before it returns, which will also close the cursor in the called procedure.

a. This example uses a hardcoded procedure name. A variable can be used for the procedure name. This variable will also be used on the ASSOCIATE LOCATOR statement. One use for variable procedure names is to avoid the overuse of duplicate cursor names, as described in Note 6 for Example 8-4 on page 237.

The procedure that is represented by the box that is titled Read And Update in Figure 8-2 on page 236 represents existing or new applications that must read the row change timestamp in addition to the current columns that are accessed for update intent. These applications might already pass a dynamic UPDATE or DELETE statement to the database server. Now, they will be required to call a data access module that returns the data as parameters.

The procedure get_employee_for_update is a working example of the previous scenario. The procedure accepts a single parameter that represents the unique key for a specific table. (The unique key can be more than one column). By definition, a unique key query returns only a single row in the result. The SQL statement string is exported from this procedure. This procedure returns the columns that might be updated. The row change timestamp for the base table is also returned. This column will be combined with the unique key on the subsequent update.

Example 8-7 shows the code for the get_employee_for_update procedure.

Example 8-7 The get_employee_for_update procedure

```
CREATE OR REPLACE PROCEDURE get_employee_for_update (
   IN p_empno CHAR(6),                                              --1
   -- The following parameters are returned by this procedure       --2
   OUT P_firstnme VARCHAR(12),
   OUT P_midinit CHAR(1),
```

```
   OUT P_lastname VARCHAR (15),
   OUT P_workdept CHAR(3),
   OUT P_edlevel SMALLINT,
   OUT P_salary DECIMAL(9,2),
   OUT p_when_Changed TIMESTAMP
   )
   LANGUAGE SQL
   SPECIFIC GETEMP4UPD

P1: BEGIN
   --LOcal variables
   DECLARE v_item INTEGER ;                                              --3
   DECLARE v_Global_Descriptor VARCHAR(128) DEFAULT 'GETEMP4UPD_D1';     --4

   CALL allocate_global_descriptor(v_Global_Descriptor);                 --5

   SET gv_sql_string = 'SELECT v.*, row change timestamp for e
       FROM vemp_info v
       JOIN dcr_db2DDL.employee e
       USING(empno) WHERE empno = ?';                                    --6

   SET SQL DESCRIPTOR GLOBAL v_Global_Descriptor COUNT = 1;              --6
   SET SQL DESCRIPTOR GLOBAL v_Global_Descriptor
      VALUE 1 TYPE = 1, LENGTH = 6, DATA = p_empno ;                     --6

   CALL consume_flexible_result_set ();                                  --7

   SET v_item = 1;                                                       --3
   -- Assign out parameters with the value from the global descriptor    --8
   GET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item p_firstnme = DATA;
   SET v_item = v_item + 1;                                              --3
   GET SQL DESCRIPTOR GLOBAL v_Global_Descriptor VALUE v_item p_midinit  = DATA;
   SET v_item = v_item + 1;
   GET SQL DESCRIPTOR GLOBAL v_Global_Descriptor VALUE v_item p_lastname = DATA;
   SET v_item = v_item + 1;
   GET SQL DESCRIPTOR GLOBAL v_Global_Descriptor VALUE v_item p_workdept = DATA;
   SET v_item = v_item + 1;
   GET SQL DESCRIPTOR GLOBAL v_Global_Descriptor VALUE v_item p_edlevel   = DATA;
   SET v_item = v_item + 1;
   GET SQL DESCRIPTOR GLOBAL v_Global_Descriptor VALUE v_item p_salary   = DATA;
   SET v_item = v_item + 1;
   GET SQL DESCRIPTOR GLOBAL v_Global_Descriptor
         VALUE v_item p_when_changed = DATA; 7                           --9

END P1
```

Notes about Example 8-7 on page 240:

1 This input column represents the unique key for the table.

2 These columns are returned to the application that called this procedure.

3 This variable is used by the SET DESCRIPTOR statement. It contains the current item value of the variable that is retrieved from the descriptor.

4 This unique global descriptor name is for this procedure. It is based on the 10-character specific name.

5 This reusable module allocates a global descriptor and exports the descriptor name for later use by the result sets creator and consumer procedures.

6 The procedure uses the SET DESCRIPTOR to assign the number of items. Because this descriptor will be used initially for input, only one item (empno) exists. After the count is set, the next step updates the first item with the type, length, and value of p_empno.

7 The consume result set flexible module is called, which will call the result set module. If it returns successfully (will be described in more detail in 2.6, “Error handling” on page 29), the global descriptor will contain the values for the output parameters.

8 The GET DESCRIPTOR statement is used to assign the values from the descriptor to the output parameters. The flexibility is demonstrated because the same descriptor is used for both input and output. The *v_item* value is incremented by 1 after each SET. This value is intended for ease of maintenance and reusability of the source as a template for cloning.

9 The row change timestamp (*p_when_changed*) is retrieved as the last variable. The placement is also intended for ease of cloning.

Example 8-8 shows the results of running a test of the get_employee_for_update procedure.

Example 8-8 CALL get_employee_for_update

```
call GET_EMPLOYEE_FOR_UPDATE('650302',' ', ' ', ' ', ' ', 0, 0.00, ' ');  --1

Results:
> call GET_EMPLOYEE_FOR_UPDATE('650302',' ', ' ', ' ', ' ', 0, 0.00, ' ') --2

Return Code = 0                                                           --3

-- The returned values from the successful execution of the flexible result set
creator and consumer modules.                                             --4
Output Parameter #2 = Dan
Output Parameter #3 = X
Output Parameter #4 = Cruikshank
Output Parameter #5 = D21
Output Parameter #6 = 20
Output Parameter #7 = 70000.00
Output Parameter #8 = 2015-10-21 08:30:22.360891
```

Notes about Example 8-8 on page 242:

1 Normally, this procedure is called by an application. The System i Navigator Run SQL Scripts tool was used in place of application code to test this procedure. For more details about development tools, see Chapter 7, “Development and deployment” on page 189.

2 The CALL statement contains the input variable for the unique key, plus empty placeholders for the output parameters. This statement might differ, depending on your development tool.

3 The return code of 0 represents a successful execution. For more information about the use of the RETURN statement in procedures, see 2.6, “Error handling” on page 29.

4 The returned data for this row is now available to the application for display. The row change timestamp is available for use as an additional search argument on the WHERE clause of an SQL UPDATE statement.

8.2.4 Extended indicators and more

The use of parameter markers and the **USING** keyword can greatly simplify the coding of dynamic SQL statements. However, the use of parameter markers and the **USING** keyword can create new challenges for you when you combine this capability with default parameters and named arguments on procedure calls. Sometimes, the application users might call a procedure to pass all of the defined parameters. Other application users will call the procedure to pass only the parameters that they are interested in.

Modifying data without extended indicators

The following examples represent the different types of modification transactions that might be performed against a table throughout the business day:

- Add new data
- Delete data
- Change data
- Reset data

The procedures were written to support the employee department table maintenance application in a Human Resources system. The goal was to take advantage of reusable and flexible SQL procedures. The examples will be reviewed in the order that they were created for the new application. The last example will be the test script with the results of executing different transactions.

Add_department_transaction

The first example is for the add procedure. In most applications, the add procedure is the first data access module that is created. Example 8-9 shows the code for the add_department_transaction procedure.

Example 8-9 ADD_DEPARTMENT_TRANSACTION procedure

```
CREATE PROCEDURE add_department_transaction (
   -- Required parameters                                          --1
   IN p_deptno CHAR(3),
   IN p_deptname VARCHAR(36),
   IN p_mgrno CHAR(6),
   IN p_location CHAR(16),
   IN p_admrdept CHAR(3)
)
```

```
   SPECIFIC ADDDPTTRNS
   COMMIT ON RETURN YES                                                    --2


Add_Dept : BEGIN ATOMIC
   -- Local variables
    DECLARE v_sql_string CLOB(2M)                                          --3
    DEFAULT 'INSERT INTO department (deptno, deptname, mgrno,location, admrdept)
          VALUES (?, ?, ?, ?, ?)';                                         --4

   DECLARE v_Global_Descriptor VARCHAR(128) DEFAULT 'ADDDPTTRNS';
   DECLARE v_item INTEGER ;                                                --5

   CALL allocate_global_descriptor(v_Global_Descriptor);

   -- Export SQL statement
    SET gv_sql_string = v_sql_string;

   CALL describe_input_descriptor();

   SET v_item = 1;                                                         --6
   -- Variable assignment begins her--e
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
      TYPE = 1, LENGTH = 3, DATA = p_deptno, INDICATOR = 0 ;
   SET v_item = v_item + 1;                                                --7
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
       TYPE = 12, LENGTH = 36, DATA = p_deptname , INDICATOR = 0 ;
   SET v_item = v_item + 1;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
      TYPE = 1, LENGTH = 6, DATA = p_mgrno , INDICATOR = 0 ;
   SET v_item = v_item + 1;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
      TYPE = 1, LENGTH = 16, DATA = p_location , INDICATOR = 0  ;
   SET v_item = v_item + 1;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
      TYPE = 1, LENGTH = 3, DATA = p_admrdept, INDICATOR = 0 ;

CALL execute_transaction();                                                --8

END Add_Dept
```

Notes about Example 8-9 on page 243:

1 The DBE chose to list all required parameters first, followed by parameters that were defined with defaults. In the add transaction, all parameters are required.

2 A commit operation will be performed if this procedure completes successfully.

3 The SQL statement string variable is declared as a CLOB with a size of 2 MB, which is the maximum size of an SQL statement. In highly normalized databases, 2 MB might be large. However, the DBE chose to standardize on the high side.

4 The procedure contains the entire INSERT SQL statement string. The DBE chose to list the columns and the corresponding parameters for both documentation and eyesight validation that the number of parameter markers equals the number of columns.

5 The variable *v_item* will be used to represent the current item that is assigned a value through the SET DESCRIPTOR SQL statement. By using a variable, the DBE can easily add new variables without needing to know the ordinal position within the descriptor.

6 The *v_item* is set to 1 before the first SET DESCRIPTOR SQL statement is issued.

7 The *v_item* variable is incremented by 1 after each SET DESCRIPTOR SQL statement is executed.

8 The reusable execute_transaction procedure is called after the data from all parameters is assigned to the corresponding descriptor variable.

Delete_department_transaction

It is logical that the creation of the add module will be followed by a delete module. The delete module will use the row change timestamp column as an additional search argument on the WHERE clause of the SQL DELETE statement. The DBE will take advantage of *flexible views* to perform this task. A flexible view is an SQL view that references a global variable on the WHERE clause.

Example 8-10 contains the SQL Data Definition Language (DDL) statements to create a timestamp global variable and an SQL view that references that global variable.

Example 8-10 Row change timestamp global variable and a flexible view

```
CREATE OR REPLACE VARIABLE gv_row_change_timestamp TIMESTAMP;                --1

CREATE OR REPLACE VIEW department_change AS
   SELECT * FROM dcr_db2ddl.DEPARTMENT
      WHERE when_changed = gv_row_change_timestamp;                          --2
```

Notes about Example 8-10:

1 The default value for the global variable is NULL.

2 The global variable is referenced on the WHERE clause within the SQL view. Any attempt to access data through this view without first assigning a valid value to the global variable will result in an empty result set.

Example 8-11 shows the code for the delete_department_transaction procedure.

Example 8-11 The delete_department_transaction procedure

```
CREATE PROCEDURE delete_department_transaction (
   --Required parameters                                                     --1
   IN p_deptno CHAR(3),                                                     --1
```

```
   IN p_last_Changed TIMESTAMP                                          --1
   )
   LANGUAGE SQL
   SPECIFIC DLTDPTTRNS
   COMMIT ON RETURN YES                                                 --2

Dlt_Dept : BEGIN ATOMIC
   -- Local variables
    DECLARE v_sql_string CLOB(2M)
    DEFAULT 'DELETE department_change WHERE deptno = ?';                --3

   DECLARE v_Global_Descriptor VARCHAR(128) DEFAULT 'DLTDPTTRNS';

   CALL allocate_global_descriptor(v_Global_Descriptor);

   -- Export SQL statement
    SET gv_sql_string = v_sql_string;
    SET gv_ROW_CHANGE_TIMESTAMP = p_last_Changed;

   CALL describe_input_descriptor();

   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE 1
      TYPE = 1, LENGTH = 3, DATA = p_deptno, INDICATOR = 0 ;            --4

   CALL execute_transaction();                                          --5

END Dlt_Dept
```

Notes about Example 8-11:

1 All parameters are required in the delete transaction. Unlike the add transaction, only the key column (p_deptno) and the row change timestamp are defined.[a]

2 A commit operation will be performed if this procedure completes successfully.

3 The procedure contains the entire DELETE SQL statement string. The statement references the department_change view. Only the deptno column is referenced on the WHERE clause. During execution, DB2 will merge the WHERE clause from the SQL statement with the WHERE clause from the view.

4 Only one value needs to be assigned to a descriptor variable.

5 The reusable execute_transaction procedure is called after the data is assigned to the corresponding descriptor variable.

a. In this case, p_deptno represents the unique key constraint for the department table. If a unique constraint was made up of multiple keys, all keys must be passed through parameters.

Change_department_transaction

The change module is the next module to create. It uses the department_change view because it also requires the table unique key and the row change timestamp global variable. In addition, not all values need to be passed. This capability is intended to support multiple applications that might update different columns (or sets of columns) within a single table.[2]

Table 8-3 shows how this example might be handled by using a flexible SQL descriptor within a procedure.

Table 8-3 Parameter mix and flexible descriptors

Pseudo CALL statement	UPDATE SET clause	Flexible descriptor COUNT
CALL Proc (K,T,a,b,c)	SET (a,b,c) = (?,?,?) WHERE key = ?	4 = 3 default parms plus key (1)
CALL Proc (K,T,a) or CALL proc (K, T, pb =>b or CALL proc(K, T, pc=> c	SET a = ? WHERE key = ? or SET b = ? WHERE key = ? or SET c = ? WHERE key = ?	2 = 1 default parm plus key (1)
CALL Proc (K,T,a,b) or CALL Proc (K,T,a,pc=>c or CALL Proc(K,T,pb=>b, pc=>c)	SET (a,b) = (?,?) WHERE key = ?or SET (a,c) = (?,?) WHERE key = ? or SET (b, c) = (?,?) WHERE key = ?	3 = 2 default parms plus key (1)

The information in Table 8-3 is described in more detail:

- The pseudo CALL statements represent various scenarios where one or more default parameters are passed to the update procedure. This CALL statement might not be syntactically correct.
- The UPDATE SET clause represents the possible combinations of columns and parameter markers based on the number of non-default parameters. Keep in mind that these parameters can be omitted or passed in any order.
- The flexible descriptor COUNT column represents the number of variables that are required in the descriptor based on the number of parameters that are passed. For example, if only the third parameter (c) is passed to the procedure, the descriptor count (after a DESCRIBE statement is executed) will be 2.

By using Table 8-3 as a guide, the DBE wrote a procedure to accommodate the various scenarios. The code for this procedure is shown in Example 8-12.

Example 8-12 The change_department_transaction procedure

```
CREATE OR REPLACE PROCEDURE Change_Department_transaction (
   --Required parameters                                          --1
   IN p_deptno CHAR(3),
   IN p_last_Changed TIMESTAMP ,
   -- Optional parameters                                        --2
   IN p_deptname VARCHAR(36) DEFAULT NULL,
   IN p_mgrno CHAR(6) DEFAULT NULL,
   IN P_location CHAR (16) DEFAULT NULL,

   -- Dynamic SQL optional parameters
   IN v_sql_string CLOB (2M)DEFAULT                             --3
      'UPDATE department_change
         SET (deptname, mgrno, location) = (?,?,?)
```

[2] This capability is common in heritage environments where the database is not well-defined.

```
        WHERE deptno = ?')

   LANGUAGE SQL
   SPECIFIC CHGDPTTRNS
   COMMIT ON RETURN YES                                                    --4

P1: BEGIN
   DECLARE v_item, v_count INTEGER ;
   DECLARE v_Global_Descriptor VARCHAR(128) DEFAULT 'CHGDPTTRNS';

   -- Program logic starts here
   CALL allocate_global_descriptor(v_Global_Descriptor);

   -- Export local variables                                               --5
   SET gv_sql_string = v_sql_string;
   SET gv_ROW_CHANGE_TIMESTAMP = p_last_Changed;
   CALL describe_input_descriptor();

   -- Get the count of variables from the descriptor header                --6
   GET SQL DESCRIPTOR GLOBAL v_global_descriptor v_count = COUNT ;
   -- Set the starting descriptor value
   SET v_item = 1;                                                         --7

   CASE v_count                                                            --8
      WHEN 2 THEN
      -- Only the key and 1 non_default value was passed
         If p_deptname IS NOT NULL THEN
            -- Department name change only
            -- UPDATE ... SET deptname = ?
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
                TYPE = 12, LENGTH = 36, DATA = p_deptname , INDICATOR = 0 ;
         ELSEIF p_mgrno IS NOT NULL THEN
            -- Manager Number change only
            -- UPDATE ... SET mgrno = ?
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
               TYPE = 1, LENGTH = 6, DATA = p_mgrno , INDICATOR = 0 ;
         ELSE
            -- Must be Location change only
            -- UPDATE ... SET location = ?
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
               TYPE = 1, LENGTH = 16, DATA = p_location , INDICATOR = 0  ;
         END IF;
      WHEN 3 THEN
      -- only the key and 2 non_default values were passed
         IF p_deptname IS NOT NULL AND p_mgrno IS NOT NULL THEN
            -- Department and manager change only
            -- UPDATE ... SET (deptname,mgrno) = (?,?)
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
                TYPE = 12, LENGTH = 36, DATA = p_deptname , INDICATOR = 0 ;
            SET v_item = v_item + 1;
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
               TYPE = 1, LENGTH = 6, DATA = p_mgrno , INDICATOR = 0 ;
         ELSEIF p_deptname IS NOT NULL AND p_location IS NOT NULL THEN
            -- Department and location change only
            -- UPDATE ... SET (deptname,location) = (?,?)
```

```
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
                TYPE = 12, LENGTH = 36, DATA = p_deptname , INDICATOR = 0 ;
            SET v_item = v_item + 1;
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
               TYPE = 1, LENGTH = 16, DATA = p_location , INDICATOR = 0  ;
         ELSE
            -- Must be manager and location change only
            -- UPDATE ... SET (mgrno,location) = (?,?)
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
               TYPE = 1, LENGTH = 6, DATA = p_mgrno , INDICATOR = 0 ;
            SET v_item = v_item + 1;
            SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
               TYPE = 1, LENGTH = 16, DATA = p_location , INDICATOR = 0  ;
         END IF;
      WHEN 4 THEN
      -- The key and all parameters have values
         -- Department, manager and location change
         -- UPDATE ... SET (deptname, mgrno,location) = (?,?,?)
         SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
             TYPE = 12, LENGTH = 36, DATA = p_deptname , INDICATOR = 0 ;
         SET v_item = v_item + 1;
         SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
            TYPE = 1, LENGTH = 6, DATA = p_mgrno , INDICATOR = 0 ;
         SET v_item = v_item + 1;
         SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
            TYPE = 1, LENGTH = 16, DATA = p_location , INDICATOR = 0  ;
      ELSE                                                              --9
         SIGNAL SQLSTATE '70020'
         SET MESSAGE_TEXT = 'The number of parameters passed is not correct';
   END CASE;

   SET v_item = v_item + 1;                                             --10
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
      TYPE = 1, LENGTH = 3, DATA = p_deptno , INDICATOR = 0  ;

   CALL execute_transaction();                                          --11
END P1
```

Notes about Example 8-12 on page 247:

1 Like the delete transaction, the key column and row change timestamp parameters are required.

2 All other parameters are optional. However, at least one default parameter must be passed.

3 The final default parameter is the SQL UPDATE statement string that contains the columns to update and the corresponding parameter markers. The number of columns on the left side of the equal predicate in the SET clause must equal the number of parameter markers on the right side of the SET clause equal predicate. If the UDPATE statement is not passed, a default string that contains all valid default columns is used.

4 A commit operation is performed if this procedure completes successfully.

5 The UPDATE statement string and row change timestamp variables are exported before the describe_input_descriptor reusable procedure is called.

6 The variable *v_count* is assigned the COUNT value from the descriptor on the successful completion of the describe_input_descriptor procedure.

7 The variable *v_item* is set to 1 to represent the first variable in the descriptor.

8 The CASE statement reduces the number of IF tests that are required to determine how many and which descriptor items need to be assigned a value. The WHEN clauses are explained in more detail:

- **WHEN 2**: Only one default parameter was passed, as shown in row 3 of Table 8-3 on page 247. An IF test is used to determine the parameter that was passed and to use the appropriate SET DESCRIPTOR statement.
- **WHEN 3**: Two of the three default variables were passed, as shown in row 4 of Table 8-3 on page 247. An IF test is used to determine the two parameters that were passed and to use the appropriate SET DESCRIPTOR statements. The *v_item* variable is incremented to represent the next item to set in the descriptor.
- **WHEN 4**: All three of the default variables were passed, as shown in row 2 of Table 8-3 on page 247. An IF test is not required. The SET DESCRIPTOR statements are executed in the order of the passed parameters. The *v_item* variable is incremented to represent the next item to set in the descriptor.

9 If the ELSE leg of the CASE statement is reached, an error is signaled to the calling application to indicate that the number of columns and parameter markers in the UPDATE statement string are not valid for this procedure.

10 The *v_item* is incremented to represent the variable or variables that were used for key columns.

11 The reusable execute_transaction procedure is called after all data is assigned to the corresponding descriptor variables.

Reset_department_defaults_transaction

The business requires the management of a seasonal workforce. A special department is been created. However, the manager and location change each season. The business needs to retain the department row historical reporting. Therefore, the department row cannot be deleted. The business wants to set the department manager and location columns to their table-defined defaults.

To accomplish this requirement, the DBE created a procedure that takes advantage of the reusable procedures and the DEFAULT option of the SQL UPDATE statement.

Example 8-13 shows the code for the reset_department_defaults_transaction procedure.

Example 8-13 The reset_department_defaults_transaction procedure

```
CREATE PROCEDURE reset_department_defaults_transaction (
   --Required parameters                                                    --1
   IN p_deptno CHAR(3),
   IN p_last_Changed TIMESTAMP ,

   -- Dynamic SQL optional parameters                                       --2
   IN v_sql_string CLOB(2M)
      DEFAULT 'UPDATE department_change SET (mgrno, location) = (DEFAULT,DEFAULT)
WHERE deptno = ?')

   LANGUAGE SQL
   SPECIFIC RSTDPTTRNS
   COMMIT ON RETURN YES                                                    --3

P1: BEGIN
   DECLARE v_item INTEGER DEFAULT 1;
   DECLARE v_Global_Descriptor VARCHAR(128) DEFAULT 'RSTDPTTRNS';

   -- Program logic starts here
   CALL allocate_global_descriptor(v_Global_Descriptor);

   -- Export local variables                                               --3
   SET gv_sql_string = v_sql_string;
   SET gv_ROW_CHANGE_TIMESTAMP = p_last_Changed;

   CALL describe_input_descriptor();
   -- Assign the key column values to the corresponding descriptor variables--4
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE v_item
      TYPE = 1, LENGTH = 3, DATA = p_deptno , INDICATOR = 0  ;

   CALL execute_transaction();                                              --5
END P1
```

Notes about Example 8-13:

1 In the reset transaction, the key and row change timestamp parameters are required.

2 The only default parameter is the SQL UPDATE statement string that contains the columns to be reset through the DEFAULT special value with the key columns and corresponding parameter markers. If not passed, a default string that contains all columns to reset is used.

3 A commit operation is performed if this procedure completes successfully.

4 Only one value needs to be assigned to a descriptor variable.

6 The reusable execute_transaction procedure is called after the data is assigned to the corresponding descriptor variable.

Transaction work flow

Example 8-14 shows an example of the order in which the previous procedures can be executed within a transaction work flow. This transaction work flow is likely executed through a user interface (UI) device. However, the creation of a UI is outside the scope of this book. Therefore, for simplicity, an SQL scripting tool was used to simulate the UI.

Example 8-14 A sample workflow

```
--Add new department
CALL add_department_transaction ( 'XXX' , 'PL/1 Application Development' ,
'650311' , 'Rochester',  'D21' );
--Retrieve the newly created department
call get_department_data_for_update('XXX',' ', ' ', ' ', ' ');
Return Code = 0
Output Parameter #2 = PL/1 Applicatiion Development
Output Parameter #3 = 650311
Output Parameter #4 = Rochester
Output Parameter #5 = 2015-12-02 10:57:08.396933                              --1
--Uh-oh, funding for the new department no longer exists, delete the department
-- using the department number and row change timestamp from above
call delete_department_transaction(
'XXX',TIMESTAMP('2015-12-02 10:57:08.396933');                                --2
--Create a department that is going somewhere
CALL add_department_transaction ( 'COE' , 'DB2 for i Center of Excellence' ,
'650311' , 'Rochester',  'D21' );
--Retrieve the department information
call get_department_data_for_update('COE',' ', ' ', ' ', ' ');
Return Code = 0
Output Parameter #2 = DB2 for i Center of Excellence
Output Parameter #3 = 650311
Output Parameter #4 = Rochester
Output Parameter #5 = 2015-12-02 11:06:08.47915                             --2
--Change the location to be more specific
CALL change_department_transaction
   ('COE' , TIMESTAMP('2015-12-02 11:06:08.47915'),                          --2
   DEFAULT, DEFAULT, P_location=> 'Roch Lab Svcs   ',                        --3
   v_sql_string => 'UPDATE department_change SET location = ?
   WHERE deptno = ? ');
-- Validate the change
call get_department_data_for_update('COE',' ', ' ', ' ', ' ');
Return Code = 0
Output Parameter #2 = DB2 for i Center of Excellence
Output Parameter #3 = 650311
Output Parameter #4 = Roch Lab Svcs                                         --4
Output Parameter #5 = 2015-12-02 11:11:14.273685                            --4
-- Cannot agree on the new name so reset it to the default
CALL reset_department_defaults_transaction
   ('COE' , TIMESTAMP('2015-12-02 11:11:14.273685'),
   v_sql_string => 'UPDATE department_change SET location = DEFAULT
   WHERE deptno = ? ');
--Discount double check
call get_department_data_for_update('COE',' ', ' ', ' ', ' ');
Return Code = 0

Output Parameter #2 = DB2 for i Center of Excellence
Output Parameter #3 = 650311
```

```
Output Parameter #4 =                                                    --5
Output Parameter #5 = 2015-12-02 11:19:23.354725                         --5

--Time to go home
```

Notes about Example 8-14 on page 252:

1 The value for the row change timestamp column is automatically generated by DB2 for all INSERT and UPDATE operations. To learn more about row change timestamps, see the CREATE TABLE statement in the DB2 for i SQL reference at this website:

https://ibm.biz/Bd42dh

2 The row change timestamp that was generated from the INSERT is passed to the delete procedure, and it is used to set the row change timestamp global variable.

3 The DEFAULT keyword is used to inform DB2 that the two parameters that were defined between the row change timestamp and the p_location parameters will use the default that was defined for the input parameter (NULL in this case).

4 The results of running the get_department_data_for_update procedure verify that the location was changed and that the row change timestamp was updated.

5 The results of running the get_department_data_for_update procedure verify that the location was reset and that the row change timestamp was updated.

Example 8-15 contains a call to the change_department_transaction procedure that passes an incorrect number of columns and variables on the UPDATE string variable.

Example 8-15 Invalid call to the change_department_transaction procedure

```
--Test for invalid parameter condition
CALL change_department_transaction
   ('XXX' , TIMESTAMP('2015-12-01 15:37:53.927415'),
   'DB2 CoE 04', '650321', DEFAULT,
   v_sql_string => 'UPDATE department_change
       SET (deptname,mgrno, location, deptno) = (?,?,?,?)            --1
       WHERE deptno = ? ');

--Results of test
SQL State: 70020                                                      --2
Vendor Code: -438
Message: [SQL0438] The number of parameters passed is not correct     --3
```

Notes about Example 8-15:

1 The UPDATE statement is coded correctly. However, the procedure CASE statement does not recognize the descriptor count of 5 (the number of parameter markers in the statement). Investigation reveals that the column deptno is included in the SET column list. This column is a key column and must appear here.

2 The SQL state code is not a standard SQL state. The value is sent by the SQL SIGNAL statement that is coded on the ELSE leg of the CASE statement in the procedure.

3 The message is tailored to the application, and it is also sent as part of the SQL SIGNAL statement. The ability to force exceptions and send tailored messages is described in 2.6, “Error handling” on page 29.

Modifying data by using extended indicators

The department table contains five columns, one of which is defined as a unique key. The DBE created four data access modules to perform the functions of create, update, reset, and delete. The Human Resources system contains around five core tables, each one requiring a maintenance application. It seems as though this situation might result in many transaction modules. The DBE needs to find a way to minimize the amount of coding effort.

You can create a single module to handle create, update, delete, and reset transactions by using *extended indicators*. Extended indicators extend the use of SQL indicators, on output operations, from 0 (not NULL) and -1 (NULL) to include -5 (use a table-defined default) and -7 (ignore this column).

Of most interest is the ability to use the indicator value of -7 to allow an update to bypass the column as though it was not a part of the UPDATE statement. Any value that exists in the local variable is ignored. You can use this support to write a generic procedure that allows the same set of SQL SET DESCRIPTOR statements to be used for both insert and update operations. A delete operation can also use same procedure. However, it needs a descriptor that contains the key column variables only.

SET OPTION EXTIND = *YES must be used in the insert or update procedure to enable the use of extended indicators. Figure 8-4 contains an overview of how extended indicators are with modular procedures.

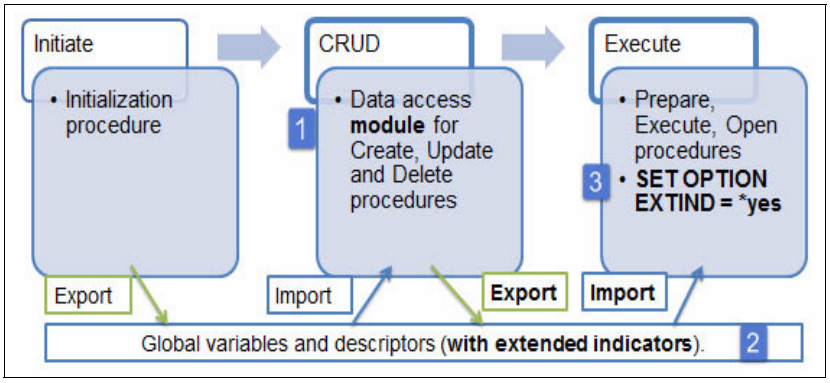

Figure 8-4 Overview of extended indicators with modular procedures

The following steps explain the highlighted areas within Figure 8-4 in more detail. The numbers correspond to the numbers in Figure 8-4:

1. This diagram represents a single module that is used for insert, update, and delete transactions. This module populates a global descriptor with the data values and indicator settings. Then, it exports the global descriptor before it calls the execute module.
2. The global descriptor is imported into the execute module.
3. The execute module is created with SET OPTION EXTIND = *YES. This procedure is the only procedure that needs this option.

Proof of concept

Example 8-16 contains the code for the proof of concept procedure.

Example 8-16 The extended_indicators_pot code

```
CREATE OR REPLACE PROCEDURE extended_indicators_pot (
   IN p_empno CHAR(6),
   IN p_bonus DECIMAL(9,2),
   IN p_bo_ind INTEGER,                                          --1
```

```
   IN p_edlevel smallint,
   IN p_el_ind INTEGER                                                    --2
   )
   LANGUAGE SQL
   SPECIFIC EXTNDINDP
   COMMIT ON RETURN YES
   SET OPTION EXTIND = *YES                                               --3

P1: BEGIN
   DECLARE sql_string CLOB (2M)                                           --4
      DEFAULT 'UPDATE employee SET (edlevel, bonus) = (?,?) WHERE empno = ?';

   ALLOCATE SQL DESCRIPTOR  'D1';                                         --5

   PREPARE S1 FROM sql_string ;                                           --6
   DESCRIBE INPUT S1 USING SQL DESCRIPTOR  'D1';                          --7

   SET SQL DESCRIPTOR  'D1' VALUE 1                                       --8
      TYPE = 5, DATA = p_edlevel , INDICATOR = p_el_ind;

   SET SQL DESCRIPTOR  'D1'  VALUE 2                                      --9
      TYPE = 3, LENGTH = 5, PRECISION = 9, SCALE = 2,
      DATA = p_bonus ,  INDICATOR =  p_bo_ind;

   SET SQL DESCRIPTOR  'D1'  VALUE 3                                      --10
      TYPE = 1, LENGTH = 6,
      DATA = p_empno ,  INDICATOR = 0 ;

   EXECUTE S1 USING SQL DESCRIPTOR  'D1';                                 --11

END P1
```

Notes about Example 8-16 on page 254:

1 This line shows the SQL indicator variable for p_bonus.

2 This line shows the SQL indicator variable for p_edlevel.

3 The SET OPTION EXTIND = *YES option is required to allow the use of extended indicator insert or update statements.

4 The SQL UPDATE statement string is defined. It contains all columns whether they are updated or not.

5 A local SQL descriptor (D1) is allocated for this procedure.[a]

6 The SQL statement string is prepared for execution.

7 The prepared statement is described into descriptor D1. The descriptor header COUNT will be set to 3, which is the total number of parameter markers.

8 The first descriptor variable represents the bonus column. The SQL SET DESCRIPTOR statement is used to assign the value for bonus and the associated SQL indicator.

9 The second descriptor variable represents the edlevel column. The SQL SET DESCRIPTOR statement is used to assign the value for edlevel and the associated SQL indicator.

10 The third descriptor variable represents the empno column, which is the unique key for the employee table. The SQL SET DESCRIPTOR statement is used to assign the value for empno and an INDICATOR value of 0 because this column will always contain a non-null value indicator.

11 The SQL EXECUTE statement runs the prepared SQL UPDATE statement to replace the parameter markers with the values from the descriptor. The indicator values in the descriptor are always used with or without the SET OPTION EXTIND = *YES. However, when the option is *YES, the indicator values of -5 and -7 are honored.

a. Short names (D1, S1, and so on) are used in this procedure for simplicity only. This type of name usage is not a preferred practice, and it must not be used for actual procedure development.

Example 8-17 contains the test script for the procedure in Example 8-16 on page 254.

Example 8-17 Test script with results

```
-- Change bonus and edlevel
CALL extended_indicators_pot('650301', 2300,0, 23,0);                    --1
SELECT empno, edlevel, bonus, row change timestamp for e AS WHEN_CHANGED
   FROM employee e
   WHERE empno = '650301';
--Results
-- EMPNO   EDLEVEL BONUS   WHEN_CHANGED
-- ------  ------- ------- --------------------------
-- 650301       23 2300.00 2015-11-24 07:29:09.476217                    --1

-- Change bonus,do not change edlevel
CALL extended_indicators_pot('650301', 0,0, 0,-7);                       --2
SELECT empno, edlevel, bonus, row change timestamp for e AS WHEN_CHANGED
   FROM employee e
   WHERE empno = '650301';
--Results
-- EMPNO  EDLEVEL BONUS WHEN_CHANGED
-- ------ ------- ----- --------------------------
```

```
-- 650301      23 0.00  2015-11-24 07:29:33.181552                          --2

-- Set edlevel to default (0)
CALL extended_indicators_pot('650301', 0,0, 23,-5);                         --3
SELECT empno, edlevel, bonus, row change timestamp for e AS WHEN_CHANGED
   FROM employee e
   WHERE empno = '650301';
-- EMPNO  EDLEVEL BONUS WHEN_CHANGED
-- ------ ------- ----- --------------------------
-- 650301      0  0.00  2015-11-24 07:32:07.580737                          --3

-- Set bonus to NULL
CALL extended_indicators_pot('650301', 0,-1, 23,-7);                        --4
SELECT empno, edlevel, bonus, row change timestamp for e AS WHEN_CHANGED
   FROM employee e
   WHERE empno = '650301';
-- EMPNO  EDLEVEL BONUS WHEN_CHANGED
-- ------ ------- ----- --------------------------
-- 650301      0  NULL  2015-11-24 07:51:53.189613                          --4
```

Notes about Example 8-17 on page 256:

Four tests were performed. Each numbered note describes the type of test and results:

1 This test validated the use of the traditional SQL indicator 0. Both bonus and edlevel were changed as a result of the test.

2 This test validated the use of SQL indicator -7. A data value of 0 was passed for both columns. The SQL indicator for bonus was 0, which allowed the column to be changed. The SQL indicator for edlevel is -7, which informs DB2 to ignore the passed data value of 0 and to not change the existing value of edlevel. The result of the test was successful.

3 This test validated the use of SQL indicator -5 to set the value of edlevel to the table-defined default of 0 for a numeric column. The test was successful.

4 This test validated the use of the traditional SQL indicator -1 to set the value of bonus to NULL. The column bonus is nullable. The test was successful.

Modify_Department_Transaction

The modify_department_transaction was developed by the DBE as single module to process all inserts, updates, and deletes for a single table. It uses many of the concepts that are described in this chapter and throughout this book. This module includes the following concepts:

- Extended indicators
- Array types
- Global variables
- Global descriptors
- Reusable procedures

Example 8-18 contains the code for the modify_department_transaction.

Example 8-18 Modify_Department_Transaction

```
CREATE OR REPLACE PROCEDURE modify_department_transaction (
   -- Required parameters
   IN p_trns_type SMALLINT,                                                 --1
   IN p_deptno CHAR(3),
```

```
   -- Optional parameters                                                    --2
   IN p_when_Changed TIMESTAMP DEFAULT NULL,
   IN p_deptname VARCHAR(36) DEFAULT NULL,
   IN p_mgrno CHAR(6) DEFAULT '*NOCHG',
   IN p_location CHAR(16) DEFAULT '*NOCHANGE',
   IN p_admrdept CHAR(3) DEFAULT NULL
   RESULT SETS 1
   LANGUAGE SQL
   COMMIT ON RETURN YES
   SPECIFIC MODDPTTRNS

P1: BEGIN
   --Declare local v_sql_strings
    DECLARE v_global_Descriptor VARCHAR(128) ;
    --Array v_sql_strings                                                   --3
    DECLARE i, ind INTEGER;
    DECLARE v_ind_ary smallint_array;

    DECLARE v_sql_string CLOB(2M) DEFAULT NULL;

   --Procedure logic begins here
     SET v_ind_ary = ARRAY[-7,-7,-7,-7,0];                                 --4

     CASE p_trns_type                                                      --5
     WHEN 1 THEN
        -- Must be INSERT
        SET v_Global_Descriptor = 'ADDDPTTRNS';
      SET v_sql_string =  'INSERT INTO department
                           (deptname, mgrno,location, admrdept, deptno )
                           VALUES (?, ?, ?, ?, ?)';
        -- All parameters will be used                                      --6
        SET v_ind_ary = ARRAY[0,0,0,0,0]; --5 set indicators to assign all values
     WHEN 2 THEN
        -- Must be UPDATE
        SET v_Global_Descriptor = 'UPDDPTTRNS';
        SET v_sql_string = 'UPDATE department_change
                           SET (deptname, mgrno, location, admrdept) =
                           (?,?,?,?) WHERE deptno = ?';
        -- Set the indicators accordingly                                   --7
        IF p_deptname IS NOT NULL THEN
           SET v_ind_ary[1] = 0;
        END IF;
        IF p_mgrno <> '*NOCHG' THEN                                         --8
          IF p_mgrno IS NULL THEN
           SET v_ind_ary[2] = -1;
          Else
           SET v_ind_ary[2] = 0;
          END IF;
        END IF;
        IF p_location <> '*NOCHANGE' THEN
          IF p_location IS NULL THEN
           SET v_ind_ary[2] = -1;
          ELSE
           SET v_ind_ary[3] = 0;
          END IF;              END IF;
```

```
      IF p_admrdept IS NOT NULL THEN
         SET v_ind_ary[4] = 0;
      END IF;
   WHEN 3 THEN
      -- Must be UPDATE...RESET
      SET v_Global_Descriptor = 'RSTDPTTRNS';
      SET v_sql_string = 'UPDATE department_change
                          SET (deptname, mgrno, location, admrdept) =
                          (?,?,?,?) WHERE deptno = ?';
      IF p_mgrno = '*RESET' THEN
         SET v_ind_ary[2] = -5;
      END IF;
      IF p_location = '*RESET' THEN
         SET v_ind_ary[3] = -5;
      END IF;
   WHEN 4 THEN
      -- Must be DELETE
      SET v_Global_Descriptor = 'DLTDPTTRNS';
      SET v_sql_string = 'DELETE department_change WHERE deptno = ?';
      SET v_ind_ary[1] = 0;
   ELSE
      SIGNAL SQLSTATE '70030'
      SET MESSAGE_TEXT = 'Invalid transaction type';
END CASE;

CALL allocate_global_descriptor(v_Global_Descriptor);

-- Export global variables
 SET gv_sql_string = v_sql_string;
 SET gv_ROW_CHANGE_TIMESTAMP = p_when_Changed;

CALL describe_input_descriptor();
-- Set the global descriptor values                                    --9
SET i = 1;
if p_trns_type <> 4 then
   -- Updateable columns
   SET ind = v_ind_ary[i] ;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE i
       TYPE = 12, LENGTH = 36, DATA = p_deptname , INDICATOR = ind ;
   SET i = i + 1;
   SET ind = v_ind_ary[i] ;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE i
      TYPE = 1, LENGTH = 6, DATA = p_mgrno , INDICATOR = ind ;
   SET i = i + 1;
   SET ind = v_ind_ary[i] ;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE i
      TYPE = 1, LENGTH = 16, DATA = p_location , INDICATOR = ind  ;
   SET i = i + 1;
   SET ind = v_ind_ary[i] ;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE i
      TYPE = 1, LENGTH = 3, DATA = p_admrdept, INDICATOR = ind ;
   SET i = i + 1;
END IF;

-- Key value follows updateable columns                                --10
```

```
   SET ind = v_ind_ary[i] ;
   SET SQL DESCRIPTOR GLOBAL v_global_descriptor VALUE i
      TYPE = 1, LENGTH = 3, DATA = p_deptno, INDICATOR = ind ;

   CALL execute_transaction();                                          --11

END P1
```

The following notes describe Example 8-18 on page 257. The highlighted numbers correspond to the highlighted numbers in Example 8-18 on page 257:

1 A new parameter p_trnstype is required. The value is used to determine the type of transaction:

1 = Create (INSERT)
2 = Change (UPDATE)
3 = Reset (UPDATE)
4 = Delete

2 The row change timestamp is optional with a default of NULL. It is not required for inserts. However, it is required for all other transaction types. The additional parameters are optional (required for INSERT) and certain parameters cannot use NULL as a default because several database columns can contain the NULL value. The following convention for default values was used:

- If the column is not null-capable, use NULL. Otherwise, consider character parameters - *NOCHANGE, which is abbreviated if the parameter is fewer than nine characters, for example, *NOCHG.
- Date/time parameters. Use zeros for single day, month, or time parameters. Use an unrealistic date for date or timestamp parameters, for example, '01-01-0001' or '12-31-9999'.
- Numerics. Use a value that is not valid for the data type, for example, all 9s or negative values. Consider the use of check constraints at the physical database layer to avoid accidental changes. For more information about check constraints, see DB2 for i SQL reference, which is at this website:

  https://ibm.biz/Bd42dh

3 The procedure uses an array of integers to contain the SQL indicators. The following CREATE TYPE DDL statement is used to create the indicator array:

```
CREATE TYPE indicator_array AS SMALLINT ARRAY[100]
```

- A hundred indicators are enough for most well-normalized tables. For information about the maximum number of parameters that are allowed within a procedure, see DB2 for i SQL reference, which is at this website:

  https://ibm.biz/Bd42dh

4 The elements in the indicator array are initialized -7 (ignore) except for the indicators that correspond to the key columns.

5 A simple CASE statement is used to process the p_trnstype parameter. If the value is 1, 2, 3, or 4, assign the appropriate indicator values for the transaction. Any other value will signal an exception.

6 For insert operations, each element of the indicator array is set to zero. The data value of each parameter is assigned to the corresponding descriptor variable.

7 For update operations, the SQL indicator is set to zero if the parameter value is not the “do not change” value.

8 For null-capable columns, two tests need to be conducted. The first test determines whether the parameter value is the “do not change” value. If true, do nothing because the current SQL indicator value is -7. If false, test for null. If true, set the SQL indicator to -1. Otherwise, set the SQL indicator to zero.

9 A single descriptor is used for all transaction types. The first step is to set the SQL indicator array index (i) to 1. The following steps are performed for each descriptor variable:

a. Assign the SQL indicator based on the index to the variable ind.

b. By using the SQL SET DESCRIPTOR statement, assign the DATA and INDICATOR values to the descriptor item where VALUE = i.

c. Increment i by 1.

d. Repeat for all non-key parameters.

10 The same process that is described in 9 is used for the key column deptno. The SQL indicator value was part of the array initialization that occurred at the beginning of the procedure logic.

11 The SQL EXECUTE statement runs the prepared statement, which replaces the parameter markers with the values from the descriptor.

Example 8-19 contains the workflow for the modify_department_transaction. The results are the same as Example 8-17 on page 256, except that a single SQL procedure is used.

Example 8-19 A sample workflow

```
--Add new department
CALL modify_department_transaction (1, 'XXX' , 'PL/1 Application Development' ,
'650311' , 'Rochester',  'D21' );

--Retrieve the newly created department
call get_department_data_for_update('XXX',' ', ' ', ' ', ' ');
Return Code = 0
Output Parameter #2 = PL/1 Application Development
Output Parameter #3 = 650311
Output Parameter #4 = Rochester
Output Parameter #5 = 2015-12-03 14:59:36.969184

--Uh oh, funding for the new department no longer exists, delete the department
-- using the department number and row change timestamp from above
call modify_department_transaction(4,'XXX',TIMESTAMP('2015-12-03 14:59:36.969184')
;
--Create a department that is going somewhere
CALL modify_department_transaction (1,'COE' , 'DB2 for i Center of Excellence' ,
'650311' , 'Rochester',  'D21' );
--Retrieve the department information
call get_department_data_for_update('COE',' ', ' ', ' ', ' ');
Return Code = 0
Output Parameter #2 = DB2 for i Center of Excellence
Output Parameter #3 = 650311
Output Parameter #4 = Rochester
Output Parameter #5 = 2015-11-17 15:07:11.849795
--Change the location to be more specific
CALL modify_department_transaction
   (2, 'COE' , TIMESTAMP('2015-11-17 15:07:11.849795'),
   DEFAULT, DEFAULT, P_location=> 'Roch Lab Svcs   ');
-- Validate the change
```

```
call get_department_data_for_update('COE',' ', ' ', ' ', ' ');
Return Code = 0
Output Parameter #2 = DB2 for i Center of Excellence
Output Parameter #3 = 650311
Output Parameter #4 = Roch Lab Svcs
Output Parameter #5 = 2015-12-03 15:14:56.820248
-- Cannot agree on the new name so reset it to the default
CALL modify_department_transaction (
   3,'COE' , TIMESTAMP('2015-12-03 15:14:56.820248 ',P_location=> '*RESET'));
--Discount double check
call get_department_data_for_update('COE',' ', ' ', ' ', ' ');
Return Code = 0
Output Parameter #2 = DB2 for i Center of Excellence
Output Parameter #3 = 650311
Output Parameter #4 =
Output Parameter #5 = 2015-12-03 16:16:03.258549
--Time to go home
```

8.3 Summary

This chapter describes how to develop flexible and reusable procedures that take advantage of the following capabilities:

- Global descriptors and global variables
- Procedure parameter omission and naming
- The import and export of data between procedures
- Single procedures that perform all add, update, delete, and read operations
- Hidden row change timestamp columns and optimistic locking

In the real world, most heritage databases are not well-normalized and the rows tend to be wide. Or, they might be highly normalized, and a single entity might be made up of several one-to-one or one-to-maybe-one dependencies. Likely, many programs are in place that update pieces of the table or entity.

An example of these programs is an application that is responsible for entering new employees, another application that is responsible for updating bonuses and commissions, and another application that is responsible for changing phone numbers. In any case, the application must read the input data from a window, mobile device, or transaction file, assign the values from the input device to the procedure variables, and issue the insert or update. In many cases, updates are performed to the entire row, whether the data changed or not.

The approach that is presented in this chapter is meant to reduce the amount of repetitive SQL code that must be written in each procedure that manipulates the database data. By using global variables and flexible descriptors, an application manipulates the data that is required for the transaction only, which reduces redundant code and also minimizes the impact of change when you alter existing database table objects.

The execute_transaction routine can handle any insert, update, or delete statement that is created dynamically, and it uses a flexible descriptor. The concept is the same for processing result sets. Each result set has a different format. One procedure can consume any result set by using a flexible descriptor. Code to handle errors can be isolated to these single procedures, eliminating the need to replicate this code in every procedure.

Flexible descriptors are not available to host-centric HLL applications by using traditional I/O methods. Therefore, multiple HLL programs are cloned from templates or contain INCLUDE statements to bring in the code at compile time. A change to an INCLUDE requires the re-creation, and subsequent testing, of all modules that reference the INCLUDE. This change adds significantly to the cost of maintenance, adds to the total time to market, and perhaps costs even more money in terms of lost revenue.

Application programmers that are familiar with the concept of Integrated Language Environment (ILE) understand the benefit of creating small, reusable service programs. The concepts in this chapter are no different. Global variables and descriptors take advantage of the ILE concepts of variable export and import.

The emerging role of the DBE creates a growing need for skilled, data-centric developers. The ability to fill these positions from within, retaining both application and business knowledge, is the ideal solution for many development shops. A skilled ILE application programmer, who is not interested in graphical UI development, might be the best candidate to become a DBE.

9

IBM i and IBM DB2 for i services

To facilitate the use of Structured Query Language (SQL) for the development of modern and portable applications, IBM uses SQL programming concepts to provide SQL programmers with various services, including both database and operating system functions. These services offer the following advantages:

- They can be used as building blocks for additional SQL programming.
- They can reduce the need for control language (CL) programming by allowing access to services that were only available by using CL commands.
- They can reduce the need for writing high-level language (HLL) programs to manage cumbersome calls to application programming interfaces (APIs) that are provided by IBM.

These services are called “DB2 for i Services” and “IBM i Services”.

DB2 for i Services are documented in the IBM Knowledge Center at the following website:

`http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/rzajq/rzajqservicesdb2.htm`

IBM i Services are documented in the IBM Knowledge Center at the following website:

`http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/rzajq/rzajqservicessys.htm`

The DB2 for i Wiki can provide additional services documentation. For more information, see this website:

`http://ibm.biz/DB2foriServices`

Most of these services are available in both IBM i 7.1 and IBM i 7.2. Many services are implemented as part of a Technology Refresh (TR), which means that they are supported after the installation of a certain program temporary fix (PTF). Specific services are available in IBM i 7.2 only.

IBM provides these services by using procedures, user-defined functions (UDFs), and views, which sometimes conceal the use of user-defined table functions (UDTFs. They are shipped in either the `QSYS2` or `SYSTOOLS` library. Library `QSYS2` contains the services that are considered part of the operating system, and they provide upward compatibility.

Library SYSTOOLS is used by IBM to provide useful functionalities and examples about how to use the power of SQL. By examining the underlying SQL code, you can gain insight about the techniques that are used and get inspiration for further development. The examples in SYSTOOLS are provided “as is” with no guarantee of upward compatibility.

DB2 for Services cover various areas:

- Application
- Performance
- Plan cache
- Utility

Services were developed as part of the DB2 for i Services for the database health monitoring area. They are known as *Health Center procedures*. You can use these procedures through normal SQL calls or through the graphical user interface (GUI) that is provided in the IBM i Navigator Database Health Center (client/server and web).

The IBM i Services cover various functional areas in the operating system:

- Application
- Java
- Journal
- Librarian
- Message handling
- Product
- PTF
- Security
- Spool
- Storage
- System health
- Communications
- Work management

During the time of writing this book, a total of 82 services were available.

Important: This number is certain to increase so we recommend that you check the IBM i Knowledge Center and the DB2 for i Wiki for updated information.

In general, DB2 for i Services are targeted for consumption by database engineers (DBEs). IBM i Services can be used by anyone who is interested in monitoring and managing the system. The use cases are numerous and an easy way to interact with system functionalities that use SQL.

This chapter includes the following topics:

- Health Center procedures
- Utility procedures
- Plan cache procedures
- DB Application Services
- Performance Services
- PTF Services
- Security Services
- Message Handling Services
- Librarian Services
- Work Management Services
- Java Services
- IBM i Application Services

9.1 Health Center procedures

This section describes a subset of the services with particular attention to the services that are valuable to the SQL programmer and that serve as good examples of SQL programming. This group of services consists of six items. Table 9-1 shows their availability by release.

Table 9-1 IBM DB2 for i Services: Health Center procedures

Health Center procedure	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.HEALTH_ACTIVITY	External procedure	Base	SF99701 Level 24
QSYS2.HEALTH_DATABASE_OVERVIEW	External procedure	Base	SF99701 Level 24
QSYS2.HEALTH_DESIGN_LIMITS	External procedure	Base	SF99701 Level 24
QSYS2.HEALTH_ENVIRONMENTAL_LIMITS	External procedure	Base	SF99701 Level 24
QSYS2.HEALTH_SIZE_LIMITS	External procedure	Base	SF99701 Level 24
QSYS2.RESET_ENVIRONMENTAL_LIMITS	External procedure	Base	SF99701 Level 24

These procedures provide an easy way to use SQL for database health activities that were originally performed only through the IBM i Navigator GUI interface. These procedures provide a point in time insight into the consumption of DB2 for i resources. By capturing this detail on a regular cadence, you can gain unique insight into the data center, a good understanding of the current situation, and a perspective about database object evolution over time. When data is saved, it can be compared.

- Are new objects created or are existing objects deleted?
- Are my objects growing in size?
- Considering my growth trend, is my database approaching its limits?
- How many deleted rows are in my tables?
- What is the percent of deleted rows versus valid rows?
- Which are my top 10 jobs?

Consider the following ideas about usage:

- The QSYS2.Health_Database_Overview procedure counts all of the different types of DB2 for i objects within the target schema or schemas and separates them by object type and subtype. Saving this information during the procedure allows the DBE to track changes in database.
- QSYS2.Health _Activity returns summary counts of database and SQL operations over a set of objects within one or more schemas. If you want to understand (or track) how many times a table is opened, closed, or cleared, this procedure is for you. If you need to monitor how many insert, delete, update, and read operations are performed, this procedure is for you.
- QSYS2.Health_Environmental_Limits returns detail about the top jobs on the system, for different SQL or application limits. The jobs do not need to exist because the “top 10” information is maintained within DB2 for i. The information is reset when you IPL the machine, the independent auxiliary storage pool (IASP) is varied ON, or when the QSYS2.Reset_Environmental_Limits() procedure is called.

Because all of these procedures are based on external procedures, we do not describe them in detail. For more information, see this website:

https://ibm.biz/Bd4fQy

9.2 Utility procedures

This group of services consists of nine items. Table 9-2 shows their availability by release.

Table 9-2 IBM DB2 for i Services: Utility procedures

Utility procedure	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.CANCEL_SQL	External procedure	Base	SF99701 Level 4
QSYS2.DUMP_SQL_CURSORS	External procedure	Base	SF99701 Level 6
QSYS2.EXTRACT_STATEMENTS	SQL procedure	Base	SF99701 Level 28
QSYS2.FIND_AND_CANCEL_QSQSRVR_SQL	SQL procedure	Base	SF99701 Level 4
QSYS2.FIND_QSQSRVR_JOBS	External procedure	Base	SF99701 Level 3
QSYS2.GENERATE_SQL	External procedure	Base	SF99701 Level 29
QSYS2.RESTART_IDENTITY	SQL procedure	Base	SF99701 Level 28
SYSTOOLS.CHECK_SYSCST	SQL procedure	Base	SF99701 Level 21
SYSTOOLS.CHECK_SYSROUTINE	SQL procedure	Base	SF99701 Level 21

In this document, we chose to document QSYS2.EXTRACT_STATEMENTS.

9.2.1 QSYS2.EXTRACT_STATEMENTS

The QSYS2.EXTRACT_STATEMENTS procedure is an easy way of partition an SQL Plan Cache snapshot. Assume that a critical peak of activity occurs in your shop and you determined that it relates to database access that is performed by a specific user profile. Due to the nature of your application environment, the workload can relate to various interfaces and various jobs. The main common identifier that you can use is the job's current user profile.

Your system might be set to programmatically dump the whole SQL Plan Cache when a certain level of CPU usage is reached by using the QSYS2.DUMP_PLAN_CACHE procedure.

Note: *JOBCTL special authority or QIBM_DB_SQLADM function usage is required to use the QSYS2.DUMP_PLAN_CACHE procedure.

When you are ready to perform your analysis, you might want to know the SQL statements that are the most expensive (longest lasting) with SIMONA as the current user.

The basic answer set of this procedure call includes start time and SQL statement text only. However, you can customize the use of the ADDITIONAL_SELECT_COLUMNS parameter easily. In a similar manner, you can specify any column that is in the base plan cache snapshot table as a filter by using the **ADDITIONAL_PREDICATES** parameter.

In Example 9-1, we query a plan cache snapshot. We add the Total_Time and Current_User_Profile columns to the data set. We filter to get only the rows where the query took longer than 1 second and the current user profile was SIMONA.

Example 9-1 Usage of QSYS2.EXTRACT_STATEMENTS procedure

```
CALL QSYS2.EXTRACT_STATEMENTS('SNAPSHOTS', 'MYDUMP',
ADDITIONAL_SELECT_COLUMNS => 'DEC(QQI6)/1000000.0 as Total_time,
                              QVC102 as Current_User_Profile ',
ADDITIONAL_PREDICATES => ' AND QQI6 > 1000000 and QVC102 = ''SIMONA''  ',
ORDER_BY => ' ORDER BY QQI6 DESC ');
```

You receive a list of statements that are good candidates for optimization. This list will be the basis for subsequent investigation by using the appropriate tools.

Note: EXTRACT_STATEMENTS reconstruct the complete query, substituting host variable and parameter marker values with the corresponding literal value.

For detailed information about tuning techniques and database monitor SQL table format, see “Database Performance and Query Optimization” at this website:

https://ibm.biz/Bd42Jk

9.3 Plan cache procedures

At the time that this book was written, this group of services consisted of nine items. Table 9-3 shows their availability by release.

Table 9-3 IBM DB2 for i Services: Plan cache procedures

Plan cache	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.CHANGE_PLAN_CACHE_SIZE	SQL procedure	Base	SF99701 Level 9
QSYS2.DUMP_PLAN_CACHE	SQL procedure	Base	Enhanced SF99701 Level 27
QSYS2.DUMP_PLAN_CACHE_PROPERTIES	SQL procedure	Base	Enhanced SF99701 Level 27
QSYS2.DUMP_PLAN_CACHE_topN	SQL procedure	Base	SF99701 Level 29
QSYS2.DUMP_SNAP_SHOT_PROPERTIES	SQL procedure	Base	SF99701 Level 27
QSYS2.END_ALL_PLAN_CACHE_EVENT_MONITORS	SQL procedure	Base	SF99701 Level 24
QSYS2.END_PLAN_CACHE_EVENT_MONITOR	SQL procedure	Base	SF99701 Level 24
QSYS2.START_PLAN_CACHE_EVENT_MONITOR	SQL procedure	Base	SF99701 Level 24
QSYS2.CLEAR_PLAN_CACHE	SQL procedure	SF99702 Level 5	SF99701 Level 34

We document the QSYS2.DUMP_PLAN_CACHE_topN procedure only, although you might be interested in understanding more about this whole set of services by researching it in the IBM i Knowledge Center.

9.3.1 QSYS2.DUMP_PLAN_CACHE_topN

This procedure creates a snapshot file from the active plan cache that contains only those queries with the largest accumulated elapsed time. The number of queries to capture is designated by the caller in the third parameter. This procedure provides a programmatic way to capture the most time-consuming queries, making it easier to compare and contrast this aspect of database performance.

Example 9-2 shows the call that was used to capture the 20 queries with the largest elapsed time and dump the details into a snapshot file that is named SNAPSHOTS/TOPN20_A.

Example 9-2 QSYS2.DUMP_PLAN_CACHE_topN usage

```
CALL QSYS2.DUMP_PLAN_CACHE_topN('SNAPSHOTS', 'TOPN20_A', 20);
```

9.4 DB Application Services

Three services are available in the DB2 for i Application Services area. Table 9-4 shows their availability by release.

Table 9-4 IBM DB2 for i Services: Application Services

Application Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.OVERRIDE_TABLE	SQL procedure	Base	SF99701 Level 24
QSYS2.DELIMIT_NAME	External UDF	Base	SF99701 Level 29
QSYS2.WLM_SET_CLIENT_INFO	External procedure	Base	Base

9.5 Performance Services

Seven services are available in the DB2 for i Performance Services area. Table 9-5 shows their availability by release.

Table 9-5 IBM DB2 for i Services: Performance Services

Performance Services	Type of service	IBM i 7.2	IBM i 7.1
SYSTOOLS.ACT_ON_INDEX_ADVICE	SQL procedure	Base	Base
SYSTOOLS.HARVEST_INDEX_ADVICE	SQL procedure	Base	Base
QSYS2.OVERRIDE_QAQQINI PROCEDURE	External procedure	Base	Base
QSYS2.RESET_TABLE_INDEX_STATISTICS	SQL procedure	Base	Base
QSYS2.SYSIXADV	Table	Base	Base
SYSTOOLS.REMOVE_INDEXES	SQL procedure	Base	Base
QSYS2.DATABASE_MONITOR_INFO	External UDF	SF99702 Level 5	SF99701 Level 34

9.6 PTF Services

This group of services consists of four items. Table 9-6 shows their availability by release.

Table 9-6 IBM i Services: PTF Services

PTF Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.PTF_INFO	View	Base	SF99701 Level 23
QSYS2.GROUP_PTF_INFO	View	Base	SF99701 Level 6
SYSTOOLS.GROUP_PTF_CURRENCY	View	SF99702 Level 3	SF99701 Level 32
SYSTOOLS.GROUP_PTF_DETAILS	UDTF	SF99702 Level 9	SF99701 Level 37

The names of the PTF Services indicate their functions:

- QSYS2.PTF_INFO is used to retrieve information about PTFs in the system and their status as provided by the List Program Temporary Fixes (QpzListPTF) API.
- QSYS2.GROUP_PTF_INFO provides information about group PTFs. It is based on the List PTF Groups (QpzListPtfGroups) API.
- SYSTOOLS.GROUP_PTF_CURRENCY compares the level of the installed PTF groups with the current level that is available on the IBM Fix Central website and indicates whether your environment is at an earlier level.
- SYSTOOLS.GROUP_PTF_DETAILS provides details about all PTFs that are in a group. It compares the installed PTF's level with the level that is available on the IBM Fix Central website and indicates whether your environment is at an earlier level.

9.6.1 GROUP_PTF_CURRENCY

This service is useful. It provides an example of the power of modern SQL in action. It is implemented as a view by using the XMLTable() and HTTPGETBLOB() table functions that access XML data from the IBM Preventive Service Planning website:

https://ibm.biz/Bd4fTd

The service parses the content and presents it as a virtual table that is compared with the view GROUP_PTF_INFO. (GROUP_PTF_INFO is a view that conceals the use of a user-defined table function (UDTF).) Because it is implemented as a view, the query that accesses it can be modified to provide the results that you want. In this case, the service retrieves only the list of groups that need to be updated, as shown in Example 9-3.

Example 9-3 GROUP_PTF_CURRENCY usage

```
SELECT ptf_group_currency, ptf_group_id, ptf_group_title, ptf_group_level_installed,
   ptf_group_level_available, ptf_group_last_updated_by_ibm
   FROM systools.group_ptf_currency
   WHERE ptf_group_release = 'R720' and ptf_group_currency='UPDATE AVAILABLE'
   ORDER BY ptf_group_level_available - ptf_group_level_installed DESC;
```

This statement produces a result that is similar to Figure 9-1.

PTF_GROUP_CURRENCY	PTF_GROUP_ID	PTF_GROUP_TITLE	PTF_GROUP_LEVEL_INSTALL...	PTF_GROUP_LEVEL_AVAILA...	PTF_GROUP_LAST_UPDATED_BY_IBM
UPDATE AVAILABLE	SF99720	Current Cumulative PTF Media Documentation	14276	15135	05/29/2015
UPDATE AVAILABLE	SF99719	720 Group Hiper	31	44	09/08/2015
UPDATE AVAILABLE	SF99718	720 Group Security	14	19	09/08/2015
UPDATE AVAILABLE	SF99702	720 DB2 for IBM i	4	8	08/21/2015
UPDATE AVAILABLE	SF99713	720 IBM HTTP Server for i	7	10	08/26/2015
UPDATE AVAILABLE	SF99715	720 Backup Recovery Solutions	11	14	08/25/2015
UPDATE AVAILABLE	SF99775	720 Hardware and Related PTFs	7	10	08/17/2015
UPDATE AVAILABLE	SF99716	720 Java	5	7	08/28/2015
UPDATE AVAILABLE	SF99480	720 WebSphere App Server V8.0	3	4	08/17/2015
UPDATE AVAILABLE	SF99481	720 WebSphere App Server V8.5	4	5	06/26/2015
UPDATE AVAILABLE	SF99717	720 Technology Refresh	1	2	05/29/2015
UPDATE AVAILABLE	SF99776	720 High Availability for IBM i	1	2	06/11/2015

Figure 9-1 GROUP_PTF_CURRENCY that is used

If you want to retrieve its source, you can use another useful system procedure that is called QSYS2.GENERATE_SQL() to retrieve the view definition (Example 9-4). GENERATE_SQL is called with the appropriate parameters to show the source.

Example 9-4 Retrieving GROUP_PTF_CURRENCY source

```
CALL QSYS2.GENERATE_SQL('GROUP_PTF_CURRENCY', 'SYSTOOLS', 'VIEW', REPLACE_OPTION => '0');
```

For more information, see the IBM DB2 for i Wiki:

https://ibm.biz/Bd4fTH

After you execute the previous call, it displays the source code for the view, as shown in Example 9-5.

Example 9-5 SYSTOOLS.GROUP_PTF_CURRENCY source code

```
SELECT CASE WHEN ACTUAL.GRPPTF IS NULL THEN 'PTF GROUP DOES NOT EXIST ON ' CONCAT CURRENT
SERVER WHEN PSPS.PSP_NUMBER IS NULL THEN 'PSP INFORMATION NOT AVAILABLE' WHEN ACTUAL.GRPPTF
= PSPS.PSP_NUMBER AND ACTUAL.PTF_GROUP_LEVEL = PSPS.PSP_LEVEL THEN 'INSTALLED LEVEL IS
CURRENT' WHEN ACTUAL.GRPPTF = PSPS.PSP_NUMBER AND
ACTUAL.PTF_GROUP_LEVEL < PSPS.PSP_LEVEL THEN 'UPDATE AVAILABLE' WHEN ACTUAL.GRPPTF =
PSPS.PSP_NUMBER AND ACTUAL.PTF_GROUP_LEVEL > PSPS.PSP_LEVEL THEN 'PSP IS DOWNLEVEL -  '
CONCAT ACTUAL.PTF_GROUP_STATUS END PTF_GROUP_CURRENCY, COALESCE(PSPS.PSP_NUMBER,
ACTUAL.GRPPTF) PTF_GROUP_ID, COALESCE(PSPS.PSP_TITLE, ACTUAL.PTF_GROUP_DESCRIPTION)
PTF_GROUP_TITLE, ACTUAL.PTF_GROUP_LEVEL PTF_GROUP_LEVEL_INSTALLED, PSPS.PSP_LEVEL
PTF_GROUP_LEVEL_AVAILABLE, PSPS.PSP_DATE AS PTF_GROUP_LAST_UPDATED_BY_IBM,
COALESCE(PSPS.PSP_RELEASE, ACTUAL.PTF_GROUP_TARGET_RELEASE) PTF_GROUP_RELEASE,
ACTUAL.PTF_GROUP_STATUS PTF_GROUP_STATUS_ON_SYSTEM FROM XMLTABLE('/all_psps/psp' PASSING
XMLPARSE(DOCUMENT
SYSTOOLS.HTTPGETBLOB('http://www-912.ibm.com/s_dir/sline003.nsf/PSPbyNumL.xml?OpenView&coun
t=500' , '')) COLUMNS PSP_RELEASE CHAR(5) PATH 'release', PSP_NUMBER CHAR(7) PATH 'number',
PSP_TITLE VARCHAR(1000) PATH 'title', PSP_LEVEL INTEGER PATH 'level', PSP_DATE CHAR(10)
PATH 'date' ) PSPS RIGHT OUTER JOIN ( SELECT SUBSTR(PTF_GROUP_NAME, 1,7) AS GRPPTF,
PTF_GROUP_LEVEL, PTF_GROUP_STATUS, PTF_GROUP_DESCRIPTION, PTF_GROUP_TARGET_RELEASE FROM  (
SELECT PTF_GROUP_NAME,PTF_GROUP_LEVEL, PTF_GROUP_STATUS, PTF_GROUP_DESCRIPTION,
PTF_GROUP_TARGET_RELEASE,  RANK() OVER (PARTITION BY PTF_GROUP_NAME ORDER BY
PTF_GROUP_LEVEL DESC) AS INSTALLED_NUMBER  FROM QSYS2.GRPPTFINFO WHERE PTF_GROUP_STATUS =
'INSTALLED') A  WHERE A.INSTALLED_NUMBER = 1 ) ACTUAL ON
(ACTUAL.GRPPTF = PSPS.PSP_NUMBER);
```

You can see the power and versatility of IBM DB2 for i:

- SYSTOOLS.HTTPGETBLOB is used to retrieve PTF content from the IBM Support website.
- XMLPARSE parses it to ensure that it is syntactically correct XML.
- XMLTABLE is used to extract data from the XML document and present it as a virtual table that can be referenced like any other table in SQL.

With this view, the content that is available on the web in HTTP format is expressed as a virtual table and then processed. Use this view in your environment, and use the same techniques to solve other business problems.

9.7 Security Services

This group of services consists of seven items. Table 9-7 shows their availability by release.

Table 9-7 IBM i Services: Security Services

Security Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.USER_INFO	View	Base	Introduced: SF99701 Lev 26 Enhanced: SF99701 Lev 29
QSYS2.FUNCTION_INFO	View	Base	SF99701 Level 26
QSYS2.FUNCTION_USAGE	View	Base	SF99701 Level 26
QSYS2.GROUP_PROFILE_ENTRIES	View	Base	SF99701 Level 23
QSYS2.SQL_CHECK_AUTHORITY()	UDF	Base	SF99701 Level 21
QSYS2.SET_COLUMN_ATTRIBUTE()	Procedure	Base	Base
QSYS2.DRDA_AUTHENTICATION_ENTRY_INFO	View	SF99702 Level 5	SF99701 Level 34

To learn about these services and see examples of their usage, see the IBM i Knowledge Center or the DB2 for i Wiki. In this book, QSYS2.SQL_CHECK_AUTHORITY() was selected for more explanation because it is implemented as a user-defined function (UDF).

9.7.1 QSYS2.SQL_CHECK_AUTHORITY()

Use this scalar function to easily check whether the user is authorized to query a specific *FILE object.

The syntax of this function is shown:

```
>>-SQL_CHECK_AUTHORITY--(--library-name--,--file-name--)--><
```

The result of this function is a smallint. The returned value is one of the following values:

- 0: If the user does not have authority to query the file, the object is not a *FILE object, or the object does not exist.
- 1: If the user is authorized to query the file.

9.8 Message Handling Services

This group consists of two services. Table 9-8 shows their availability by release.

Table 9-8 IBM i Services: Message Handling Services

Message Handling Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.REPLY_LIST_INFO	View	SF99702 Level 3	SF99701 Level 32
QSYS2.JOBLOG_INFO	UDTF	SF99702 Level 3	SF99701 Level 32

To learn about these services and see examples of their usage, see the DB2 for i Wiki. In this book, QSYS2.JOBLOG_INFO was selected for further explanation because it is implemented as a user-defined table function (UDTF).

9.8.1 QSYS2.JOBLOG_INFO

With this UDTF, you can retrieve job log messages for any job that is active on the system. Each message in the job log becomes a row in the resulting table.

This DB2 for i Service is documented in the IBM DB2 for i Wiki at this website:

https://ibm.biz/Bd4fT8

Messages can be filtered on any column. In Example 9-6, the first-level and second-level error message text is presented. Only diagnostic (error) messages are collected.

Example 9-6 QSYS2.JOBLOG_INFO usage

```
SELECT message_text, message_second_level_text
   FROM table(QSYS2.JOBLOG_INFO('067197/simona/it024387a')) a
   WHERE a.message_type = 'DIAGNOSTIC'
   ORDER BY ordinal_position desc
```

Figure 9-2 shows the result set of the previous query.

MESSAGE_TEXT	MESSAGE_SECOND_LEVEL_TEXT
No authority to job 066332/QLWISVR/ADMIN3.	&N Cause . . . . . : User tried to display a job with a different user name and user does not have special

Figure 9-2 QSYS2.JOBLOG_INFO result

9.9 Librarian Services

Librarian Services consist of two items. Table 9-9 shows their availability by release.

Table 9-9 IBM i Services: Librarian Services

Librarian Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.LIBRARY_LIST_INFO	View	SF99702 Level 3	SF99701 Level 32
QSYS2.OBJECT_STATISTICS()	UDTF	Introduced: Base Enhanced: SF99702 Level 5	Introduced: SF99701 Level 3 Enhanced: SF99701 Level 34

To get an up-to-date list and information about these services and how to use them, see the DB2 for i Wiki. This book describes the QSYS2.OBJECT_STATISTICS() user-defined table function (UDTF) only.

9.9.1 QSYS2.OBJECT_STATISTICS()

This UDTF retrieves information about objects (including both SQL and non-SQL) that are contained in a schema. A wide variety of information can be retrieved by using this function:

- Owner
- Creation date
- Size
- Last used timestamp

For example, you can use this table function to check the journals and journal receivers that exist in your schema or whether your schema contains indexes that were never used, as shown in Example 9-7.

Example 9-7 QSYS2.OBJECT_STATISTICS() usage examples

```
SELECT *
   FROM TABLE (QSYS2.OBJECT_STATISTICS('SIMOTEST ','JRN JRNRCV') ) AS X;

SELECT objname, objsize
   FROM TABLE (QSYS2.OBJECT_STATISTICS('SNAPSHOTS', '*ALL') ) AS X
   WHERE last_used_timestamp is null and sql_object_type = 'INDEX';
```

Note: The syntax allows many options to filter the results. For more information, see the IBM DB2 for i Wiki for documentation about these topics:

- Various object types can be queried together.
- The object type can be prefixed with an asterisk (*) or not. *JRN and JRN are both acceptable.
- IBM i 7.2 has more parameters, such as OBJECT_NAME and an extra TEXT output column that is available, than IBM i 7.1.

The DB2 for i Wiki is at this website:

https://ibm.biz/Bd42AR

In general, this service is a handy way to retrieve information about selected objects that are contained in a library. This service provides a simple way of filtering for specific conditions that can be of interest. If you compare this service with the “native” CL commands **WRKOBJ** and **WRKOBJPDM**, it is obvious that no easy and immediate method exists of discovering the objects that were never used, that exceed a specific size, or that belong to a certain user profile. If you use QSYS2.OBJECT_STATISTICS, you are fully empowered.

Tip: We use this function frequently to check “moldy” SAVFs that our colleagues tend to forget in QGPL.

9.10 Work Management Services

Ten services are currently available. Table 9-10 shows their availability by release.

Table 9-10 IBM i Services: Work Management Services

Work Management Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.ACTIVE_JOB_INFO()	UDTF	SF99702 Level 5 Enhanced: SF99702 Level 9	SF99701 Level 34
QSYS2.GET_JOB_INFO()	UDTF	Introduced: Base Enhanced: SF99702 Level 5 Enhanced: SF99702 Level 9	Introduced: SF99701 Level 23 Enhanced: SF99701 Level 29 Enhanced: SF99701 Level 34
QSYS2.MEMORY_POOL	UDTF	SF99702 Level 9	N/A
QSYS2.MEMORY_POOL_INFO	View	SF99702 Level 9	N/A
QSYS2.OBJECT_LOCK_INFO	View	SF99702 Level 9	N/A
QSYS2.RECORD_LOCK_INFO	UDTF	SF99702 Level 9	N/A
QSYS2.SCHEDULED_JOB_INFO()	View	SF99702 Level 5	SF99701 Level 34
QSYS2.SYSTEM_STATUS	UDTF	SF99702 Level 9	N/A
QSYS2.SYSTEM_STATUS_INFO	View	SF99702 Level 9	N/A
QSYS2.SYSTEM_VALUE_INFO	View	Base	SF99701 Level 26

QSYS2.ACTIVE_JOB_INFO(), QSYS2.OBJECT_LOCK_INFO, and QSYS2.SYSTEM_STATUS_INFO are described.

9.10.1 QSYS2.ACTIVE_JOB_INFO()

This user-defined table function (UDTF) returns one row for every active job in the system. It provides similar information to the **WRKACTJOB** command and the Open List of Jobs (QGYOLJOB) API.

Two uses for this table function are available:

- Retrieve details for active jobs with an optional parameter for retrieving only a subset of jobs
- Measure elapsed statistics for active jobs (parm RESET_STATISTICS)

Note: IBM i 7.1 requires the caller to provide values for all of the parameters. IBM i 7.2 supports default parameter values.

QSYS2.ACTIVE_JOB_INFO simplifies the identification of the users who are using resources on your system, for example, the top temporary storage consumer, top CPU consumer, or top disk I/O counts. Or, you can use QSYS2.ACTIVE_JOB_INFO to determine the distribution of the workload across the subsystems. Information can be grouped, or you can use the basic result set for details.

In Example 9-8, this function is used to gather summary information by subsystem.

Example 9-8 QSYS2.ACTIVE_JOB_INFO usage

```
SELECT count(*) as "Number of Jobs", subsystem as "Subsystem", sum(temporary_storage) as
"Tot. Temp. Storage", sum(cpu_time) as "Tot CPU Time", sum(total_disk_io_count) as "Tot
I/O"
   FROM TABLE(qsys2.active_job_info()) x
   WHERE subsystem is not null
   GROUP BY subsystem
   ORDER BY 4 desc ;
```

Figure 9-3 shows sample results. The query can be refined to subset the data further (subsystem name, and so on, to determine the jobs that are your top resource consumers).

Number of Jobs	Subsystem	Tot. Temp. Storage	Tot CPU Time	Tot I/O
31	QWEBQRY21	1781	1448741	109589
9	QHTTPSVR	964	590142	63573
87	QSYSWRK	839	370150	225087
40	QUSRWRK	417	27323	30911
24	QSERVER	108	2668	12127
2	QINTER	13	971	539
8	QCMN	18	101	554
1	QCTL	2	18	652
1	QSPL	2	14	137
1	QBATCH	2	10	104

Figure 9-3 QSYS2.ACTIVE_JOB_INFO result

The DB2 for i Wiki includes interesting and sophisticated examples, which can provide many more ideas.

9.10.2 QSYS2.OBJECT_LOCK_INFO

The QSYS2.OBJECT_LOCK_INFO view returns one row for every lock that is held for every object on the partition. The values that are returned for the columns in the view closely relate to the values that are returned by the Retrieve Lock Information API and the Retrieve Lock Request Information API. For more information, see the API documentation.

Important: When you query this view, you use a WHERE clause to restrict the result set to avoid excessive use of system resources. By default, information is retrieved for the entire system, and it will take a long time to run.

In Example 9-9, we query the QSYS2.OBJECT_LOCK_INFO view to gather information about locks that are held on objects in library `QPFRDATA`. We are interested in understanding the jobs that are holding locks, locating the programs and modules that are involved, and information about the lock types and counts.

Example 9-9 QSYS2.OBJECT_LOCK_INFO view

```
SELECT object_name, object_type, member_lock_type, lock_state, lock_status,   lock_scope,
job_name,  lock_count, program_library_name, program_name,   module_library_name,
module_name,  procedure_name
   FROM qsys2.object_lock_info
   WHERE object_schema ='QPFRDATA' ;
```

Figure 9-4 shows sample results. The query can be customized to your business needs. More result columns than the columns that are shown can be specified in the SELECT statement.

OBJECT_NAME	OBJECT_TYPE	MEMBER_LO...	LOCK_STA...	LOCK_ST...	LOCK_SC...	JOB_NAME	LOCK_C...	PROG...	PROGRAM_N...	MODULE_LIB...	MODULE_N...	PROCEDUR
QAPMAPPN	*FILE	-	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMAPPN	*FILE	MEMBER	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMAPPN	*FILE	DATA	*SHRUPD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDBOPEN	QBUILDSS1	QDBOPEN	QDBOPEN
QAPMBUS	*FILE	-	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMBUS	*FILE	MEMBER	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMBUS	*FILE	DATA	*SHRUPD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDBOPEN	QBUILDSS1	QDBOPEN	QDBOPEN
QAPMCIOP	*FILE	-	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMCIOP	*FILE	MEMBER	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMCIOP	*FILE	DATA	*SHRUPD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDBOPEN	QBUILDSS1	QDBOPEN	QDBOPEN
QAPMCONF	*FILE	-	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMCONF	*FILE	-	*SHRRD	HELD	JOB	071410/QSYS/CRTPFRDT...	1	QSYS	QDMCOPEN	-	-	-
QAPMCONF	*FILE	MEMBER	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMCONF	*FILE	DATA	*SHRUPD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDBOPEN	QBUILDSS1	QDBOPEN	QDBOPEN
QAPMCONF	*FILE	MEMBER	*SHRRD	HELD	JOB	071410/QSYS/CRTPFRDT...	1	QSYS	QDMCOPEN	-	-	-
QAPMCONF	*FILE	DATA	*SHRUPD	HELD	JOB	071410/QSYS/CRTPFRDT...	1	QSYS	QDBOPEN	QBUILDSS1	QDBOPEN	QDBOPEN
QAPMDIOP	*FILE	-	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMDIOP	*FILE	MEMBER	*SHRRD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDMCOPEN	-	-	-
QAPMDIOP	*FILE	DATA	*SHRUPD	HELD	JOB	071409/QSYS/CRTPFRDTA	1	QSYS	QDBOPEN	QBUILDSS1	QDBOPEN	QDBOPEN

Figure 9-4 QSYS2.OBJECT_LOCK_INFO results

This query is an efficient way to understand the locks that are in place for a specific object. This query offers an immediate understanding of the environment. You do not need sophisticated or specific tools to analyze the collected data. You can get live information by using simple SQL statements.

Tip: A QSYS2.RECORD_LOCK_INFO view is also available.

9.10.3 QSYS2.SYSTEM_STATUS_INFO

The QSYS2.SYSTEM_STATUS_INFO view returns a single row that contains details about the current partition. The information that is returned is similar to the detail that is shown from the Work with System Status (**WRKSYSSTS**) and the Work with System Activity (**WRKSYSACT**) commands.

Note: QSYS2.SYSTEM_STATUS_INFO does not reset the statistical columns. To reset the statistical columns, use the associated table function, which is SYSTEM_STATUS().

In Example 9-10, we are querying the SYSTEM_STATUS_INFO view to collect information about the system environment:

- Jobs that are running
- CPU usage
- Available capacity,
- Used and available storage
- Temporary storage allocation

Example 9-10 Querying QSYS2.SYSTEM_STATUS_INFO

```
SELECT
host_name as "Host Name", total_jobs_in_system as "Tot. Jobs", active_jobs_in_system as
"Act. Jobs", interactive_jobs_in_system as "Int. Jobs" , active_threads_in_system as "Act.
Threads" , elapsed_time as "Elapsed Time", elapsed_cpu_used as "Elapsed CPU" ,
current_cpu_capacity as "Cur. CPU Capacity", average_cpu_utilization as "Avg. CPU Util." ,
minimum_cpu_utilization as "Min CPU Util.", maximum_cpu_utilization as  "Max CPU Util.",
system_asp_storage as "Sys ASP Storage", total_auxiliary_storage as "Tot. Aux. Storage" ,
system_asp_used as "Sys. ASP Used" , current_temporary_storage as "Cur. Temp. Storage" ,
maximum_temporary_storage_used as "Max. Temp. Storage Used"
   FROM  qsys2.system_status_info  ;
```

A single row that contains all of the requested information is returned. Example results are shown in Figure 9-5 and Figure 9-6.

Host Name	Tot. Jobs	Act. Jobs	Int. Jobs	Act. Threads	Elapsed Time	Elapsed CPU	Cur. CPU Capacity	Avg. CPU Util.	Min CPU Util.	Max CPU Util.
OS72	336	206	0.00	687	1322	0.70	1.00	0.51	0.51	0.51

Figure 9-5 QSYS2.SYSTEM_STATUS_INFO result

Max CPU Util.	Sys ASP Storage	Tot. Aux. Storage	Sys. ASP Used	Cur. Temp. Storage	Max. Temp. Storage Used
0.51	634701	634701	59.28	14761	14831

Figure 9-6 QSYS2.SYSTEM_STATUS_INFO results

Assume that the following scenario exists to address a business need. We manage multiple systems or logical partitions (LPARs). We want to monitor all of them. We need summary information that is reported collectively. We know that it is possible to address queries to remote databases by using IBM Distributed Relational Database Architecture (IBM DRDA®). And, we can use three-part naming in our statements.

At the time of writing this book, a single SQL statement can reference two databases in an INSERT INTO <LOCAL> SELECT FROM <REMOTE> environment. We will use a temporary table to hold the results to present the information from various databases, as shown in Example 9-11.

Example 9-11 System information from multiple systems

```
DECLARE GLOBAL TEMPORARY TABLE session.sys_sts_tbl  AS
   (SELECT current_timestamp as timestamp, host_name  , total_jobs_in_system,
   active_jobs_in_system, interactive_jobs_in_system , active_threads_in_system  ,
   elapsed_time , elapsed_cpu_used  , current_cpu_capacity , average_cpu_utilization ,
   minimum_cpu_utilization , maximum_cpu_utilization , system_asp_storage  ,
   total_auxiliary_storage  , system_asp_used  , current_temporary_storage ,
   maximum_temporary_storage_used
      FROM qsys2.system_status_info)
      WITH NO DATA;

INSERT INTO  session.sys_sts_tbl
   (SELECT current_timestamp as timestamp, host_name  , total_jobs_in_system,
   active_jobs_in_system, interactive_jobs_in_system , active_threads_in_system  ,
   elapsed_time , elapsed_cpu_used  , current_cpu_capacity , average_cpu_utilization ,
   minimum_cpu_utilization , maximum_cpu_utilization , system_asp_storage  ,
   total_auxiliary_storage  , system_asp_used  , current_temporary_storage ,
   maximum_temporary_storage_used
      FROM qsys2.system_status_info );

INSERT INTO session.sys_sts_tbl
   (SELECT current_timestamp as timestamp,  host_name  , total_jobs_in_system,
   active_jobs_in_system, interactive_jobs_in_system , active_threads_in_system  ,
   elapsed_time , elapsed_cpu_used  , current_cpu_capacity , average_cpu_utilization ,
   minimum_cpu_utilization , maximum_cpu_utilization , system_asp_storage  ,
   total_auxiliary_storage  , system_asp_used  , current_temporary_storage ,
   maximum_temporary_storage_used
      FROM  trpibmi.qsys2.system_status_info);

   SELECT timestamp , host_name as "Host Name", total_jobs_in_system as "Tot. Jobs",
   active_jobs_in_system as "Act. Jobs", interactive_jobs_in_system as "Int. Jobs" ,
   active_threads_in_system as "Act. Threads" , elapsed_time as "Elapsed Time",
   elapsed_cpu_used as "Elapsed CPU" , current_cpu_capacity as "Cur. CPU Capacity",
   average_cpu_utilization as "Avg. CPU Util." , minimum_cpu_utilization as "Min CPU
```

```
   Util.", maximum_cpu_utilization as  "Max CPU Util.", system_asp_storage as "Sys ASP
   Storage", total_auxiliary_storage as "Tot. Aux. Storage" , system_asp_used as "Sys. ASP
   Used" , current_temporary_storage as "Cur. Temp. Storage" ,
   maximum_temporary_storage_used as "Max. Temp. Storage Used"
      FROM SESSION.SYS_STS_TBL;

select * from qtemp.SYS_STS_TBL;
```

By issuing the statements in the previous example toward each of the systems (LPARs) that you manage and by querying the resulting temporary table, you can obtain information about all of your LPARs collectively.

Alternatively, think of this approach as “dressing up this idea with a table function”, as shown in Example 9-12.

Example 9-12 System status

```
CREATE OR REPLACE FUNCTION SIMO.SYS_STS()
RETURNS TABLE(
HOST_NAME VARCHAR(255) ,
TOTAL_JOBS_IN_SYSTEM INTEGER,
active_jobs_in_system INTEGER,
interactive_jobs_in_system DECIMAL(5,2), active_threads_in_system INTEGER , elapsed_time
INTEGER, elapsed_cpu_used  DECIMAL(5,2), current_cpu_capacity DECIMAL(5,2),
average_cpu_utilization DECIMAL(5,2), minimum_cpu_utilization DECIMAL(5,2),
maximum_cpu_utilization DECIMAL(5,2), system_asp_storage BIGINT , total_auxiliary_storage
BIGINT, system_asp_used DECIMAL(5,2) , current_temporary_storage INTEGER,
maximum_temporary_storage_used   INTEGER
)
LANGUAGE SQL
DISALLOW PARALLEL
MODIFIES SQL DATA
NOT FENCED
BEGIN
DECLARE TABLE_ALREADY_EXISTS CONDITION FOR '42710';
DECLARE CONTINUE HANDLER FOR TABLE_ALREADY_EXISTS
DELETE FROM SESSION.SYS_STS_TBL;
DECLARE GLOBAL TEMPORARY TABLE SESSION.SYS_STS_TBL as (  select
HOST_NAME  ,
TOTAL_JOBS_IN_SYSTEM ,
active_jobs_in_system,
interactive_jobs_in_system , active_threads_in_system  , elapsed_time , elapsed_cpu_used  ,
current_cpu_capacity , average_cpu_utilization , minimum_cpu_utilization ,
maximum_cpu_utilization , system_asp_storage  , total_auxiliary_storage  , system_asp_used
, current_temporary_storage , maximum_temporary_storage_used    from
qsys2.system_status_info) with no data;

INSERT INTO SESSION.SYS_STS_TBL (select HOST_NAME  ,
TOTAL_JOBS_IN_SYSTEM ,
active_jobs_in_system,
interactive_jobs_in_system , active_threads_in_system  , elapsed_time , elapsed_cpu_used  ,
current_cpu_capacity , average_cpu_utilization , minimum_cpu_utilization ,
maximum_cpu_utilization , system_asp_storage  , total_auxiliary_storage  , system_asp_used
, current_temporary_storage , maximum_temporary_storage_used    from
qsys2.system_status_info );
INSERT INTO SESSION.SYS_STS_TBL (select HOST_NAME  ,
TOTAL_JOBS_IN_SYSTEM ,
active_jobs_in_system,
```

```
interactive_jobs_in_system , active_threads_in_system  , elapsed_time , elapsed_cpu_used  ,
current_cpu_capacity , average_cpu_utilization , minimum_cpu_utilization ,
maximum_cpu_utilization , system_asp_storage  , total_auxiliary_storage  , system_asp_used
, current_temporary_storage , maximum_temporary_storage_used    from
trpibmi.qsys2.system_status_info);

RETURN
SELECT HOST_NAME  ,
TOTAL_JOBS_IN_SYSTEM ,
active_jobs_in_system,
interactive_jobs_in_system , active_threads_in_system  , elapsed_time , elapsed_cpu_used  ,
current_cpu_capacity , average_cpu_utilization , minimum_cpu_utilization ,
maximum_cpu_utilization , system_asp_storage  , total_auxiliary_storage  , system_asp_used
, current_temporary_storage , maximum_temporary_storage_used
FROM SESSION.SYS_STS_TBL;
END;
```

Example 9-13 demonstrates querying the previous table function.

Example 9-13 System status query

```
select  HOST_NAME  ,
TOTAL_JOBS_IN_SYSTEM ,
active_jobs_in_system,
interactive_jobs_in_system , active_threads_in_system  , elapsed_time , elapsed_cpu_used  ,
current_cpu_capacity , average_cpu_utilization , minimum_cpu_utilization ,
maximum_cpu_utilization , system_asp_storage  , total_auxiliary_storage  , system_asp_used
, current_temporary_storage from  table(SYS_STS()) as X ;
```

Figure 9-7 shows the result that was produced by using Example 9-13.

HOST_NAME	TOTAL_JOBS_IN_SYSTEM	ACTIVE_JOBS_IN_SYSTEM	INTERACTIVE_JOBS_IN_SYSTEM	ACTIVE_THREADS_IN_SYSTEM	ELAPSED_TIME	... continued below...
OS72	367	210	0.00	721	518	
TRPIBMI	3065	240	0.00	1187	517	

ED_TIME	ELAPSED_CPU_US...	CURREN...	AVERAGE_CPU...	MINIMUM_CPU_...	MAXIMUM_CPU...	SYSTEM_ASP_STORAGE	TOTAL_AUXILIARY_STORAGE	SYSTEM_ASP_US...	CURRENT_TEMPORARY_STOR...
518	0.80	1.00	0.51	0.51	0.51	634701	634701	85.82	24607
517	0.70	1.00	0.47	0.47	0.47	343597	343597	35.16	9052

Figure 9-7 System status result

If your environment is flexible or highly dynamic, you might prefer to pass database names as variables and define a loop that repeats the INSERT statement for each database name that you pass.

If you want to keep this information for statistical reasons, you must use a normal, permanent table.

9.11 Java Services

Two services are available for Java. Table 9-11 shows their availability by release.

Table 9-11 IBM i Services: Java Services

Java Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.SET_JVM()	Procedure	SF99702 Level 5	SF99701 Level 34
QSYS2.JVM_INFO()	View	SF99702 Level 5	SF99701 Level 34

In this book, only the QSYS2.SET_JVM() is described.

9.11.1 QSYS2.SET_JVM()

You can use this procedure to manage specific Java virtual machine (JVM) jobs in an easy way. A simple procedure call can be used to manage your jobs:

- GC_ENABLE_VERBOSE: Enables verbose garbage collection detail.
- GC_DISABLE_VERBOSE: Disables verbose garbage collection detail.
- GENERATE_HEAP_DUMP: Generates information about the JVM's heap. It generates a dump of all of the heap space allocations that were not yet freed.
- GENERATE_SYSTEM_DUMP: Generates system detail for the JVM. It generates a binary format raw memory image of the job that was running when the dump was initiated.
- GENERATE_JAVA_DUMP: Generates Java detail for the JVM. It generates multiple files that contain diagnostic information for the JVM and the Java applications that are running within the JVM.

This procedure is an alternative to using the Work with JVM Jobs (**WRKJVMJOB**) CL command.

By using the call in Example 9-14, you can stop gathering verbose garbage collection details for a specific job.

Example 9-14 QSYS2.SET_JVM usage

```
CALL qsys2.set_jvm('066518/QWQADMIN/WQLIB85','GC_DISABLE_VERBOSE');
```

Use the JVM_INFO view to identify the jobs of interest.

9.12 IBM i Application Services

Only one service is available in this category. Table 9-12 shows its availability by release.

Table 9-12 IBM i Services - Application Services

Application Services	Type of service	IBM i 7.2	IBM i 7.1
QSYS2.QCMDEXC()	Procedure	Base	Introduced: Base Enhanced: SF99701 Level 28

9.12.1 QSYS2.QCMDEXC()

You can use this procedure to call any IBM i CL batch command through SQL.

Note: This powerful tool allows your SQL application to take advantage of IBM functions that are only available in CL.

In Example 9-15, the Display Program References (**DSPPGMREF**) CL command is used to find a list of the system objects that are referred to by programs whose names start with a certain string and that are in a specific library.

Example 9-15 QSYS2.QCMDEXC usage

```
CALL qsys2.qcmdexc('dsppgmref pgm(qwebqry/cmrun*) output(*outfile) objtype(*all)
outfile(simo/dsppgmref)');
```

Example 9-16 shows the use of QSYS2.QCMDEXC to add a library to the job's library list. The library name is a variable value that is provided at run time.

Example 9-16 Using QSYS2.QCMDEXC in a procedure

```
CREATE OR REPLACE PROCEDURE  simo.mylibl (in v_library_name varchar(10))
LANGUAGE SQL
BEGIN
CALL qsys2.qcmdexc('addlible ' concat v_library_name);
END;
```

Note: After the procedure is called, you can use the QSYS2.LIBRARY_LIST_INFO view if you want to check your library list.

A

Allocating, describing, and manipulating descriptors

This appendix contains an explanation of Structured Query Language (SQL) descriptors. It describes what they are and how to use them when you develop dynamic SQL applications. In addition, this appendix includes details for the following SQL statements that are used to allocate, describe, and manipulate the following SQL descriptors:

- ALLOCATE/DEALLOCATE SQL DESCRIPTOR
- DESCRIBE INPUT SQL DESCRIPTOR
- DESCRIBE OUTPUT
- SET/GET SQL DESCRIPTOR
- USING SQL DESCRIPTOR

This appendix includes the following topics:

- Introduction to SQL descriptors
- Allocating SQL descriptors
- DEALLOCATE SQL DESCRIPTOR
- Describing SQL descriptors
- Manipulating data to and from descriptors

Introduction to SQL descriptors

An *SQL descriptor* is an alternative means of coding values that replace parameter marker placeholders in a prepared SQL statement. The use of an SQL descriptor is beneficial when the list of values is long (for example, a dynamic INSERT statement). Also, SQL descriptors are useful when you develop dynamic, reusable SQL modules, as described in Chapter 8, “Creating flexible and reusable procedures” on page 227.

SQL descriptors

An SQL descriptor is a structure that contains a list of columns and values. Descriptors are similar to data structures that are used in host-centric high-level languages (HLL) because they contain both record formats and data. However, the similarity ends there. Unlike HLL fixed-format structures, SQL descriptors are dynamic, that is, the structure changes, depending on the type of SQL statement, select or non-select, that is processed.

Two types of descriptors are available. One type is defined with the ALLOCATE DESCRIPTOR statement, and it is referred to as an SQL descriptor. The other type is referred to as an SQL descriptor area (SQLDA), and it is defined by using host-centric language code structures in embedded SQL. For this appendix, the focus is on SQL descriptors. Where the term *descriptor* is used, it implies SQL descriptors. For more information about SQLDAs, see DB2 for i SQL reference, which is at this website:

https://ibm.biz/Bd42dh

How to use SQL descriptors

The meaning of the information in an SQL descriptor depends on its use. Certain descriptors provide information about parameter markers in a prepared statement, or the IN and INOUT parameters of a procedure. These SQL descriptors are referred to as *input descriptors*.

Other descriptors provide information about the result table columns in a prepared statement, the INOUT or OUT parameters of a procedure, or the columns in a table or view. These SQL descriptors are referred to as *table* or *output descriptors*.

The SQL descriptor is divided into two parts: a header structure consists of four fields, which are followed by an array of structures. Each occurrence of the array is made up of two structures, which are referred to as *base* and *extended*. Within the structures are attributes that are referred to as *items*. The information that is contained within the items is used to communicate between SQL and the procedure. The information that relates to SQL descriptors will be described in more detail throughout this appendix.

To further explain the difference between table (output) and input descriptors, Example A-1 contains a procedure with three parameters: one input only, one both input and output, and one output only.

Example A-1 VARIOUS_PARAMETER_TYPE procedure

```
CREATE PROCEDURE Various_Parameter_Types (
   IN I_type CHAR(1),                                          1
   INOUT IO_dept CHAR(3),                                      1
   OUT O_total DECFLOAT)                                       1
   LANGUAGE SQL

P1: BEGIN
   IF I_type = 'C' THEN
```

```
      SELECT COUNT(*) INTO O_total
      FROM employee
      WHERE workdept = IO_dept;
   ELSE
       SELECT SUM(salary) INTO O_total
      FROM employee
      WHERE workdept = IO_dept;
   END IF;

END P1
```

Notes about Example A-1:

1 Three parameters are used in this procedure: one input only, one both input and output and one output only.

To allocate the descriptors that are associated with the parameters in Example A-1 on page 286, the following set of SQL statements will be executed inside a procedure. See Example A-2.

Example A-2 Allocating the descriptors

```
ALLOCATE SQL DESCRIPTOR 'Redbook_D1';
SET v_sqlString = 'CALL Various_Parameter_Types (?,?,?)';
PREPARE Various_Parameter_Types_stmt FROM v_sqlstring;
DESCRIBE OUTPUT Various_Parameter_Types_stmt USING SQL DESCRIPTOR Redbook_D1;
DESCRIBE INPUT Various_Parameter_Types_stmt USING SQL DESCRIPTOR Redbook_D1;
```

The statement string was described twice, one time by using the DESCRIBE (OUTPUT) statement and again by using the DESCRIBE INPUT statement. These statements will be described later in this section.

Table A-1 contains a list of the base structure items that are contained within the descriptor `Redbook_D1` after the execution of the DESCRIBE (OUTPUT) statement.

Table A-1 Pseudo descriptor for DESCRIBE OUTPUT

Item	NAME	TYPE	LENGTH	PRECISION	SCALE
1	IO_DEPT	CHARACTER	3	0	0
2	O_TOTAL	REAL	4	24	0

Only the parameters that are defined as INOUT or OUT appear in Table A-1.

Table A-2 contains a list of the base structure items that are contained within the descriptor `Redbook_D1` after the execution of the DESCRIBE INPUT statement.

Table A-2 Pseudo descriptor for DESCRIBE INPUT

Item	NAME	TYPE	LENGTH	PRECISION	SCALE
1	I_TYPE	CHARACTER	1	0	0
2	IO_DEPT	CHARACTER	3	0	0

Only the parameters that are defined as IN or INOUT appear in Table A-2.

You can learn more about the makeup of descriptors at the following website:

https://ibm.biz/Bd4PKN

The following statements are used in procedures to manipulate SQL descriptors:

- ALLOCATE/DEALLOCATE SQL DESCRIPTOR
- DESCRIBE INPUT SQL DESCRIPTOR
- DESCRIBE (OUTPUT)
- SET/GET SQL DESCRIPTOR
- USING SQL DESCRIPTOR

After the descriptor is fully updated with data, it can be used by the following SQL statements:

- OPEN
- FETCH
- CALL
- EXECUTE

The previous statements all support the USING SQL DESCRIPTOR clause for assigning host variables to input descriptors. In addition, except for the OPEN statement, they also support INTO SQL DESCRIPTOR for assigning values to table (output) descriptors. For more information and examples about using these statements with descriptors, see Chapter 8, “Creating flexible and reusable procedures” on page 227.

Allocating SQL descriptors

The ALLOCATE/DEALLOCATE SQL DESCRIPTOR statements are used to create and destroy SQL descriptors.

Figure A-1 shows the syntax for the ALLOCATE DESCRIPTOR.

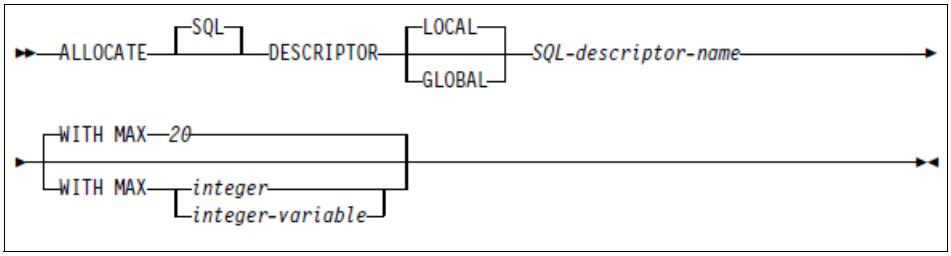

Figure A-1 ALLOCATE DESCRIPTOR syntax

The following information describes several of the parameters in more detail:

- The SQL qualifier of the ALLOCATE SQL DESCRIPTOR statement is optional. You might want to include it for clarity when you reference the SQL descriptor in other SQL statements.
- LOCAL or GLOBAL: An SQL descriptor can be defined for a specific invocation (LOCAL) or for the entire session (GLOBAL). A global descriptor only needs to be allocated one time to allow the descriptor to be used in any SQL procedure that is called within the same session.

- Descriptor names can be up to 128 characters. Use distinctive names when you define a descriptor. The preferred practice is to use variables for global descriptor names. In addition, a global variable can be used to export global descriptor names for later use by flexible procedures.
- WITH MAX: You can use the WITH MAX clause to define the number of variable entries that are contained in the SQL descriptor. If you do not use it, a default value of 20 variables is defined for the descriptor. Set the value when you use local descriptors because you need to know the number of entries when you allocate the descriptor. Use the default for global descriptors because this value will likely change throughout the session.

A procedure that allocates a global descriptor is shown in Example A-3.

Example A-3 ALLOCATE_GLOBAL_DESCRIPTOR Procedure

```
CREATE OR REPLACE PROCEDURE Allocate_Global_Descriptor (
    IN p_Global_Descriptor VARCHAR(128))                                    1
P1: BEGIN
-- Procedure logic begins here
   ALLOCATE SQL DESCRIPTOR GLOBAL p_Global_Descriptor;                      2
   -- Construct SQL statement
   SET gv_Global_Descriptor = p_Global_Descriptor;                          3

   END P1
```

Notes about Example A-3:

1 A parameter that contains the global descriptor name is required.

2 The global descriptor is allocated by using the name that is passed to this procedure. The initial size is 20 variables.

3 The global descriptor name is exported to a global variable for later use. For information about the use of global descriptors in a multiple step transaction, see 8.2.2, “Simplified SQL descriptor usage” on page 233.

DEALLOCATE SQL DESCRIPTOR

SQL descriptors are implicitly deallocated when the procedure ends (local) or when the session ends (global). You can use the DEALLOCATE DESCRIPTOR statement to explicitly deallocate a descriptor if you need to reallocate the size of the descriptor.

Figure A-2 shows the syntax of the DEALLOCATE DESCRIPTOR SQL statement.

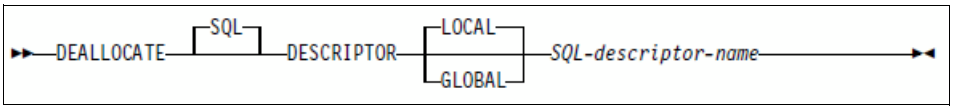

Figure A-2 DEALLOCATE DESCRIPTOR syntax

The SQL descriptor name must match an existing local or global descriptor. If the descriptor was allocated as a global descriptor, it must be deallocated as a global descriptor.

A procedure that deallocates a global descriptor is shown in Example A-4.

Example A-4 DEALLOCATE_GLOBAL_DESCRIPTOR procedure

```
CREATE OR REPLACE PROCEDURE Deallocate_global_descriptor (
   IN p_Global_Descriptor VARCHAR(128) DEFAULT gv_Global_Descriptor)              1

P1: BEGIN

   DEALLOCATE SQL DESCRIPTOR GLOBAL p_Global_Descriptor ;                         2

END P1
```

Notes about Example A-4:

1 The descriptor name is either passed explicitly or imported from a global variable.
2 A previously allocated global descriptor will be deallocated.

Describing SQL descriptors

SQL DESCRIBE statements are used to populate descriptors with information about a prepared dynamic SQL statement. THe DESCRIBE INPUT statement is used with input descriptors. The statements that are used to populate a table descriptor are listed:

- DESCRIBE OUTPUT
- DESCRIBE CURSOR
- DESCRIBE PROCEDURE
- DESCRIBE TABLE
- PREPARE... USING

DESCRIBE INPUT

The DESCRIBE INPUT statement is used to populate an input descriptor with information about the IN and INOUT parameter markers of a select or non-select prepared statement.

Figure A-3 shows the syntax for the DESCRIBE INPUT statement.

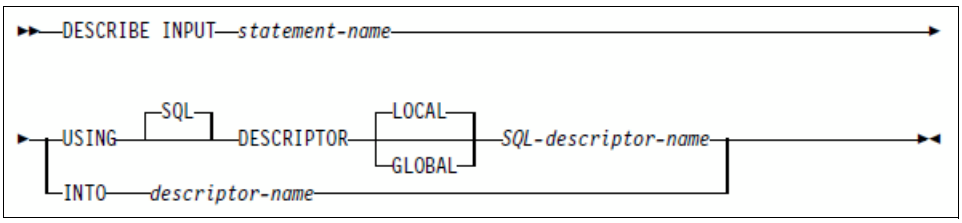

Figure A-3 DESCRIBE INPUT syntax

The following list describes several of the parameters in more detail:

- The statement name is the same as the statement name of the previously prepared statement.
- USING SQL DESCRIPTOR references the name of a previously allocated SQL DESCRIPTOR.
- INTO is not used when you use SQL DESCRIPTORS.

Example A-5 contains a procedure that is used to describe the parameter markers that are contained in a dynamic SQL INSERT statement.

Example A-5 Describe_Input_Descriptor procedure

```
CREATE OR REPLACE PROCEDURE Describe_Input_Descriptor (
IN p_Global_Descriptor VARCHAR(128)                                    1
, v_sqlString CLOB (2M))                                               2

P1: BEGIN
DECLARE v_parameter_markers SMALLINT;                                  3

PREPARE example213_s1 FROM v_sqlString;                                4
GET DIAGNOSTICS v_parameter_markers = DB2_NUMBER_PARAMETER_MARKERS;     5

SET SQL DESCRIPTOR GLOBAL p_Global_Descriptor COUNT = v_parameter_markers;   6
DESCRIBE INPUT example213_s1 USING SQL DESCRIPTOR GLOBAL p_Global_Descriptor;   7

END P1
```

Notes about Example A-5:

1 The SQL descriptor name is passed as a variable to the procedure.

2 The statement string is passed to the procedure as a variable.

3 This variable is used as output for the GET DIAGNOSTICS SQL statement.

4 The statement string variable that contains the dynamic SQL statement is prepared.

5 The GET DIAGNOSTICS SQL statement is used to return the number of parameter markers that are contained in the prepared statement. For more information about GET DIAGNOSTICS, see the IBM i Knowledge Center:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome

6 The SET DESCRIPTOR GLOBAL statement is used to set the descriptor header count. The SET DESCRIPTOR statement is described in “SET SQL DESCRIPTOR” on page 302.

7 The DESCRIBE INPUT statement is used to set the descriptor detail entries with information based on the IN or INOUT parameter markers that are contained within the statement string.

The following examples of SQL statements are used to call the procedure in Example A-5.

Example A-6 shows the CALL statement for the DESCRIBE_INPUT_DESCRIPTOR procedure that passes a statement string that contains an INSERT statement.

Example A-6 Calling the DESCRIBE_INPUT_DESCRIPTOR procedure

```
CALL Allocate_Global_Descriptor (‘Redbooks’);
CALL Describe_Input_Descriptor(
   'Redbooks',
   'INSERT INTO employee (empno, lastname, midinit, workdept, edlevel, salary)
VALUES(?,?,?,?,?,?)');
```

Table A-3 contains a list of the base structure items that are contained within the descriptor after the procedure is called in Example A-5 on page 291. For a complete list of the descriptor items, see “Get-item-info” on page 310.

Table A-3 Pseudo descriptor for dynamic INSERT

Item	NAME	TYPE	LENGTH	PRECISION	SCALE
1	0001	CHARACTER	6	0	0
2	0002	VARCHAR	12	0	0
3	0003	CHARACTER	1	0	0
4	0004	VARCHAR	15	0	0
5	0005	CHARACTER	3	0	0
6	0006	DECIMAL	5	9	2

Notes about Table A-3:

- The descriptor contains six base occurrences, one for each parameter marker that is defined in the INSERT VALUES clause.
- The item represents the ordinal position of the parameter marker as it appears in the prepared statement. Item is not a part of the descriptor but it is shown for clarity only.
- NAME contains the ordinal position of the parameter marker. For table descriptors, NAME contains the actual field name.
- TYPE contains the data type code for the parameter marker based on the definition of the table (employee) that is referenced on the INTO clause of the INSERT statement. It was translated for clarity. For a complete list of data type codes, search on `GET DESCRIPTOR` in the IBM i Knowledge Center.
- LENGTH represents the maximum length of the data, in bytes or characters, that is contained within the host variable that is represented by a parameter marker. The actual number of characters of the host variable is used if the data type is a character or graphic string, an XML type, or a datetime type, the length represents the number of characters (not bytes). For example, parameter 1 is 6 characters compared to parameter 6, which is 5 bytes. For a complete description of data type codes and lengths, search on `GET DESCRIPTOR` in the IBM i Knowledge Center.
- PRECISION represents the exact length of a numeric parameter. For character parameters, the value is 0. For example, parameter 6 is defined as DECIMAL with a LENGTH of 5 bytes and PRECISION of 9 characters. Parameter 2 is defined as VARCHAR with a LENGTH of 12 and a PRECISION of 0 (not numeric).
- SCALE represents the exact length of the decimal positions for parameters that are defined as DECIMAL or NUMERIC. Otherwise, it is zero.

Example A-7 shows the SQL CALL statement for the procedure that is shown in Example A-5 on page 291.

Example A-7 CALL Describe_Input_Descriptor

```
CALL Allocate_Global_Descriptor ('Redbooks');
CALL Describe_Input_Descriptor(
   'Redbooks'                                                          1
   ,'Select empno, lastname, midinit, workdept, edlevel, salary        2
     FROM employee
     WHERE empno = ?'
   );
```

Notes about Example A-7:

1 The name of the global descriptor to use in this invocation.
2 The statement string to prepare and describe for this invocation.

Table A-4 contains a list of the base structure items that are contained within the descriptor after the procedure is called.

Table A-4 Pseudo descriptor for Example A-7

Item	ITEM NAME	TYPE	ITEM LENGTH	ITEM PRECISION	ITEM SCALE
1	0001	CHARACTER	6	0	0

Note about Table A-4:

The descriptor contains one base occurrence because only one parameter marker is defined in the WHERE clause.

DESCRIBE OUTPUT

The DESCRIBE OUTPUT statement is used to populate a descriptor with information about the columns in the result table of a prepared SELECT statement or the INOUT or OUT parameters in a prepared CALL statement.

Figure A-4 contains the syntax for the DESCRIBE OUTPUT SQL statement.

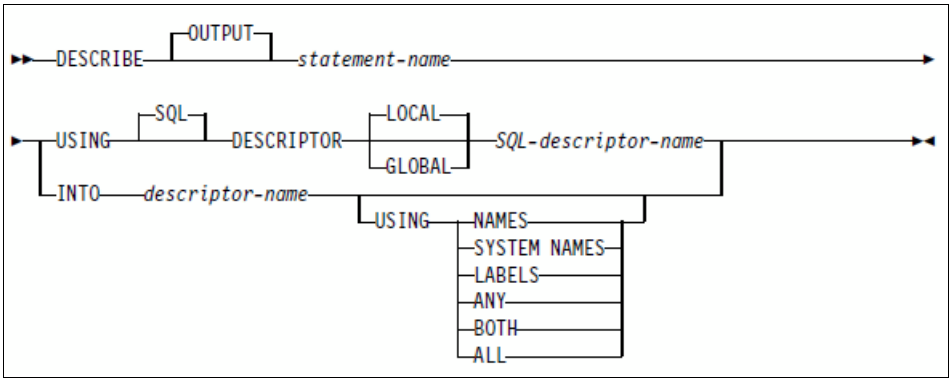

Figure A-4 DESCRIBE OUTPUT statement syntax

The following information describes several of the parameters in more detail:

- The OUTPUT designator is optional. A table descriptor is assumed.
- The statement name is the same as the statement name of the previously prepared statement.
- USING SQL DESCRIPTOR references the name of a previously allocated SQL DESCRIPTOR.
- INTO is not relevant with SQL descriptors.

Example A-8 shows a procedure that prepares and describes a dynamic SQL statement.

Example A-8 DESCRIBE_OUTPUT_DESCRIPTOR procedure

```
CREATE PROCEDURE Describe_Output_Descriptor (
   IN p_Global_Descriptor VARCHAR(128) DEFAULT gv_global_descriptor                1
   , v_sqlString CLOB (2M) DEFAULT gv_sql_string)                                  1
   SPECIFIC dscrboutda
P1: BEGIN

   PREPARE dscrboutda_d1 FROM v_sqlString ;
   DESCRIBE dscrboutda_d1 USING SQL DESCRIPTOR GLOBAL p_Global_Descriptor;         2

END P1
```

Notes about Example A-8:

1 The global descriptor and statement string parameters use global variables as defaults. If a parameter is not specified, the value in the global variable is used.

2 The descriptor is populated with the result table columns of a select statement or the INOUT or OUT columns of a routine. In addition, if the prepared statement is a SELECT, the descriptor header COUNT is set to the number of columns in its result table. Therefore, the SET DESCRIPTOR is not required.

Example A-9 shows the SQL CALL statement for the procedure that is shown in Example A-8.

Example A-9 CALL Describe_Output_Descriptor procedure

```
CALL Allocate_Global_Descriptor ('Redbooks');                                      1
CALL Describe_Output_Descriptor(DEFAULT,'CALL Various_Parameter_Types (?,?,?;       2

SET gv_sql_string = 'CALL Various_Parameter_Types (?,?,?)';                        3
CALL Describe_Output_Descriptor();

SET gv_sql_string =                                                                4
   'SELECT empno, firstnme, midinit, lastname, workdept, edlevel, salary
    FROM employee WHERE empno LIKE ?';
CALL Describe_Output_Descriptor();

CREATE OR REPLACE VIEW VEMP_INFO AS                                                5
      SELECT empno, firstnme, midinit, lastname, workdept, edlevel, salary
      FROM EMPLOYEE;
SET gv_sql_string = 'Select * FROM vemp_info WHERE empno LIKE ?';                  6
CALL Describe_Output_Descriptor();
```

Notes about Example A-9 on page 294:

1 The global descriptor global variable is set in this procedure from Example A-3 on page 289. The descriptor name is `Redbooks`.

2 The reserved word DEFAULT is not required but it provides good documentation in addition to being a placeholder for a parameter that will use the defined default.

3 The global variable *gv_sql_string* is populated before the procedure is called. The SQL CALL statement does not need to include the parameter values. They will be imported from the global variables. The descriptor definition for 2 and 3 is identical to Table A-1 on page 287.

4 The statement string is a select statement with seven output columns and one input parameter marker. The descriptor definition is shown in Table A-5 after the procedure is called. Only the seven output columns are listed in Table A-5.

5 A view is created that matches the previous select statement without the WHERE clause.

6 The statement string is a select statement that references the SQL view with the WHERE clause and one parameter marker. The descriptor definition is the same as number 4 after the procedure is called.

Table A-5 Pseudo descriptor for Example A-9 on page 294

Item	NAME	TYPE	LENGTH	PRECISION	SCALE
1	EMPNO	CHARACTER	6	0	0
2	FIRSTNME	VARCHAR	12	0	0
3	MIDINIT	CHARACTER	1	0	0
4	LASTNAME	VARCHAR	15	0	0
5	WORKDEPT	CHARACTER	3	0	0
6	EDLEVEL	SMALLINT	2	4	0
7	SALARY	DECIMAL	5	9	2

PREPARE...USING

The PREPARE statement can be used to combine the functions of PREPARE and DESCRIBE OUTPUT in a single statement.

Example A-10 contains a code snippet that shows the difference between PREPARE...USING and PREPARE and DESCRIBE.

Example A-10 PREPARE...USING versus PREPARE and DESCRIBE output

```
PREPARE...USING statement:
   PREPARE example217_s1 USING SQL DESCRIPTOR example217_d1 FROM v_sqlString;
is the equivalent of PREPARE and DESCRIBE
   EXEC SQL PREPARE example217_s1 FROM v_sqlString;
   EXEC SQL DESCRIBE example217_s1 USING SQL DESCRIPTOR example217_d1
```

DESCRIBE CURSOR

The DESCRIBE CURSOR statement is used to populate a descriptor with information about the columns in the result table of an opened or allocated SQL cursor. The DESCRIBE CURSOR statement is not a substitute for a DESCRIBE or DESCRIBE INPUT to describe a prepared statement string that contains a SELECT statement.

Figure A-5 contains the syntax for the DESCRIBE CURSOR SQL statement.

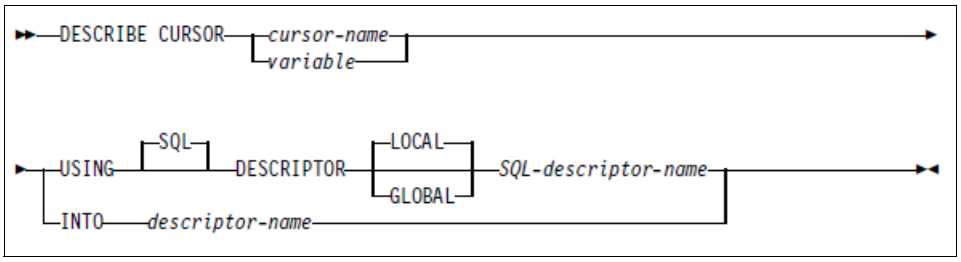

Figure A-5 DESCRIBE CURSOR statement syntax

Minor differences exist between the syntax of the DESCRIBE OUTPUT statement and the syntax of the DESCRIBE CURSOR statement. The differences are described:

- CURSOR is used instead of OUTPUT.
- The cursor name can be contained in a local variable. If a variable is used, the name must be in uppercase (unless delimited), left-aligned, and padded with blanks.

Example A-11 shows a procedure that uses the DESCRIBE CURSOR statement to populate a table descriptor.

Example A-11 DESCRIBE_CURSOR_DESCRIPTOR procedure

```
CREATE PROCEDURE Describe_Cursor_Descriptor (
   IN p_Global_Descriptor VARCHAR(128) DEFAULT gv_global_descriptor
   ,IN v_sqlString CLOB (2M) DEFAULT gv_sql_String)
   RESULT SETS 1;
P1: BEGIN

   -- Result set cursor
   DECLARE result_set_cursor CURSOR FOR dynamic_stmt;                          1
   -- Procedure logic begins here
   PREPARE dynamic_stmt FROM v_sqlString ;                                     2

   --Cursor must be opened before DESCRIBE
   OPEN result_set_cursor;                                                     3

   DESCRIBE CURSOR result_set_cursor USING SQL DESCRIPTOR GLOBAL p_Global_Descriptor;   4

END P1
```

Notes about Example A-11:

1 Declares a result set for the cursor that is defined in this procedure.
2 Prepares the dynamic statement string for the declared cursor.
3 Opens the cursor for the result set that is declared in this procedure.
4 Describes the result set of the cursor that is declared in this procedure.

Example A-12 shows the SQL CALL statement for the procedure in Example A-11 on page 296.

Example A-12 Calling DESCRIBE_CURSOR_DESCRIPTOR

```
CALL Allocate_Global_Descriptor ('Redbooks');
SET gv_sql_string = 'SELECT empno, firstnme, midinit, lastname, workdept, edlevel, salary
                     FROM employee';
CALL describe_cursor_descriptor();                                                    1

CALL Allocate_Global_Descriptor ('Redbooks');
SET gv_sql_string = 'SELECT empno, firstnme, midinit, lastname, workdept, deptname,
                            edlevel, salary
                     FROM employee INNER JOIN department ON workdept = deptno
                     ORDER BY workdept, lastname';
CALL describe_cursor_descriptor();                                                    2
```

Notes about Example A-12:

1 The statement string is exported to the global variable, which is imported as the default for the DESCRIBE_CURSOR_DESCRIPTOR procedure. The populated pseudo descriptor is the same as the pseudo descriptor that is shown in Table A-5 on page 295.

2 Because the statement string is passed as a variable, this procedure can be used to service many different applications. The second example contains a join of two tables. The pseudo descriptor is shown in Table A-1 on page 287.

Table A-6 shows the pseudo result table descriptor for Example A-12.

Table A-6 Pseudo result table descriptor for Example A-12

Item	NAME	TYPE	LENGTH	PRECISION	SCALE
1	EMPNO	CHARACTER	6	0	0
2	FIRSTNME	VARCHAR	12	0	0
3	MIDINIT	CHARACTER	1	0	0
4	LASTNAME	VARCHAR	15	0	0
5	WORKDEPT	CHARACTER	3	0	0
6	DEPTNAME	VARCHAR	36	0	0
6	EDLEVEL	SMALLINT	2	4	0
7	SALARY	DECIMAL	5	9	2

Note about Example A-12 on page 297:

All columns that are listed in the SELECT list of the JOIN query appear in the descriptor.

DESCRIBE PROCEDURE

The DESCRIBE PROCEDURE statement is used to populate a descriptor with information about the result sets that are returned by a called procedure. The DESCRIBE PROCEDURE statement is not a substitute for the DESCRIBE INPUT or OUTPUT statements, which describe the parameters that are defined for the procedure. The DESCRIBE PROCEDURE statement does not provide information about the result set table.

Information, which is provided by this statement, such as the number of result sets, can be used to determine whether additional SQL statements need to be executed to work with the result sets.

Figure A-6 contains the syntax for the DESCRIBE PROCEDURE statement.

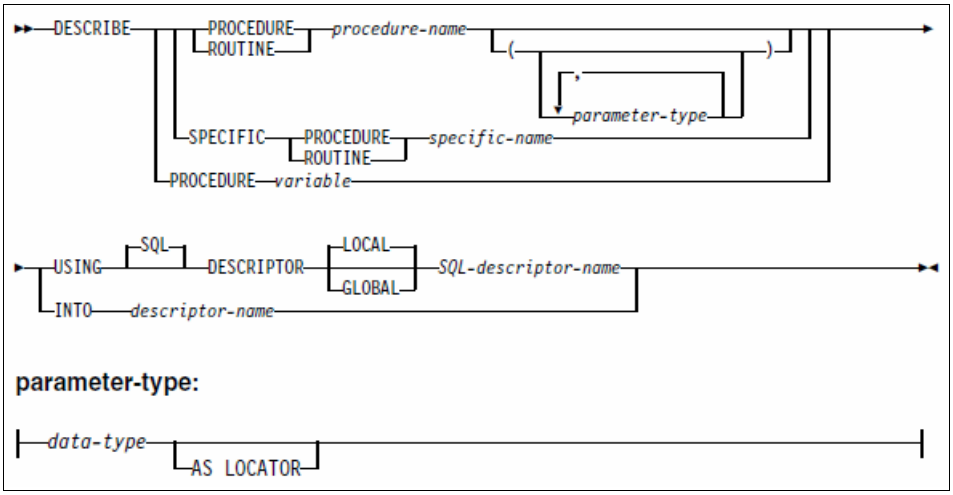

Figure A-6 DESCRIBE PROCEDURE statement syntax

The following information describes several of the parameters in more detail:

- The procedure name can be either the name on the CREATE PROCEDURE or the specific procedure name.
- The procedure name can be contained within a local variable. For variables, the rules that are defined under DESCRIBE CURSOR, apply.
- If the procedure has parameters, each parameter type must be included. For more information about data types, see Chapter 2, “Introduction to SQL Persistent Stored Module” on page 5.

Example A-13 shows a procedure that uses the DESCRIBE PROCEDURE statement.

Example A-13 DESCRIBE_PROCEDURE_DESCRIPTOR procedure

```
CREATE OR REPLACE PROCEDURE DESCRIBE_PROCEDURE_DESCRIPTOR (
   IN p_Global_Descriptor VARCHAR(128) DEFAULT gv_global_descriptor
   ,OUT o_result_sets INTEGER)

P1: BEGIN
   DECLARE v_result_sets INTEGER;                                          1

   CALL PSM_PROC1();                                                       2

   DESCRIBE PROCEDURE PSM_PROC1 USING SQL DESCRIPTOR GLOBAL p_Global_Descriptor;   3

   GET SQL DESCRIPTOR GLOBAL p_Global_Descriptor v_result_sets = COUNT;    4
```

```
   IF v_result_sets > 0 THEN                                                5
      SET o_result_sets = v_result_sets;
   END IF;

END P1
```

Notes about Example A-13 on page 298:

1 An INTEGER variable is declared to hold the number of result sets that are defined in the called procedure.

2 The procedure must be called before the DESCRIBE statement is executed.

3 The DESCRIBE PROCEDURE statement populates the descriptor header with information about the results that are defined in the called procedure.

4 The GET DESCRIPTOR statement is used to return the number of result sets. This statement will be described later in this appendix.

5 An IF control statement is used to check the number of result sets. If a result set is defined, additional work is performed, such as fetching from the associated result set. For a description of result set consumption, see 4.7, “Producing and consuming result sets” on page 91.

Example A-14 shows the SQL CALL statement for the procedure that is shown in Example A-13 on page 298.

Example A-14 CALL DESCRIBE_PROCEDURE_DESCRIPTOR procedure

```
CALL Allocate_Global_Descriptor ('Redbooks');

CALL Describe_Procedure_Descriptor(DEFAULT,0);

Return Code = 0                                                             1
Output Parameter #2 = 1                                                     2
```

Notes about Example A-14:

1 This information was obtained from the log that was generated by the Run SQL Scripts tool, which is provided with System i Navigator. For more information about tools, see Chapter 7, “Development and deployment” on page 189.

2 The returned output parameter, which is shown as a zero (0) on the CALL statement, indicates that one result set is available from the called procedure.

DESCRIBE TABLE

The DESCRIBE TABLE statement is used to populate a descriptor with information about the columns that are defined in a table. The table can be defined by using Data Definition Language (DDL) or IBM i proprietary Data Description Specifications (DDS).

Figure A-7 contains the syntax for the DESCRIBE TABLE SQL statement.

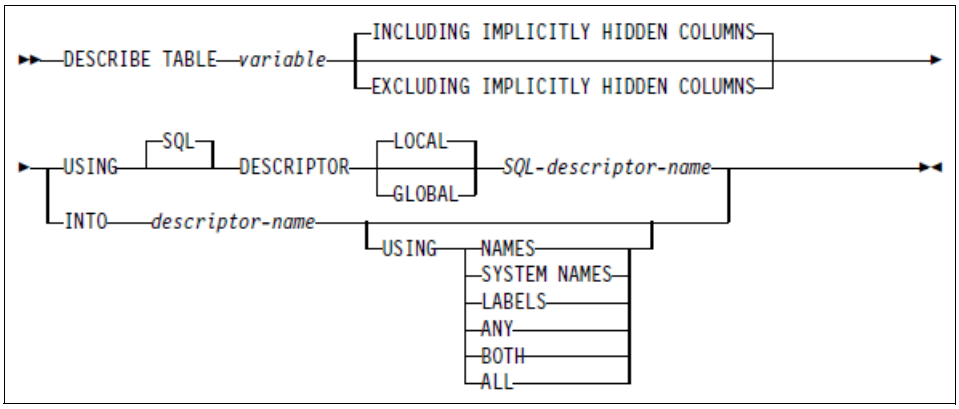

Figure A-7 DESCRIBE TABLE syntax

Minor differences exist between the syntax of the DESCRIBE OUTPUT statement and the syntax of the DESCRIBE TABLE statement. The differences are described:

- TABLE is used instead of OUTPUT.
- The table name must uppercase, or delimited. It must be contained within a variable.
- INCLUDING/EXCLUDING IMPLICITLY HIDDEN COLUMNS is optional with the DESCRIBE TABLE statement.

Example A-15 shows a procedure that uses the DESCRIBE TABLE statement.

Example A-15 DESCRIBE_TABLE_DESCRIPTOR procedure

```
CREATE OR REPLACE PROCEDURE DESCRIBE_TABLE_DESCRIPTOR (
   IN p_qualified_table_name VARCHAR(300)                                  1
   , IN p_Global_Descriptor VARCHAR(128) DEFAULT gv_global_descriptor)

P1: BEGIN
   DECLARE v_qualified_table_name VARCHAR(300);                            2

   SET v_qualified_table_name = UPPER(p_qualified_table_name);             3

   DESCRIBE TABLE v_qualified_table_name
      USING SQL DESCRIPTOR GLOBAL p_Global_Descriptor;                     4

END P1
```

Notes about Example A-15:

1 The required table name parameter is defined large enough to hold a fully qualified table name.

2 A host variable is defined to contain the table name after it is translated to an uppercase table name.

3 The UPPER function is used to translate the table name to ensure that the uppercase requirement is met.

4 The DESCRIBE TABLE populates the global descriptor with the column information of the described table.

Example A-16 shows the SQL CALL statement for the procedure in Example A-15 on page 300.

Example A-16 CALL DESCRIBE_TABLE_DESCRIPTOR procedure

```
CALL Allocate_Global_Descriptor ('Redbooks');

--Describe view
CALL describe_table_descriptor('VEMP_INFO');                          1
CALL describe_table_descriptor('Vemp_Info');                          1
CALL describe_table_descriptor('"Vemp_Info"');                        1
--Describe qualified base table
CALL describe_table_descriptor('DCR_DB2DDL.DEPARTMENT');              2
--Describe DDS PF
CALL describe_table_descriptor('KEYEDPF');                            3
--Describe DDS LF
CALL describe_table_descriptor('KEYEDLF');                            3
```

Notes about Example A-16:

1 These examples describe an SQL view that uses uppercase, lowercase, and delimiters. In all cases, the same descriptor information is created as described in Table on page 297.

2 This example describes a qualified table. The descriptor information is described in Table A-7.

3 These examples describe DDS files. In both cases, the descriptor information is the same as the information that is shown in Table A-8 on page 301.

Table A-7 Pseudo descriptor for database table DEPARTMENT

Item	NAME	TYPE	LENGTH	PRECISION	SCALE
1	DEPTNO	CHARACTER	3	0	0
2	DEPTNAME	VARCHAR	36	0	0
3	MGRNO	CHARACTER	6	0	0
4	ADMRDEPT	CHARACTER	3	0	0
5	LOCATION	CHARACTER	16	0	0

Table A-8 describes the pseudo descriptor for database tables KEYEDPF and KEYEDLF.

Table A-8 Pseudo descriptor for database tables KEYEDPF and KEYEDLF

Item	NAME	TYPE	LENGTH	PRECISION	SCALE
1	ORDID	CHARACTER	6	0	0
2	ORDCUST	VARCHAR	4	0	0
3	ORDQTY	INTEGER	4	9	0
4	ORDAMNT	DECIMAL	4	7	2
5	ORDDESC	CHARACTER	300	0	0

Manipulating data to and from descriptors

After the completion of a successful DESCRIBE or PREPARE...USING statement, the SQL descriptor will contain a header entry that is followed by *n* number of base entries. The base entries will not contain data or indicator values. This data must be assigned before the data is retrieved from the descriptor.

Descriptor data and indicator values can be assigned to an SQL input descriptor by using any of the following statements:

- SQL SET DESCRIPTOR
- CALL... USING SQL DESCRIPTOR (IN and INOUT)

Descriptor data and indicator values can be assigned to an SQL result table descriptor by using any of the following statements:

- SET DESCRIPTOR
- CALL... INTO SQL DESCRIPTOR (INOUT and OUT)
- FETCH... INTO SQL DESCRIPTOR (single)
- FETCH... USING SQL DESCRIPTOR... INTO row-storage-area (multiple)

Descriptor data and indicator values can be retrieved from an SQL result table descriptor by using the GET DESCRIPTOR statement.

SET SQL DESCRIPTOR

The SET DESCRIPTOR statement is used to alter the information that is contained within the items that are defined in the descriptor header information structure and the individual occurrences of array structure entries.

Figure A-8 contains the syntax diagram for the SET SQL DESCRIPTOR statement.

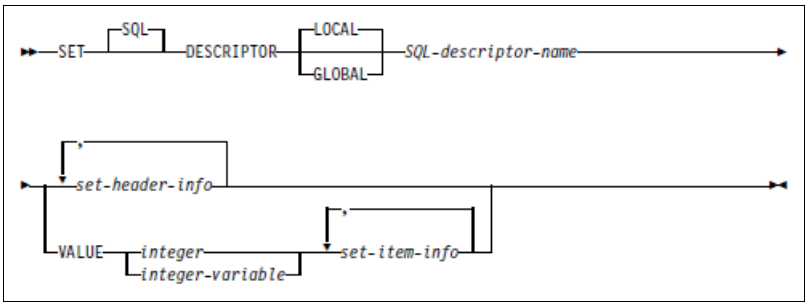

Figure A-8 SETSQL DESCRIPTOR syntax

The following information describes several of the parameters in more detail:

- You must allocate the SQL descriptor *before* you use the SET statement.
- Set-header-info is described in more detail later.
- VALUE is an optional numeric literal or host variable that specifies the item that is being altered. The value must be greater than 1 but less than the maximum number of items that are allocated for the descriptor.
- Either a valid header variable or VALUE must be specified.
- Set-item-info is described in more detail later.

Set-header-info

Set-header-info refers to the variables that are contained within the header portion of the SQL descriptor.

Figure A-9 contains the syntax diagram for the set-header-info clause.

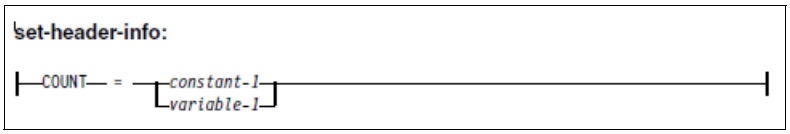

Figure A-9 SET DESCRIPTOR set-header-info clause syntax

COUNT is the only header variable that can be altered by using the SET SQL DESCRIPTOR statement. It must be set before you use an input descriptor on a CALL, OPEN, or EXECUTE statement. The value for count must be numeric, and it can be a literal or local variable. The value must be greater than 0 and not greater than 8000.

Example A-5 on page 291 shows a procedure that uses the SET SQL DESCRIPTOR statement to set the count for a global descriptor.

Set-item-info

Set-item-info refers to the attributes of a specific column or parameter marker within the SQL descriptor. A separate SET DESCRIPTOR statement must be used to set the attributes of each item in the item array. Each item or attribute can only be specified one time in a single SET DESCRIPTOR statement.

Figure A-10 contains the syntax diagram for the set-item-info clause.

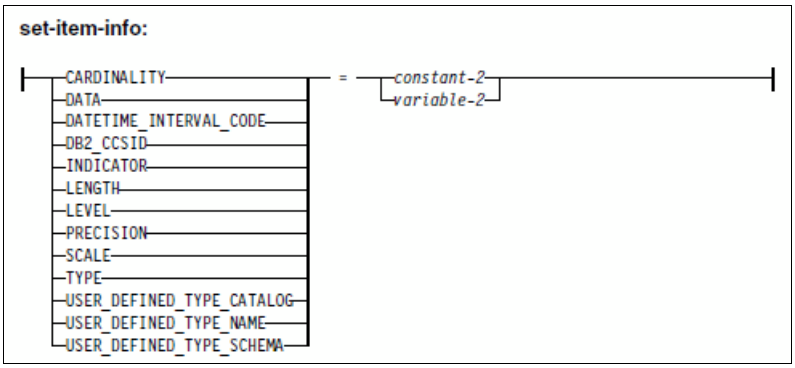

Figure A-10 SET DESCRIPTOR set item info syntax

The following information describes several of the parameters in more detail:

- CARDINALITY specifies the cardinality for the item. Cardinality is only allowed when TYPE is an array. For more information about arrays, see Chapter 4, “Procedures” on page 73.
- DATA Specifies the data value for the item. The data must be a local variable that matches the TYPE for this item.
- DATETIME_INTERVAL_CODE specifies the specific datetime data type. DATETIME_INTERVAL_CODE must be specified if TYPE is set to 9. The following list shows the valid datetime codes:
  - 1 DATE
  - 2 TIME
  - 3 TIMESTAMP

- DB2_CCSID specifies the coded character set identifier (CCSID) of character, graphic, XML, or datetime data. The value is not applicable for all other data types. If the DB2_CCSID is not specified or 0 is specified, the following situations occur:
  - For XML data, the SQL_XML_DATA_CCSID QAQQINI option setting is used.
  - Otherwise, the CCSID of the variable is determined by the CCSID of the job.
- INDICATOR: When extended indicator variables are not enabled, a negative value indicates that the value that is described by this descriptor item is the null value. If INDICATOR is not set, the value of INDICATOR is 0. When extended indicator variables are enabled, the following situations occur:
  - The -1, -2, -3, -4, or -6 indicates that the value that is described by this descriptor item is the null value.
  - The -5 indicates that the value that is described by this descriptor item is the DEFAULT value.
  - The -7 indicates that the value that is described by this descriptor item is the UNASSIGNED value.
  - A 0 or a positive value indicates that a DATA value will be provided for this descriptor item.

  For examples that use extended indicators, see Chapter 8, “Creating flexible and reusable procedures” on page 227.
- LENGTH specifies the maximum length of the data. If the data type is a character or graphic string type, XML type, or a datetime type, the length represents the *number of characters* (not bytes). If the data type is a binary string or any other type, the length represents the *number of bytes*. If LENGTH is not specified, a default length is used. For a description of the defaults, see Table 1 of the SET DESCRIPTOR statement in the DB2 for i SQL reference at this website:

  `https://ibm.biz/Bd4Pex`
- LEVEL specifies the level of the item descriptor:
  - 0: Item is a primary descriptor entry.
  - 1: Item is for a secondary descriptor entry. This level is for an array entry.
- PRECISION specifies the precision for descriptor items of data type DECIMAL, NUMERIC, DECFLOAT, DOUBLE, REAL, FLOAT, and TIMESTAMP. If PRECISION is not specified, a default precision is used.
- SCALE specifies the scale items of data type DECIMAL or NUMERIC. If SCALE is not specified, a default scale is used.
- TYPE specifies a data type code that represents the data type of the descriptor item. A valid type code must be specified for each descriptor item. To see a list of the data type codes and lengths for the GET DESCRIPTOR statement, see Table 2 in the DB2 for i SQL reference at this website:

  `https://ibm.biz/Bd4PeX`

For more information about these and other items, see the DB2 for i SQL reference at this website:

`https://ibm.biz/Bd42dh`

Example A-17 contains the source code for a procedure that dynamically constructs a WHERE clause based on the values of the search arguments that are passed to the procedure. The size of the input descriptor is based on the number of parameter markers that are defined in the completed WHERE clause.

Example A-17 Flexible_Selection_Using_Descriptors procedure

```
CREATE OR REPLACE PROCEDURE Flexible_Selection_Using_Descriptors (
   IN p_Global_Descriptor VARCHAR(128) DEFAULT gv_Global_Descriptor                1
   , p_sqlString CLOB (2M)DEFAULT 'SELECT * FROM vemp_info'                       2
   ,IN p_edlevel SMALLINT DEFAULT NULL                                            3
   ,IN p_workdept CHAR(3) DEFAULT NULL)                                           3

   RESULT SETS 1
P1: BEGIN
   DECLARE example224_c1 CURSOR FOR example224_s1;                                4
   --Procedure logic begins here
   PREPARE example224_s1 FROM p_sqlString ;                                       5

   DESCRIBE INPUT example224_s1 USING SQL DESCRIPTOR GLOBAL p_Global_Descriptor;  6

   If p_edlevel IS NOT NULL AND p_workdept IS NULL THEN
      SET SQL DESCRIPTOR GLOBAL p_Global_Descriptor VALUE 1                       7
          TYPE = 5, DATA = P_EDLEVEL , INDICATOR = 0 ;
   ELSEIF p_edlevel IS NULL AND p_workdept IS NOT NULL THEN
      SET SQL DESCRIPTOR GLOBAL p_Global_Descriptor  VALUE 1                      7
         TYPE = 1, DATA = P_WORKDEPT , LENGTH = 3, INDICATOR = 0 ;
   ELSEIF p_edlevel IS NOT NULL AND p_workdept IS NOT NULL THEN
      SET SQL DESCRIPTOR GLOBAL p_Global_Descriptor VALUE 1                       7
         TYPE = 5, DATA = P_EDLEVEL , INDICATOR = 0 ;
      SET SQL DESCRIPTOR GLOBAL p_Global_Descriptor  VALUE 2                      7
         TYPE = 1, DATA = P_WORKDEPT , LENGTH = 3,  INDICATOR = 0 ;
   END IF;

   OPEN example224_c1 USING SQL DESCRIPTOR GLOBAL p_Global_Descriptor;            8
   END IF;
END P1
```

Notes on Example A-17 on page 305:

1 A valid SQL global descriptor must be allocated before the procedure is called.

2 If the SQL string is not passed, a default string is used, which allows the user to pass an SQL select statement that contains any table or view that references the selection parameters. For examples of calling this procedure, see Example A-18 on page 307.

3 The selection parameters, p_edlevel and p_workdept, are defined with the default, NULL. Therefore, either one, both, or none of the selection parameters can be passed to the procedure.

4 The cursor example224_c1 is declared, referencing the prepared statement example224_s1. Only one cursor is required for any and all combinations of selection criteria.

5 The statement example224_s1 is prepared based on the passed string. No string construction occurs within the program.

6 After the successful completion of the DESCRIBE INPUT SQL statement, the global descriptor header count is set to the number of parameter markers in the prepared statement. The base entries of the global descriptor contain the information for the columns, if any, that are referenced on the WHERE clause of the SQL statement. If the MAX ITEMS value of the descriptor is smaller than the number of parameter markers that are contained in the prepared statement, an error occurs.

7 The SET DESCRIPTOR is used to assign the value of the input parameter to the appropriate descriptor base item:

- If only a single selection parameter was passed, the DATA item of the first base item structure is assigned the selection parameter value. The LENGTH item is not specified for the numeric selection parameter. However, the LENGTH item is set for the character selection variable.
- If both selection parameters were passed, the DATA item of the first occurrence of the item structure is set with the first selection parameter value, and the DATA item of the second occurrence of the item structure is set with the second selection parameter value.

8 The cursor is opened by using the global descriptor. If the global descriptor header COUNT is 0 (no parameters), the global descriptor is ignored and the cursor is opened.

Example A-18 shows several SQL CALL statements that are used to run the FLEXIBLE_SELECTION_USING_DESCRIPTORS procedure. Each run produces a different result set based on various search arguments that are provided by using a flexible WHERE clause. For more information about flexible procedures, see Chapter 8, “Creating flexible and reusable procedures” on page 227.

Example A-18 CALL Flexible_Selection_Using_Descriptors procedure

```
CALL Allocate_Global_Descriptor ('Flexible_Selection');                          1

CALL Flexible_Selection_Using_Descriptors();                                     2

CALL Flexible_Selection_Using_Descriptors(
   DEFAULT,'SELECT * FROM vemp_info WHERE edlevel = ?', 16);                     3

CALL Flexible_Selection_Using_Descriptors(
   DEFAULT,'SELECT * FROM vemp_info WHERE workdept = ?', DEFAULT, 'E21');        4

CALL Flexible_Selection_Using_Descriptors(
   DEFAULT,'SELECT * FROM vemp_info WHERE edlevel = ? AND workdept = ?', 16, 'E21');  5
```

Notes about Example A-18:

1 The allocate_global_descriptor procedure is called to allocate a descriptor with a default of 20 entries.

2 The flexible_selection_using_descriptors procedure is called without any parameters.

3 The flexible_selection_using_descriptors procedure is called and passes an SQL string that contains a WHERE clause with one of the following options:

- A single search argument and a single value: edlevel (3)
- A single search argument, the DEFAULT (NULL) for edlevel, and a value for workdept (4).
- Two search arguments, edlevel and workdept (5).

The result set that is produced by calling the Flexible_Selection_Using_Descriptors() procedure is shown in Figure A-11.

```
EMPNO  FIRSTNME  MIDINIT LASTNAME   WORKDEPT EDLEVEL SALARY
------ --------- ------- ---------- -------- ------- --------
000010 CHRISTINE I       HAAS       A00           18 52750.00
000020 MICHAEL   L       THOMPSON   B01           18 41250.00
000030 SALLY     A       KWAN       C01           20 38250.00
000050 JOHN      B       GEYER      E01           16 40175.00
000060 IRVING    F       STERN      D11           16 32250.00
000070 EVA       D       PULASKI    D21           16 36170.00
000090 EILEEN    W       HENDERSON  E11           16 29750.00
000100 THEODORE  Q       SPENSER    E21           14 26150.00
000110 VINCENZO  G       LUCCHESSI  A00           19 46500.00
000120 SEAN              O'CONNELL  A00           14 29250.00
000130 DELORES   M       QUINTANA   C01           16 23800.00
000140 HEATHER   A       NICHOLLS   C01           18 28420.00
```

Figure A-11 Result table for all rows (only first 15 rows shown)

The result set that was produced by calling the Flexible_Selection_Using_Descriptors procedure and passing the following parameters (`DEFAULT, 'SELECT * FROM vemp_info WHERE edlevel =?', 16`) is shown in Figure A-12.

```
EMPNO  FIRSTNME  MIDINIT LASTNAME  WORKDEPT EDLEVEL SALARY
------ --------- ------- --------- -------- ------- --------
000050 JOHN      B       GEYER     E01           16 40175.00
000060 IRVING    F       STERN     D11           16 32250.00
000070 EVA       D       PULASKI   D21           16 36170.00
000090 EILEEN    W       HENDERSON E11           16 29750.00
000130 DELORES   M       QUINTANA  C01           16 23800.00
000150 BRUCE             ADAMSON   D11           16 25280.00
000170 MASATOSHI J       YOSHIMURA D11           16 24680.00
000190 JAMES     H       WALKER    D11           16 20450.00
000200 DAVID             BROWN     D11           16 27740.00
000260 SYBIL     P       JOHNSON   D21           16 17250.00
000320 RAMLAL    V       MEHTA     E21           16 19950.00
000340 JASON     R       GOUNOT    E21           16 23840.00
200170 KIYOSHI           YAMAMOTO  D11           16 24680.00
200340 ROY       R       ALONZO    E21           16 23840.00
```

Figure A-12 Result table for WHERE edlevel = 16

The result set that was produced by calling the Flexible_Selection_Using_Descriptors procedure and passing the following parameters (`DEFAULT,'SELECT * FROM vemp_info WHERE workdept =?', DEFAULT, 'E21'`) is shown in Figure A-13.

```
EMPNO  FIRSTNME MIDINIT LASTNAME WORKDEPT EDLEVEL SALARY
------ -------- ------- -------- -------- ------- --------
000100 THEODORE Q       SPENSER  E21           14 26150.00
000320 RAMLAL   V       MEHTA    E21           16 19950.00
000330 WING             LEE      E21           14 25370.00
000340 JASON    R       GOUNOT   E21           16 23840.00
200330 HELENA           WONG     E21           14 25370.00
200340 ROY      R       ALONZO   E21           16 23840.00
```

Figure A-13 Result table for WHERE workdept = 'E21'

The result set that was produced by calling the Flexible_Selection_Using_Descriptors procedure and passing the following parameters (`DEFAULT,'SELECT * FROM vemp_info WHERE edlevel =? AND workdept =?', 16, 'E21'`) is shown in Figure A-14.

```
EMPNO  FIRSTNME MIDINIT LASTNAME WORKDEPT EDLEVEL SALARY
------ -------- ------- -------- -------- ------- --------
000320 RAMLAL   V       MEHTA    E21           16 19950.00
000340 JASON    R       GOUNOT   E21           16 23840.00
200340 ROY      R       ALONZO   E21           16 23840.00
```

Figure A-14 Result table for WHERE edlevel = 16 AND workdept = 'E21'

GET SQL DESCRIPTOR

The GET DESCRIPTOR statement extracts the information that is contained within the items that are defined in the descriptor header information structure and the individual occurrences of column/parameter array structure entries and places them into the declared local variables of an SQL procedure.

Figure A-15 contains the syntax diagram for the GET SQL DESCRIPTOR statement.

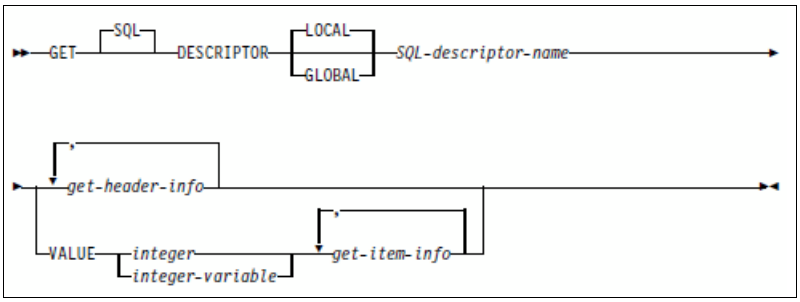

Figure A-15 GET DESCRIPTOR syntax diagram

The following information describes several of the parameters in more detail:

- You must allocate the SQL descriptor *before* you use the GET statement.
- Get-header-info is described in more detail later.
- VALUE is an optional numeric literal or local variable that specifies the item that is being altered. The value must be greater than 1 but less than the maximum number of items that are allocated for the descriptor.
- Either a valid header variable or VALUE must be specified.
- Get-item-info is described in more detail later.

Get-header-info

Figure A-16 contains the syntax diagram for the get-header-info clause.

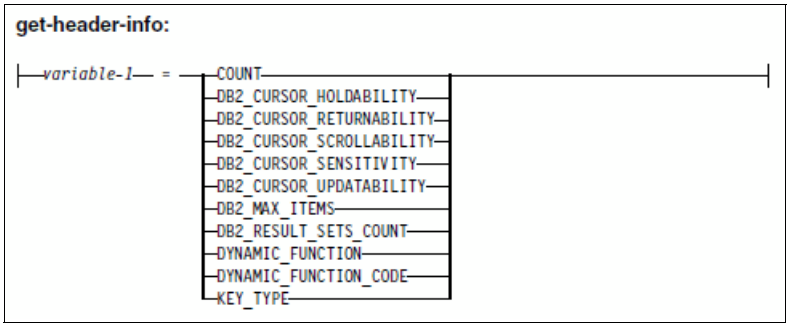

Figure A-16 GET DESCRIPTOR get header info clause

The following information describes several of the parameters in more detail:

- The variable-1 identifies a local variable that is declared in the procedure. The data type of the variable must be compatible with the descriptor information item. For more information, see Table 1 Data Types for GET DESCRIPTOR Items of the GET DESCRIPTOR statement in the DB2 for i SQL reference:

  https://ibm.biz/Bd4PgQ

- COUNT returns the current number of items that are described in the item host structure array. The local variable that is defined to contain the COUNT item must be declared as INTEGER.

For more information, see the GET DESCRIPTOR statement in the DB2 for i SQL reference:

https://ibm.biz/Bd4PgQ

Get-item-info

Figure A-17 contains the syntax diagram for the get-item-info clause.

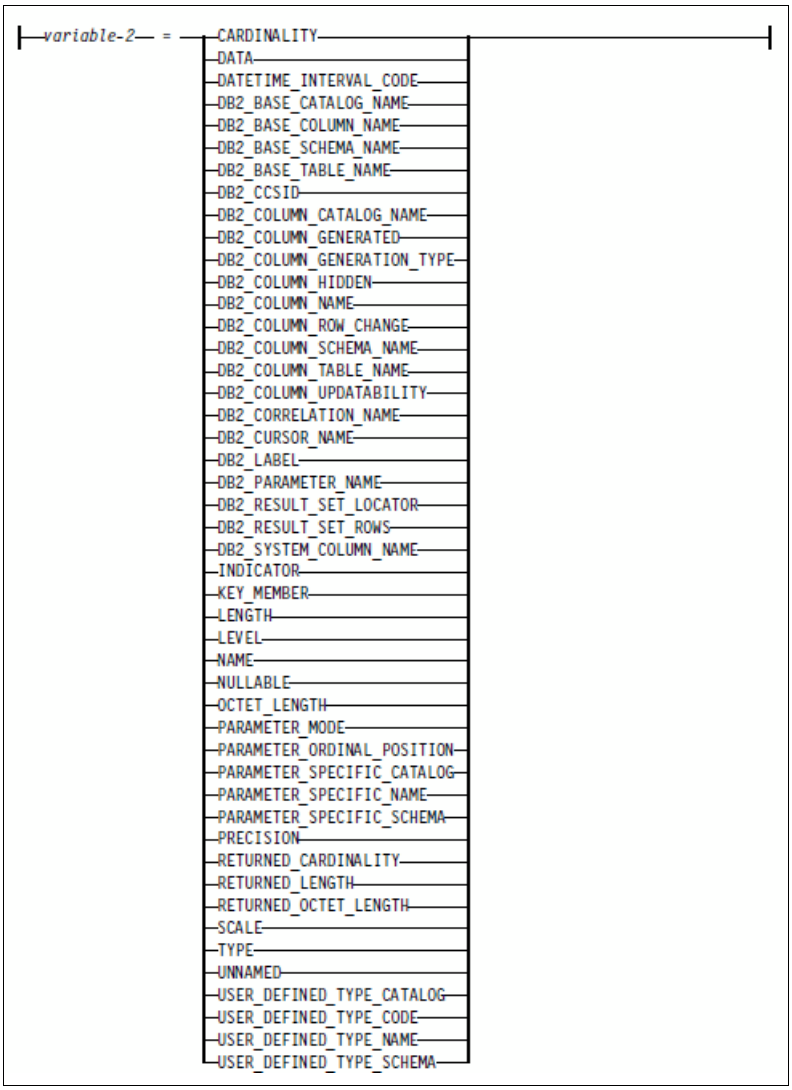

Figure A-17 GET DESCRIPTOR get item info clause

For more information about the items that are contained in this clause, see the GET DESCRIPTOR statement in the DB2 for i SQL reference:

https://ibm.biz/Bd4PgQ

Example A-19 shows a procedure that returns the data from a descriptor by dynamically constructing a “table-less” statement that uses the GET DESCRIPTOR statement to assign values.

Example A-19 Procedure that returns data from a descriptor

```
CREATE OR REPLACE PROCEDURE List_descriptor (
   IN p_Global_Descriptor VARCHAR(128) DEFAULT gv_global_descriptor)
   RESULT SETS 1
   LANGUAGE SQL
P1: BEGIN
   --Declare variables                                                          1
   DECLARE VARTYPE , VARLEN , VARPRE , VARSCALE , VARIND, VARCOUNT , I INTEGER ;  2
   DECLARE VARNAME VARCHAR ( 128 ) ;
      DECLARE v_sqString VARCHAR(1000) DEFAULT
         'SELECT 0 value, VARCHAR(128) NAME , 0 TYPE , 0 LENGTH ,0 PRECISION , 0 SCALE
         FROM sysibm.sysdummy1
         UNION
         SELECT * FROM TABLE (VALUES ('                                        3

   GET SQL DESCRIPTOR GLOBAL P_GLOBAL_DESCRIPTOR VARCOUNT = COUNT ;             4
   SET I = 1 ;

   L1 : WHILE ( I <= VARCOUNT ) DO
      GET SQL DESCRIPTOR GLOBAL P_GLOBAL_DESCRIPTOR VALUE I                     5
         VARNAME = NAME , VARTYPE = TYPE , VARLEN = LENGTH ,
         VARPRE = PRECISION , VARSCALE = SCALE ;
      IF VARTYPE = 0 THEN
         LEAVE L1 ;
      END IF ;
         -- Build table less query                                             6
      SET v_sqString = RTRIM(v_sqString)
              CONCAT(i)
              CONCAT(', ') CONCAT(RTRIM(VARNAME))
              CONCAT(', ') CONCAT(VARTYPE)
              CONCAT(', ') CONCAT(VARLEN)
              CONCAT(', ') CONCAT(VARPRE)
              CONCAT(', ') CONCAT(VARSCALE)
              CONCAT(')');
      IF i < VARCOUNT THEN
         -- More than 1 row
         SET v_sqString = RTRIM(v_sqString) CONCAT(', (');
      END IF;
      SET I = I + 1 ;
   END WHILE ;
   -- Complete the tableless query
   SET v_sqString = RTRIM(v_sqString) CONCAT(') AS T1');
   CALL Declare_And_Prepare (RTRIM(v_sqString));                                7
END P1
```

Notes about Example A-19 on page 311:

1. The variables that are required for the assignment of the GET DESCRIPTION items are defined here.
2. One declaration can be used for all variables of the same type.
3. A character large object (CLOB) variable is defined to hold the constructed SQL state. The FROM references TABLE then (VALUES, which indicates that the result table will be based on the values that are provided within the procedure. This statement is referred to as a “table-less” query.
4. The number of descriptor entries is extracted and assigned to local variable VARCOUNT. This variable will be used as the terminator in the WHILE loop.
5. For each item entry (VALUE = i), use the GET DESCRIPTOR statement to assign the items NAME, TYPE, LENGTH, PRECISION, and SCALE to the equivalent variables.
6. As the items are processed, the values for the SQL string are concatenated to the sql string variable.
7. After all items are processed, the sql string is completed and passed to the DECLARE_AND_EXECUTE procedure. This procedure declares an SQL CURSOR based on the sql string and then opens the cursor to return the result set.

Example A-20 shows the SQL CALL statement that is used to run the LIST_DESCRIPTOR procedure and the SQL string that was passed to the DECLARE_AND_PREPARE procedure.

Example A-20 CALL LIST_DESCRIPTOR procedure

```
CALL List_descriptor ();

Constructed SQL string
SELECT * FROM TABLE (VALUES
   (1, 1, 1, 6, 0, 0),
   (2, 2, 12, 15, 0, 0),
   (3, 3, 1, 1, 0, 0),
   (4, 4, 1, 3, 0, 0),
   (5, 5, 5, 2, 4, 0),
   (6, 6, 3, 5, 9, 2)
   ) AS T1;
```

Note: The LIST_DESCRIPTOR procedure was used to produce the pseudo descriptor tables that were used in “How to use SQL descriptors” on page 286.

B

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 associated with this book is available in softcopy on the Internet from the IBM Redbooks web server. Point your web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG248326

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG248326.

Using the web material

The additional web material that accompanies this book includes the following file:

File name	*Description*
SQLScripts.zip	SQL Scripts zipped code samples

System requirements for downloading the web material

The web material requires the following system requirements:

- IBM i 7.2 with the latest technology refresh

Downloading and extracting the web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material .zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *External Procedures, Triggers, and User-Defined Functions on IBM DB2 for i*, SG24-6503
- *IBM DB2 Web Query for i Version 2.1 Implementation Guide*, SG24-8063
- *OnDemand SQL Performance Analysis Simplified on DB2 for i5/OS in V5R4*, SG24-7326
- *Modernizing IBM i Applications from the Database up to the User Interface and Everything in Between*, SG24-8185

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- DB2 for i SQL reference:

  https://ibm.biz/Bd42dh
- SQL programming:

  https://ibm.biz/Bd42dA
- IBM i Knowledge Center:

  http://www.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome
- IBM DB2 for i Wiki:

  https://ibm.biz/Bd4fTH
- IBM Data Studio:

  https://ibm.biz/Bd4qbU
- IBM i Access Client Solutions:

  https://ibm.biz/Bd4q8R
- *IBM Data Studio debugger and IBM DB2 for i*, by Kent Milligan:

  https://ibm.biz/Bd4q8V

- IBM i Run SQL Scripts graphical debugger:
  - https://ibm.biz/Bd4qhL
  - https://ibm.biz/Bd4qh3
- DB2 for i Services are documented in the IBM Knowledge Center:

  http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/rzajq/rzajqservicesdb2.htm
- IBM i Services are documented in the IBM Knowledge Center:

  http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/rzajq/rzajqservicessys.htm
- IBM DB2 for i Wiki services:

  https://ibm.biz/Bd4yEL
- “Database Performance and Query Optimization”:

  https://ibm.biz/Bd42Jk

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Redbooks

SQL Procedures, Triggers, and Functions on IBM DB2 for i

SG24-8326-00
ISBN 0738441643

IBM

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

SG24-8326-00

ISBN 0738441643

Printed in U.S.A.