Oracle Rdb7™

Guide to SQL Programming

Release 7.0

Part No. A42867-1

ORACLE®

Guide to SQL Programming

Release 7.0

Part No. A42867-1

Contents

3 Introduction to SQL Module Language

4 Writing Module SQL Procedures

5 Processing SQL Modules and Host Language Files

6 Using Precompiled SQL

7 Creating Images for Program Execution

8 Declaring and Using Parameters

9 Using Date-Time Data Types

Part III Run-Time Processing

10 Handling Run-Time Errors

13 Using Stored Routines

14 Using External Routines

Part V Your Program's Context

15 Attaching to Databases

16 Managing Transaction Context

17 Managing Multiple Connections in Programs

Part VI Data Manipulation in Programs

18 Using Cursors

19 Inserting, Updating, and Deleting Data

20 Using the Multiple Schema Option

A Using SQL International Options

Index

Examples

Figures

Tables

Send Us Your Comments

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

You can send comments to us in the following ways:

- Electronic mail — nedc_doc@us.oracle.com
- FAX — 603-897-3334 Attn: Oracle Rdb Documentation
- Postal service

  Oracle Corporation
  Oracle Rdb Documentation
  One Oracle Drive
  Nashua, NH 03062
  USA

If you like, you can use the following questionnaire to give us feedback. (Edit the online release notes file, extract a copy of this questionnaire, and send it to us.)

Name _____ Title _____

Company _____ Department _____

Mailing Address _____ Telephone Number _____

Book Title _____ Version Number _____

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?

- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available).

Preface

About This Manual

This manual describes how to design and develop host language application programs that use SQL (structured query language) to store, modify, and retrieve data from Oracle Rdb databases.

Most businesses need to carefully manage information to remain viable. They must find a way to collect, store, and process the information essential to their businesses. They must ensure a way to keep data secure, consistent, and up-to-date. These days, more and more companies are turning to industry-standard database software tools, such as SQL, to provide such support.

Intended Audience

This manual is intended for programmers who write and maintain database applications. To profit fully from this manual, you should already understand the following:

- Programming in one or more host languages
- SQL language syntax
- Basic concepts and terminology of relational database management systems

If you are unfamiliar with SQL, you should begin by reading the *Oracle Rdb7 Introduction to SQL* before proceeding. That companion manual introduces you to the SQL interface.

How This Manual Is Organized

In this manual, chapters are grouped by category to make accessing information easier. Brief descriptions of each chapter are shown in the following table:

SQL Programming Overview

Chapter 1	Introduces the SQL programming interfaces.

Developing Application Programs That Use SQL

Chapter 2	Provides general information about developing host language programs and describes methods for including SQL statements in programs.
Chapter 3	Introduces the SQL module processor.
Chapter 4	Describes the common elements in writing a procedure in SQL module language and explains how to write single-statement and multistatement SQL module procedures.
Chapter 5	Describes how to process SQL modules and host language files to create an executable image.
Chapter 6	Describes how to embed SQL statements in host language source files and how to process those files using the SQL precompiler.
Chapter 7	Describes how to create an executable image by linking object modules and how to debug and run your program.
Chapter 8	Provides information on declaring and using parameters in host language source files.
Chapter 9	Describes how to use date-time data types in programs.

Run-Time Processing

Chapter 10	Describes how to detect run-time errors, retrieve error messages, and either recover from errors or roll back a transaction.
Chapter 11	Describes how to use dynamic SQL.

Programmatic Structures

Chapter 12	Describes how to use compound statements.
Chapter 13	Describes how to create and use stored procedures and functions.
Chapter 14	Describes how to create and use external procedures and functions.

Your Program's Context

Chapter 15	Describes how to attach to and detach from databases, including databases that reside on remote nodes.
Chapter 16	Describes how to specify and start a transaction and discusses options for data access.
Chapter 17	Introduces the concept of SQL connections for use in querying, testing, and prototyping programs.

Data Manipulation in Programs

Chapter 18	Describes table and list cursors and how to use cursors to retrieve data.
Chapter 19	Describes how to insert, update, and delete data in a database.
Chapter 20	Describes how to write programs for multischema databases.
Appendix A	Describes the SQL options for handling international data that is not in English and provides information about collating sequences used by Oracle Rdb.

Related Manuals

For more information on Oracle Rdb, see the other manuals in the documentation set, especially the following:

- *Oracle Rdb7 Introduction to SQL*
- *Oracle Rdb7 SQL Reference Manual*

Refer to the *Oracle Rdb7 Release Notes* for descriptions of all manuals in the Oracle Rdb documentation set.

SQL Standards

SQL is both a data definition (DDL) and data manipulation (DML) language for relational databases. Using the SQL interface, you can create a database, load it with data, and read and update both data and data definitions. The SQL interface to Oracle Rdb conforms to the entry-level of the SQL standard ANSI X3.135-1992, ISO 9075:1992, commonly referred to as the ANSI/ISO SQL standard or SQL92.

Conventions

In this manual, Oracle Rdb refers to Oracle Rdb for OpenVMS and Oracle Rdb for Digital UNIX software. Version 7.0 of Oracle Rdb software is often referred to as V7.0.

Oracle CDD/Repository software is referred to as the dictionary, the data dictionary, or the repository.

OpenVMS means both the OpenVMS Alpha and OpenVMS VAX operating system.

This manual uses icons to identify information that is specific to an operating system or platform. Where material pertains to more than one platform or operating system, combination icons or generic icons are used. For example:

	This icon denotes the beginning of information specific to the Digital UNIX operating system.
	This icon combination denotes the beginning of information specific to both the OpenVMS VAX and OpenVMS Alpha operating systems.
♦	The diamond symbol denotes the end of a section of information specific to an operating system or platform.

In examples, an implied carriage return occurs at the end of each line. You must press the Return key at the end of a line of input.

Examples do not always include prompts. Generally, prompts are shown when depicting interactive sequences exactly; otherwise, they are omitted.

The following conventions are also used in this manual:

e, f, t	Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page.
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
...	Horizontal ellipsis points in statements mean that parts of the statement not directly related to the example have been omitted.
boldface text	Boldface type in text indicates a term defined in the text.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
$	The dollar sign represents the DIGITAL Command Language prompt in OpenVMS and the Bourne shell prompt in Digital UNIX.

Technical Changes and New Features

This section identifies the new and updated portions of this manual since it was last released with Oracle Rdb Version 6.0.

The major V7.0 new features and technical changes described in this manual include the following:

- Holding cursors open across transactions

  SQL cursors can now remain open across transaction boundaries. The WITH HOLD clause of the DECLARE CURSOR statement indicates that the cursor will remain open after the transaction ends. A cursor that has been held open retains its position when a new SQL transaction begins. For more information, see Section 18.5.

- Creating stored functions

  In addition to defining stored procedures, you can now define stored functions. A **stored function** is a set of operations performed on an Oracle Rdb database by one or more SQL statements. It accepts a set of input parameters and returns a single result. You invoke a stored function by using the function name in a value expression. For more information, see Chapter 13.

- Returning the value of a stored function

  SQL provides the RETURN statement, which returns the result of a stored function. For more information, see Section 13.3.2.

- Using the CALL statement in a compound statement

  You can now use the CALL statement within a compound statement. As a result, you can use it in a stored procedure or function to call another stored procedure. You can also use the CALL statement to invoke external procedures. For more information, see Section 12.3.8.

- Using the SIGNAL statement in a compound statement

SQL now provides the SIGNAL statement for use within a compound statement. SIGNAL accepts a single character value expression that is used as the SQLSTATE. The current routine and all calling routines are terminated and the signaled SQLSTATE is passed to the application. For more information, see Section 12.9.

- Using external procedures and calling into the database with external routines

  SQL now provides external procedures and it also allows external routines to contain SQL statements, letting you bind to new schema instances and perform database operations from the external routine. **External routines** are external functions or external procedures written in a 3GL language such as C or FORTRAN, linked into a shareable image or shared module, and registered in a database schema. For more information, see Chapter 14.

- Cascading delete for modules

  DROP MODULE CASCADE lets you drop a module and any objects that refer to it. For more information, see Section 13.6.

- Dropping functions and procedures

  You can now drop external procedures and stored functions and procedures. For more information, see Section 13.6 and Section 14.4.

- Specifying the DEFAULT, CONSTANT, and UPDATABLE clauses when declaring variables within compound statements

  You can specify the default value of a variable to be any value expression including subqueries, conditional, character, date-time, and numeric expressions. Additionally, the variable can now inherit the default from the named domain. For more information, see Section 12.3.1.

- Using a header file to eliminate Digital C informational messages

  SQL provides a header file that eliminates informational messages by providing prototypes for explicitly called SQL routines. For more information, see Section 6.7.3.

- Using the sql_sqlda.h header file with C programs

  You can now use the sql_sqlda.h header file in C language programs to obtain definitions of the SQLDA and SQLDA2 structures. Previously, you could only obtain definitions of these structures by using the SQL precompiler statement EXEC SQL INCLUDE SQLDA or EXEC SQL INCLUDE SQLDA2. For more information, see Section 11.3.2.

- Declaring external references to the SQLCA structure

You can now declare an external reference to the SQLCA structure when you use the SQL precompiler with the C language; use the optional EXTERNAL keyword at the end of your EXEC SQL INCLUDE SQLCA statement. For more information, see Section 8.6.2.

- Setting debug flags using SQL

  SQL supports a new SET FLAGS statement for interactive and dynamic SQL. The SET FLAGS statement lets you enable and disable the database system's debug flags during execution, including letting you monitor the contents of variables in compound statements. For more information, see Section 12.7.

- Using the SQL_ALTERNATE_SERVICE_NAME configuration parameter

  The configuration parameter SQL_ALTERNATE_SERVICE_NAME lets you specify an alternate TCP/IP service. This is especially useful for accessing different versions of OpenVMS databases through TCP/IP from an OpenVMS or Digital UNIX client. For more information, see Section 15.2.6.

The major V6.1 new features and technical changes described in this manual include the following:

- Using SQL on a Digital UNIX system

  Information about using SQL on a Digital UNIX system, including compiling and linking on Digital UNIX, is now described in this manual.

- Authenticating users for remote access

  Oracle Rdb lets you explicitly provide user name and password information in SQL statements that attach to the database. In addition, it lets you pass the information to an SQL module language or precompiled SQL program by using a parameter and new command line qualifiers. You can also pass the information to Oracle Rdb using configuration parameters. For more information, see Chapter 15.

- Modifying the INTEGER data type for SQL module language

  The SQL module language syntax has been extended to allow specification of precise INTEGER module parameters in the number of bytes. See Section 8.12.2 for more information.

- Using portable SQL routines

  SQL provides the following routines for use on both OpenVMS and Digital UNIX operating systems:

  - sql_get_error_text

This routine passes error text with formatted output to programs for processing. It is similar to the SQL$GET_ERROR_TEXT routine, which is available only on OpenVMS systems. For more information, see Section 10.3.2.

- sql_get_message_vector

  This routine retrieves information from the message vector about the status of the last SQL statement. For more information, see Section 10.2.4.

- sql_get_error_handler, sql_register_error_handler, and sql_deregister_error_handler

  These routines now work on Digital UNIX, but otherwise have not changed from previous versions of Oracle Rdb. For more information, see Section 10.2.5.

- sql_signal

  This routine signals that an error has occurred during the execution of an SQL statement. It is equivalent to the SQL$SIGNAL routine, which is available only on OpenVMS systems. For more information, see Section 10.3.1.

In addition to the Oracle Rdb technical changes and new features reflected in this manual, modifications were made to clarify or correct the documentation.

Part I

SQL Programming Overview

This part presents a high-level view of how to use SQL in your application programs.

1
Introduction to SQL Programming

This chapter introduces you to programming with the SQL interface to Oracle Rdb. In the sections that follow, you will become familiar with:

- The programming interfaces that are available
- The advantages of SQL module language over precompiled SQL
- How to find sample programs that illustrate the main features of each SQL interface

To support complex database demands, Oracle Rdb provides two programming interfaces, the SQL module processor and the SQL precompiler. With these programming interfaces, you can develop application programs that write formatted reports, perform complex calculations, or update data.

1.1 What Are the Two SQL Programming Interfaces?

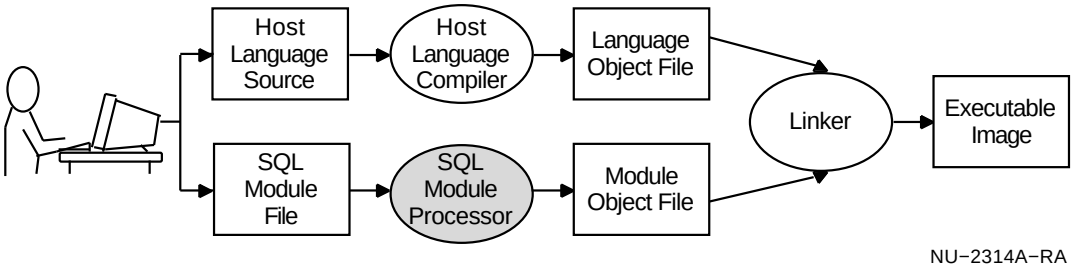
There are two ways to combine SQL with host language programs. You can create a separate module for SQL language statements or you can enter them directly in the host language program:

- **SQL module processor** compiles a module file containing SQL statements to create an object file. You link the module object file with a host language program object file to produce an executable image.
- **SQL precompiler** converts SQL statements that you embed in host language programs along with the program code into a form host language program compilers can process.

1.1.1 SQL Module Processor

For any supported host language, you can use SQL module language to create an SQL module file of SQL procedures. In your host language source file, you can specify calls to these SQL procedures. The steps you take to create a program that uses SQL module language are illustrated in Figure 1–1.

Figure 1–1 Using the SQL Module Processor in Program Development



In this process, you create both a host language source file and an SQL module file and process them in parallel. You compile the host program with the host compiler and the SQL module file with the SQL module processor. The host language compilation produces a host language object file, and the SQL module compilation generates an SQL object file. You link the two object files together to produce an executable image.

SQL module files contain one or more SQL procedures; each procedure can contain one or more SQL statements. The SQL module processor checks the syntax and semantics of the SQL module file source code and compiles the SQL single-statement and multistatement procedures.

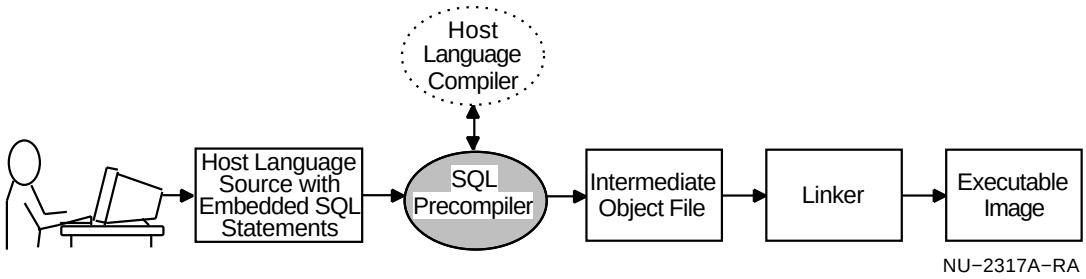
With the SQL module processor, you can perform additional tasks, including:

- Defining various module characteristics, such as the module name, the user authorization id, and the module's level of optimization
- Controlling output information, such as creating a listing file for error messages
- Flagging deprecated or nonstandard SQL syntax

1.1.2 SQL Precompiler

If your host language is supported by the SQL precompiler, you can embed SQL statements directly in a host language source file. Figure 1–2 shows how to use the SQL precompiler to generate programs that use SQL statements.

Figure 1–2 Using the SQL Precompiler in Program Development



You process your program embedded with SQL statements by running the SQL precompiler. The SQL precompiler checks the syntax and semantics of the embedded SQL statements, compiles the SQL syntax and invokes the host language compiler to create an object module. You link the object module to create an executable image.

With the SQL precompiler, you can perform additional tasks, including:

- Controlling output information, such as creating a listing file for error messages
- Flagging deprecated or nonstandard SQL syntax
- Specifying whether the SQL precompiler assigns a G-floating or a D-floating interpretation to the DOUBLE data type.

1.2 Choosing a Programming Interface

Oracle Rdb recommends that you use the SQL module processor instead of the SQL precompiler to develop your application programs. Oracle Rdb provides the SQL precompiler for compatibility with existing applications and for programming environments that require the use of a precompiler.

The SQL module processor is more powerful, flexible, and efficient and has many other advantages including:

- Cleaner abstraction of languages

The host language source program contains only host language statements because the SQL statements are isolated in separate modules.

- Programs written in languages for which there is an ANSI/ISO standard can avoid embedding code that does not conform to the standard by isolating SQL statements in SQL modules. [1]
- Improved modularity and shareability

  Because the SQL module language is isolated in separate modules, you can use different host language programs to call the SQL procedures, improving the flexibility and maintenance of application programs.

  For example, you can use one module that performs a complex database transaction in different application programs. Or, you can call the procedures in this module from different application programs written in different host languages.
- Better integration with CASE tools

  The SQL module processor integrates better with computer-aided software engineering (CASE) tools. Generally, it is easier to store, maintain, and share source and object code created by the SQL module processor.

  In contrast, the SQL precompiler modifies the host language source code so you cannot fully exploit, for example, a language-sensitive editor or a symbolic debugger.
- Better target for application generators

  It is much easier for application generators (programs that automatically create SQL code) to generate code for the SQL module processor that can be compiled directly, instead of also generating correct host language code for the SQL precompiler.
- Available from any host language

  Module language allows procedures that contain SQL statements to be called from any host language. The SQL precompiler only supports specific languages.
- More language features available

  SQL module language does not restrict the use of host language features not supported by the precompiler (such as pointer variables in C, block structure, macros, user-defined data types, and references to array elements).

[1] Note that the SQL precompiler creates a rewritten host language source file that replaces the embedded statements with external procedure calls and, therefore, this file can conform to an ANSI/ISO standard.

Programs that support pointer variables can take full advantage of dynamic SQL and use the SQL Descriptor Area (SQLDA) or the Extended SQL Descriptor Area (SQLDA2) with the SQL module language.

1.3 Finding Online Program Examples

SQL provides sample databases and programs that you can use to become more familiar with SQL programming.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the samples directory is defined by the logical name SQL$SAMPLE. ♦

Digital UNIX

On Digital UNIX, the samples directory is located in the following directory:

```
/usr/lib/dbs/sql/vnn/examples
```

The subdirectory `vnn` signifies the version of Oracle Rdb, for example, `v70`. ♦

The samples directory includes source programs that both define and query sample personnel databases. You can create an executable image of any source program for which your system has language support and then run the image to see how the program works. You can also print program sources and use them as references when you create your own applications.

Part II

Developing Application Programs That Use SQL

This part explains how to develop application programs using SQL. It presents information on the following topics:

- Understanding the application development cycle
- Using SQL module language
- Using precompiled SQL
- Linking, running, and debugging your program
- Specifying the parameters your program uses to exchange information with SQL
- Using SQL's date-time data types

2

SQL Program Development Cycle

Designing SQL application programs follows the same steps as those used in designing any host language program; however, their design does involve additional considerations. For example, the SQL statement syntax and the way you include SQL statements in your host language program (embedded or module) influence SQL application program design.

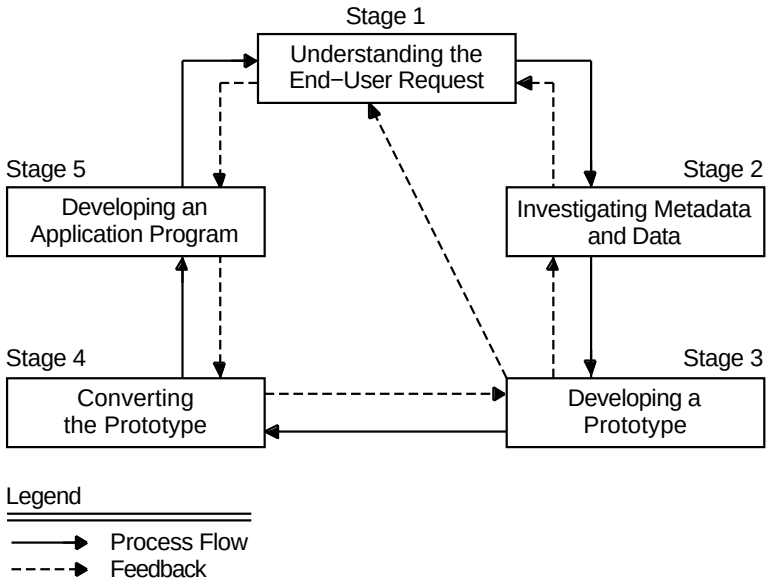
This chapter provides a summary of the stages in the application program development cycle. It describes the following:

- Understanding the application program development cycle
- Understanding end-user requirements
- Investigating metadata and data
- Developing a prototype for the application
- Converting the prototype to an application program
- Developing an application program

2.1 Overview of the Application Program Development Cycle

Figure 2–1 shows each stage of the application program development cycle and how they interact.

Figure 2–1 Stages of the Application Development Cycle



Like most processes, the application program development cycle has a well-defined flow of control through each stage, but timely feedback and iteration through the stages improve effectiveness and efficiency.

2.2 Understanding End-User Requirements

The first stage is to understand what the end users require from the database system, so you can translate this request into SQL. Your overall goal is to ensure that the final application program satisfies the users' requirements.

Too often, users are not fully satisfied with the completed application. "I know what I said, but that is not what I meant" is a familiar complaint. To avoid this problem, make sure you *and* the users understand the requirements and that your understandings are the same.

Sometimes, users do not completely understand the database system and how they can use it. You may need to teach them about the database system and what it can and cannot do, to help them make more precise requirements. As a result, the requirements may change. The original requirements also evolve over time as users think about and use the application. Occasionally, the conditions that originally inspired the user requirements change, and therefore the program requirements change. Oracle Rdb helps accommodate

these changing conditions with a variety of interfaces and a flexible distributed environment.

Periodically confirm your understanding of the end-user requirements. If you regularly communicate with your users throughout the application development cycle, especially to verify the prototype, the users are more likely to be satisfied with the final application program.

2.3 Investigating Metadata and Data

To create a query or to construct any SQL statement that correctly satisfies end-user requirements, you need to investigate both the metadata and the data available in the database system.

In general, ask yourself the following questions:

- What tables, views, columns, functions, and procedures currently exist in the database that can help satisfy the requirements?
- Do I need to modify or create metadata (tables, views, columns, indexes, or constraints)? Do I need to modify the data to satisfy the requirements?
- What indexes or constraints are defined on these columns? Do I need to add or subtract indexes or constraints?
- What is the quantity and quality of the data in the database system? Is it sufficient to satisfy the requirements? Is the data accurate, valid, and up-to-date? Is the data complete, or at least comprehensive enough to satisfy the requirements?
- Will this query be executed only this one time, or will users want to ask the same question many times? If the query will be asked frequently, should it be stored and reused, or included in a program?
- What privileges are required to access the data?

You can use either the interactive user interface or the programming interface at this stage to help you answer these questions. For example, you can use the interactive SQL SHOW statement to investigate the metadata and the SELECT statement to investigate system tables.

2.4 Developing a Prototype

In the next stage, you develop a prototype to help satisfy the requirements of the end user. Developing a prototype often requires you to return to the users several times before the prototype is correct.

You can develop your prototype using either interactive SQL or programmatic SQL. If you create and test interactive SQL statements before including them in a module or embedding them in a host language program, debugging your application program is easier and you shorten the program development cycle.

Interactively developing prototypes lets you:

- Find and correct syntax and semantic errors

  Interactive SQL can help you eliminate syntax errors in SQL statements. You can use the EDIT statement to to correct syntax errors.

- Evaluate the effectiveness of your queries

  You can refine queries so that they are more efficient. For example, you can try alternative approaches to a query (such as comparing a join to a subquery) to see which performs most efficiently. Or, you can add an index to improve performance.

- Understand the types of data validation your program must handle

  Using interactive SQL, you can test input and output values so that your application program handles them properly. You can also determine if you need additional validity checking for your input operations beyond the current table or column constraints.

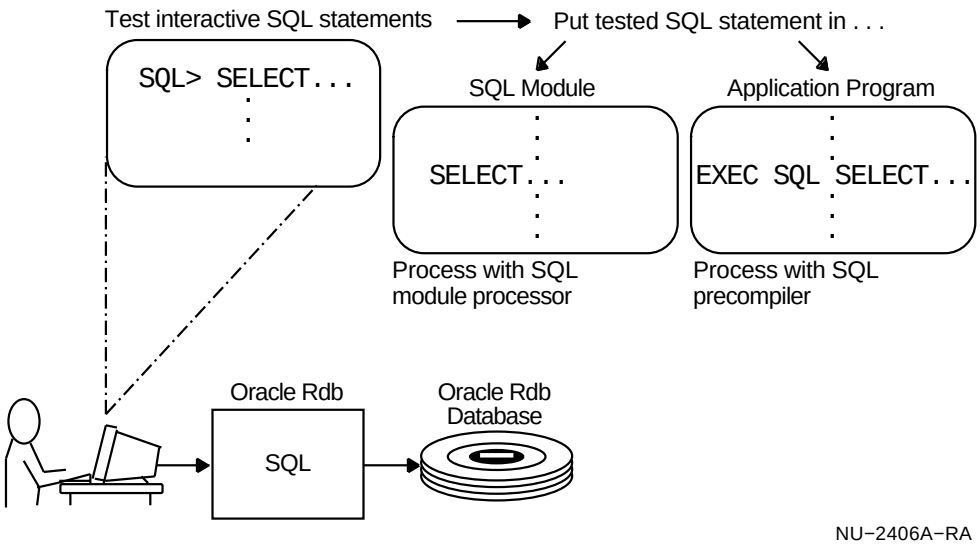
- Anticipate run-time errors

  Interactive SQL can give you a clear idea of the run-time errors your program may encounter. You can test the conditions—such as invalid input, a constraint violation, or a lock conflict caused by resource contention—that produce a specific run-time error.

Once you develop a correct prototype (one that is both syntactically correct *and* satisfies the original end-user requirements), you can save it in a script for easy re-execution or in a file for easy conversion to a program. Figure 2–2 illustrates this process.

If the requirements involve queries that will be executed only once, you may not need to develop an application program. You can save the output in a data file. Consider converting queries to views if you expect users to execute them frequently.

Figure 2–2 Using Interactive SQL as a Prototyping Tool



2.5 Converting the Prototype to an Application Program

Convert the SQL statement or statements developed interactively to the syntax used in your chosen programming interface. Some SQL statements have a slightly different syntax when used interactively than when used with a programming interface. Also, you need to use variables, instead of literals, to express values.

Important questions to ask yourself are:

- What values do I need to communicate between my application program and the SQL statements?

  You need to define main parameters for the module processor or host variables for the precompiler.

- What columns (whether for data retrieval or storage) may contain null values?

  You may need to declare indicator parameters for any main parameters associated with these columns.

- How will I handle both anticipated and unanticipated errors when I execute my application program?

At a minimum, you need to define execution status parameters. Usually, you want to handle errors to avoid problems in your application program or to provide additional control of your application program while it is executing.

See Chapter 8 for more information on using main and indicator parameters. Chapter 10 explains how to use execution status parameters.

- What are the SQL data types of each of the parameters? What comparable data types will I use in my application program?

  You need to correctly map data types between the host language program and the procedure.

- What data validation code may be needed in my application program?

  You may want to perform data validation in your application program in addition to the validation performed by current constraints. You may even decide to validate data entered by end users before inserting it into the database instead of using database constraints. Validating data in your application program minimizes locking because the validation is done outside the scope of a database transaction.

- What SQL keywords and user-defined names (such as columns and tables) may conflict with host language program keywords?

  In an established database system, it is not easy to change column names. However, you can create a view with names that your host language supports. You can then declare parameters that correspond to each column of the view.

  See the *Oracle Rdb7 SQL Reference Manual* for a list of keywords.

2.6 Developing an Application Program

The final stage is to develop the application program. This stage involves several steps and varies depending on which programming interface you use.

The basic steps are:

1. Edit the module and host language source code.
2. Compile the module and host language source code, and, if necessary, correct syntax and semantic errors.
3. Link object files and libraries into an executable image.
4. Run the executable image to verify results, and, if necessary, debug the image.

See Chapter 3 through Chapter 5 for more information on the steps you follow to develop an application program using the SQL module processor.

See Chapter 6 for more information on the steps you follow to develop an application program using the SQL precompiler.

3

Introduction to SQL Module Language

This chapter provides information about using the SQL module processor (SQL module language) to create applications that use SQL statements for database access. In the sections that follow, you will become familiar with how to:

- Develop an application using SQL module language
- Create SQL module source files
- Call SQL module procedures from a host language program
- Write portable code
- Find additional information about the SQL module processor

3.1 Developing SQL Module Language Application Programs: Basic Steps

Using SQL module language, you can create an **SQL module** file, which contains one or more procedures. Each procedure contains one or more SQL statements.

You call the procedures in the SQL module from programs written in traditional third-generation languages (3GLs) or from an Oracle Rally external link.

You can develop an SQL module language application program using the following steps, which are illustrated by the numbered callouts in Figure 3–1:

❶ Create the SQL module file.

The SQL module processor provides an SQL module language for creating an SQL module file. The module contains one or more procedures, each containing parameter declarations and one or more SQL statements. The host language program calls a particular SQL module procedure and supplies a sequence of actual parameters that correspond in number and in data type to the parameter declarations in the procedure of the module file.

❷ Use the SQL module processor to compile the SQL module source code file.

Compile the SQL module file by invoking the SQL module processor executable image and specifying the SQL module file name and a number of optional qualifiers on the same command line.

See the *Oracle Rdb7 SQL Reference Manual* for a complete description of each qualifier and examples.

❸ Edit the SQL module source code, if necessary.

After creating the SQL module file or after compiling it, check the module file carefully to correct any errors that you or the SQL module processor has found.

❹ Create the host language source code.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, you can use the SQL module processor with any host programming language that supports the OpenVMS Calling Standard. ♦

Digital UNIX

On Digital UNIX, you can use the SQL module processor with Digital C, Digital UNIX C, Digital COBOL, Digital FORTRAN, and Digital Pascal. ♦

❺ Compile the host language source code.

Use the host language compiler to create an object module. If the compilation is successful, the compilation process creates a host language object file.

❻ Edit the host language source code, if necessary.

If the host language program generated errors when you compiled it, edit the host language source code to correct the errors.

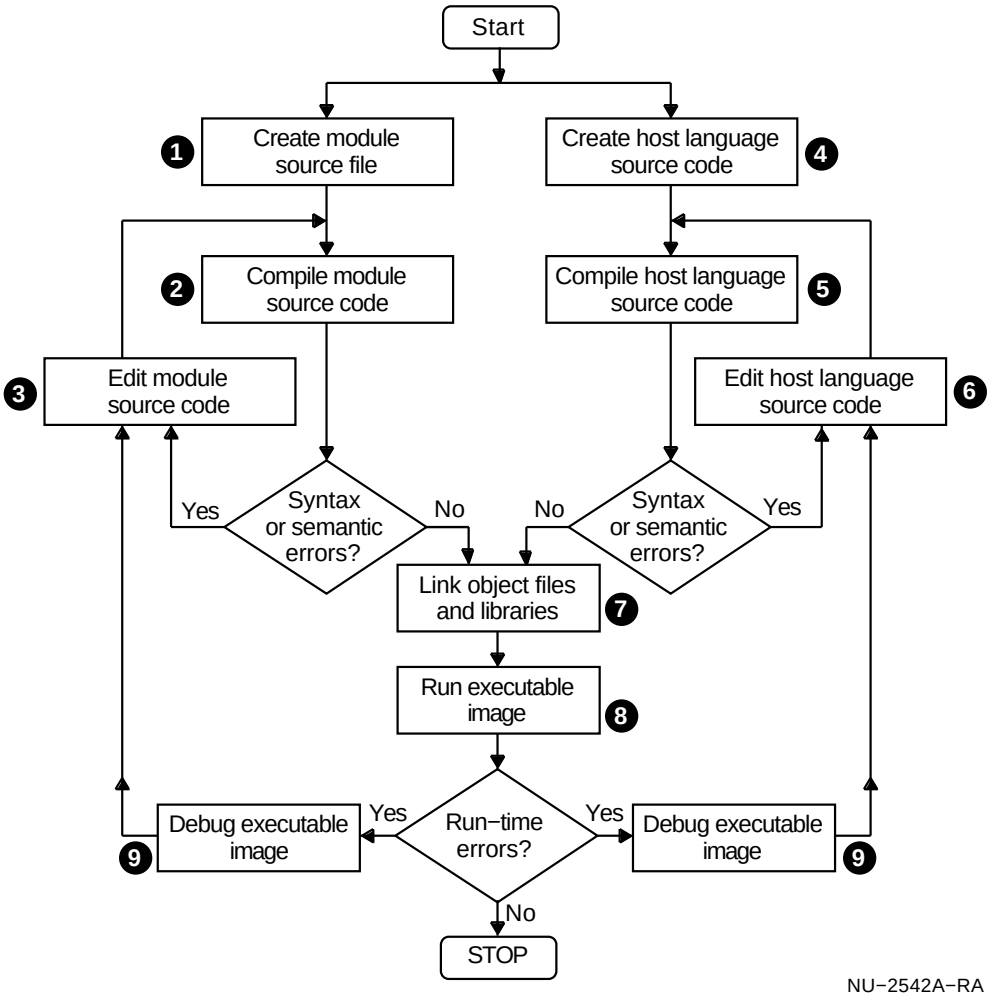
❼ Link the object modules from the host language and SQL module processor compilations to create an executable image.

❽ Run the executable image to verify its results.

❾ If you discover any run-time errors, debug the executable image.

After discovering the source of the errors, edit the host language program, the module file, or both files, and recompile one or more files as necessary until valid object files are created.

Figure 3–1 Developing Applications with the SQL Module Processor



3.2 Creating an SQL Module Source File

When you create an SQL module file, you create a source file written in **SQL module language**, a language that includes only SQL statements, procedures, and some special SQL keywords. An SQL module file contains the following components:

- Comments (optional)

- Module header information, which includes the following:
  - MODULE name clause
  - DIALECT clause (optional)
  - CHARACTER SET clauses (optional)
  - LANGUAGE clause
  - CATALOG name clause (optional)
  - SCHEMA name clause (optional)
  - AUTHORIZATION identifier clause (optional)
  - ALIAS clause (optional)
  - Other optional clauses, such as clauses to set character length, date format, how keywords are treated, how quotation marks are interpreted, privilege checking on execution, view update rules, and whether or not parameters are preceded by colons
- DECLARE statements section (optional)
- One or more procedures, each of which can contain a single SQL statement or one or more SQL statements in a compound statement, and the parameters to be used by those statements.

  A procedure that contains a compound statement is called a **multistatement procedure**.

Section 3.2.1 through Section 3.2.11 provide details about creating an SQL module. Chapter 4 provides information on writing SQL module procedures.

Example 3–1 shows an SQL module file with a set of numbered callouts that identify module components. Each callout is explained in a numbered list following the example.

Example 3–1 Parts of an SQL Module

```
-- The procedures in this SQL module are called by the C program      ❶
-- sql_intro_load_empl_h.c.
--
-------------------------------------------------------------------------
-- Header Information Section
-------------------------------------------------------------------------
                                                                       ❷
MODULE           INTRO_LOAD_EMPL_C      -- Module name
```

(continued on next page)

Example 3–1 (Cont.) Parts of an SQL Module

```
DIALECT          SQL92
LANGUAGE         C                      -- Language of calling program
AUTHORIZATION    SAMPLE_USER            -- Authorization ID
ALIAS            RDB$DBHANDLE           -- Default alias
PARAMETER COLONS                        -- Parameters are prefixed by colons
-------------------------------------------------------------------------------
-- DECLARE Statements Section
-------------------------------------------------------------------------------
-- Declare the alias using the file name.
DECLARE ALIAS FOR FILENAME intro_personnel                                  ❸
-------------------------------------------------------------------------------
-- Procedure Section
-- In every procedure, declare SQLCODE, a parameter that stores a value
-- that represents the execution status of SQL statements.
-------------------------------------------------------------------------------

-- This procedure uses the executable statement, SET TRANSACTION, to start
-- a transaction.  The EMPLOYEES, JOB_HISTORY, and DEPARTMENTS tables are
-- reserved because they are used in constraint checking.

PROCEDURE  SET_TRANS                                                        ❹
   (SQLCODE);

    SET TRANSACTION READ WRITE RESERVING
                EMPLOYEES FOR EXCLUSIVE WRITE,
                JOB_HISTORY FOR SHARED READ,
                DEPARTMENTS FOR SHARED READ;


-- This procedure inserts the employee data into the EMPLOYEES table.

PROCEDURE INSERT_DATA                                                       ❹
   (SQLCODE
-- Declare the parameters by which the values are passed between the SQL
-- module and the host language.
   :EMPLOYEE_RECORD RECORD
      P_EMPLOYEE_ID    CHAR(5),
      P_LAST_NAME      CHAR(14),
      P_FIRST_NAME     CHAR(10),
      P_MIDDLE_INITIAL CHAR(1),
      P_ADDRESS_DATA   CHAR(25),
      P_CITY           CHAR(20),
      P_STATE          CHAR(2),
      P_POSTAL_CODE    CHAR(5),
      P_BIRTHDAY       CHAR(10),
      P_SEX            CHAR(1)
    END RECORD);

-- The list of names that follows the INSERT clause identifies the columns
-- in the table that are to be used for the insert operation.
-- The list of names in the VALUES clause corresponds to the variables
-- declared in the procedure.
```

(continued on next page)

Example 3–1 (Cont.) Parts of an SQL Module

```
INSERT INTO EMPLOYEES
      (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
      ADDRESS_DATA, CITY, STATE, POSTAL_CODE, BIRTHDAY, SEX )
VALUES
     (:EMPLOYEE_RECORD.P_EMPLOYEE_ID, :EMPLOYEE_RECORD.P_LAST_NAME,
      :EMPLOYEE_RECORD.P_FIRST_NAME, :EMPLOYEE_RECORD.P_MIDDLE_INITIAL,
      :EMPLOYEE_RECORD.P_ADDRESS_DATA, :EMPLOYEE_RECORD.P_CITY,
      :EMPLOYEE_RECORD.P_STATE, :EMPLOYEE_RECORD.P_POSTAL_CODE,
      CAST(:EMPLOYEE_RECORD.P_BIRTHDAY AS DATE ANSI),
      :EMPLOYEE_RECORD.P_SEX  );
.
.
.
```

The callouts in Example 3–1 refer to the following:

❶ Comment

❷ Module header

❸ Declare statement

❹ Procedure

Section 3.3 explains how to call an SQL module procedure from a host language program, and shows a program that calls the SQL module sql_intro_load_empl_c.sqlmod, shown in Example 3–1.

3.2.1 Including Blank Lines and Comments in an SQL Module

Double hyphens (--) specify that all remaining text on a line is a comment. The SQL module processor ignores any text to the right of double hyphens when it processes source files. As Example 3–1 shows, you can specify comments and leave lines blank when you need to make your SQL module source file easier to read and understand.

3.2.2 Naming a Module

To name a module, you must specify the MODULE clause and optionally include a module name in the first line (excluding comment lines and blank lines) of a module. The following module segment assigns MSDB_MOD as the module name:

```
MODULE          MSDB_MOD          -- Module name
```

When you omit the module name, SQL uses the name SQL_MODULE by default. Although the module name is not required, the MODULE keyword is.

If you plan to combine multiple program modules in one executable or shareable image, the names of all modules, including stored modules, must be unique. For example, an SQL module cannot have the same name as host language modules or other SQL modules with which it may later be linked. Module names that are not unique cause an error regarding ambiguous global symbols when you link object files into an image.

There is no relationship between the actual file name of the module and the name of the module in the module header.

3.2.3 Specifying the Dialect

The DIALECT clause controls many settings for the session, including the following:

- Whether the length of character string literals, columns, and domains are interpreted as characters or octets
- Whether double quotes (") are interpreted as string literals or delimited identifiers
- Whether or not identifiers can be keywords
- Which views are read-only
- Whether columns with the DATE or CURRENT_TIMESTAMP data type are interpreted as VMS or ANSI/ISO format
- Whether parameter names begin with a colon
- Whether the session character sets change depending on the dialect specified

The following example shows how to specify the dialect setting as SQL92:

```
DIALECT   SQL92                -- Set the dialect.
```

The DIALECT clause lets you specify the settings for the module with one clause, instead of specifying each setting individually. The dialect settings include SQL92, SQL89, MIA, and SQLV40. The SQLV40 setting is the default. Set the dialect to SQL92 or MIA unless you need to maintain compatibility with an earlier dialect. For information on the particular settings for each dialect, see the description of the SET DIALECT statement in the *Oracle Rdb7 SQL Reference Manual*.

Because the module processor processes the module clauses sequentially, the DIALECT clause can override the settings of clauses specified before it, or be overridden by clauses specified after it in an SQL module file.

3.2.4 Specifying Character Sets for a Session

You can specify the following character sets for an SQL session or module:

- Default character set

  The default character set determines the character set for database columns with a character data type that is not qualified by a character set or national character set.

- Identifier character set

  The identifier character set determines the character set for user-supplied database object names such as table names, domain names, and column names. The character set must contain ASCII.

- National character set

  The national character set determines the character set for all columns with the data type NCHAR or NCHAR VARYING, and for character string literals qualified by the national character set.

- Literal character set

  The literal character set determines the character set for literals that are not qualified by a character set or national character set.

The *Oracle Rdb7 SQL Reference Manual* provides a set of tables listing the character sets that you can use for the default character set, the identifier character set, the national character set, and the literal character set. The following example sets the default character set to KANJI:

```
DEFAULT CHARACTER SET     KANJI      -- Set the default character set
```

SQL uses DEC_MCS for the default, identifier, national, and literal character sets if you specify these character sets using the SET command and if you do not set the dialect of the session to MIA. However, if you change the dialect after setting it to MIA, the character sets do not change. Only the rules associated with the new dialect are changed.

3.2.5 Identifying the Host Language That Calls Module Procedures

After naming the module, you must specify the LANGUAGE clause and one of the following keywords to identify the language used by calling host language programs: ADA, BASIC, C, COBOL, FORTRAN, PASCAL, PLI (for PL/I), or GENERAL. For example:

```
LANGUAGE        C                 -- Language of calling program
```

The language identifier determines both the kinds of data types that the SQL module processor considers valid when you later declare procedure parameters, and the mechanism that SQL expects the calling module to use for passing parameter values at run time.

Specify the GENERAL keyword for languages that do not have a corresponding keyword in the LANGUAGE clause.

3.2.6 Specifying the Catalog

The optional CATALOG clause lets you specify a catalog name in the module header. SQL uses this name by default when you name schema objects within a multischema database and do not explicitly include the catalog name. SQL uses RDB$CATALOG as the default if you omit the CATALOG clause in your module file.

Specifying a default catalog name, and thus overriding the RDB$CATALOG default, enables you to name schemas and schema objects within that catalog without the need to include the catalog name. Omitting the catalog name saves you the task of typing the name each time that you name an object in a module.

Suppose that you do not want to specify the ADMINISTRATION catalog name in every schema object that you plan to identify in an SQL module. Change the default catalog for the module by including the following CATALOG clause:

```
CATALOG ADMINISTRATION
```

Then, you can name a schema object such as the EMPLOYEES table within the PERSONNEL schema by entering the following unique multischema name:

```
SELECT * FROM PERSONNEL.EMPLOYEES;
```

As Section 3.2.7 shows, you can also omit the schema name if you change the default schema with the SCHEMA clause.

If you have multischema databases that contain schemas with objects that have the same names, you can duplicate SQL modules for use with a variety of schemas by not explicitly specifying the catalog and schema name in the SQL statements, but instead changing the default catalog and schema name in the module header.

The CATALOG clause affects only those module procedures that use databases enabled with the optional multischema attribute.

3.2.7 Specifying the Schema

The optional SCHEMA clause lets you specify a schema name in the module header. SQL uses this name by default if you do not explicitly include the schema name when naming schema objects within a multischema database. If you do not explicitly name a schema in the SCHEMA clause, SQL uses the implicit authorization identifier, which is the user name of the person compiling the module, as the default schema. If the authorization identifier is explicitly named in the AUTHORIZATION clause, SQL uses that name as the default schema.

Specifying a schema name in the SCHEMA clause overrides the schema default and enables you to name schema objects within that schema and catalog without having to include the schema name. Omitting the schema name saves you the task of typing the complete name of schema objects in a module.

Suppose that you do not want to specify either the ADMINISTRATION catalog name or the PERSONNEL schema for the schema objects you that plan to identify in an SQL module. Change the default catalog and schema for the module by including the following module clauses:

```
CATALOG ADMINISTRATION
SCHEMA PERSONNEL
```

Then, you can name a schema object such as the EMPLOYEES table within the PERSONNEL schema in the ADMINISTRATION catalog by entering just the table name:

```
SELECT * FROM EMPLOYEES;
```

The SCHEMA clause affects only those module procedures that use multischema databases.

3.2.8 Specifying an Authorization Identifier

The optional AUTHORIZATION clause specifies the authorization identifier for the module. If you omit the authorization identifier, SQL selects the user name of the user compiling the module as the default authorization. Thus, if you use the RIGHTS clause in the module header, SQL compares the user name of the person who executes a module with the authorization identifier with which the module was compiled and prevents any user other than the one who compiled the module from invoking it. When you use the RIGHTS clause, SQL bases privilege checking on the default authorization identifier (in compliance with the ANSI/ISO standard). By default, SQL does not perform privilege checking using the authorization identifier; it uses the privileges of the invoker.

The following example sets the authorization identifier:

```
AUTHORIZATION    SAMPLE_USER            -- Authorization ID
```

When you do not explicitly name a schema in the SCHEMA clause or an alias in the ALIAS clause, the authorization identifier for the module becomes the default schema and default alias. Thus, in this case, the default schema, default authorization identifier, and default alias have the same name.

3.2.9 Specifying the Alias

The optional ALIAS clause specifies the default database alias for the module. If you omit the alias in the module header, SQL uses the authorization identifier explicitly named in the AUTHORIZATION clause as the default alias. SQL uses RDB$DBHANDLE as the default alias when you do not explicitly name an authorization identifier in the module header. Use the ALIAS clause when you must reference more than one database within a module.

The following example specifies the default alias;

```
ALIAS            RDB$DBHANDLE         -- Default alias
```

Oracle Rdb suggests that you use the same alias for the default database in all modules linked to create an executable image if the application needs to refer to only one database across multiple modules.

If the image will include modules processed with the SQL precompiler, you should specify RDB$DBHANDLE in the AUTHORIZATION clause of all SQL modules in the image. The alias RDB$DBHANDLE is always designated as the default database in precompiled SQL programs.

3.2.10 Specifying That Parameters Must Include Colons

You can specify that parameters in module procedures be preceded by colons (:) by using the PARAMETER COLONS clause in the SQL module header, as shown in the following example:

```
PARAMETER COLONS                    -- Parameters are prefixed by colons
```

Currently, the default behavior of the SQL module processor does not allow colons. However, because the use of colons is required by the ANSI/ISO SQL standard, Oracle Rdb recommends that you specify the PARAMETER COLONS clause and use colons with parameters. If you specify the dialect as SQL92, SQL enforces the use of colons even if you do not specify the PARAMETER COLONS clause.

3.2.11 Specifying DECLARE Statements in Modules

You can specify DECLARE statements, such as the following, in the SQL module header:

- DECLARE ALIAS statements

  This statement specifies that name and source of the database to be used by the module. If you are working with only one database and do not need repository access while attached to that database, you may prefer to omit a DECLARE ALIAS statement from the module and assign a file specification to the logical name SQL$DATABASE or configuration parameter SQL_DATABASE. Otherwise, include a DECLARE ALIAS statement in the module for each database that you intend to access.

  See Section 15.1 for more information about database attachment options.

- DECLARE CURSOR statements or dynamic DECLARE CURSOR statements

  This statement declares a cursor. You must include a DECLARE CURSOR statement for each cursor to which statements in a module procedure refer. See Chapter 18 for information about statements that declare and use cursors.

- DECLARE STATEMENT statements

  This statement documents a statement name that is used later in dynamic SQL. See Chapter 11 for more information on dynamic SQL.

- DECLARE TABLE statements

  These statements specify table and view definitions in a source file. At compile time, the SQL module processor does not have to attach to a database or access the repository if an SQL source file contains definitions for the tables and views that are accessed by the program at run time. The DECLARE TABLE statement is useful when you want to compile an SQL module that refers to a table or view that does not yet exist, or when you want to eliminate at compile time the resource overhead that is associated with database attachment.

- DECLARE TRANSACTION statement

  This statement establishes default characteristics for all transactions started implicitly by the program. You do not need to specify a DECLARE TRANSACTION statement if you want SQL defaults for transactions or if you plan to explicitly start transactions by calling a procedure for a SET TRANSACTION statement. See Chapter 16 for complete information about transactions.

Do not use a semicolon (;) to end DECLARE statements in SQL modules. The semicolon is valid only in module procedures.

3.3 Calling SQL Module Procedures from a Host Language Program

The format for calling SQL module procedures from a host language program is specific to the host language. Consult your host language documentation for the format that you should use. The host language documentation also details how to override a passing mechanism default.

Example 3–2 shows excerpts from the C host language program, sql_intro_load_empl_h.c, which calls SQL procedures in sql_intro_load_empl_c.sqlmod, as shown in Example 3–1.

Example 3–2 Host Language That Calls an SQL Module

```
   .
   .
   .
/*
 This structure represents the record to be inserted into the database.
 Character strings are one character longer to hold the null value that
 terminates the string.  */

struct Employee_struct {
        char employee_id[6];
        char last_name[15];
        char first_name[11];
        char middle_initial[2];
        char address_data[26];
        char city[21];
        char state[3];
        char postal_code[6];
        char ascii_birthdate[11];
        char sex[2];
        };

/* This structure represents the format of each record in the stream file. */
struct Employee_struct_buf {
        char employee_id[5];
        char last_name[14];
        char first_name[10];
```

(continued on next page)

Example 3–2 (Cont.) Host Language That Calls an SQL Module

```
        char middle_initial[1];
        char address_data[25];
        char city[20];
        char state[2];
        char postal_code[5];
        char ascii_birthdate[10];
        char sex[1];
        char linefeed[1];
        };

/* Return status and SQLCODE variables for error handling. */

int  return_status;
long sqlcode;

/* Function prototypes */

void SET_TRANS    (long *sqlcode);
void INSERT_DATA  (long *sqlcode, struct Employee_struct *employee_rec_prt);
void COMMIT_TRANS (long *sqlcode);
   .
   .
   .
/* Start the transaction by calling the SQL module language procedure
   SET_TRANS in sql_intro_empl_h.sqlmod.  If the call to SET_TRANS returns
   a value other than 0, use sql_signal to display the error.  */

        SET_TRANS(&sqlcode);
        if (sqlcode != 0)
        {
        printf("\nStart transaction failed.\n");
        sql_signal();
        }

   .
   .
   .
/* Invoke the INSERT_DATA procedure to insert a row in the EMPLOYEES table.
   The date is converted from text to binary by the CAST function in the SQL
   procedure.  If the call to INSERT_DATA returns a value other than 0,
   use sql_signal to display the error.  */

        INSERT_DATA (&sqlcode, &employee_rec);
        if (sqlcode != 0) sql_signal();

    }
```

(continued on next page)

Example 3–2 (Cont.) Host Language That Calls an SQL Module

```
/* Commit the transaction by calling the SQL module language procedure.
   COMMIT_TRANS. */

        COMMIT_TRANS(&sqlcode);
   .
   .
   .
```

3.4 Writing Portable Applications Using the SQL Module Processor

You might need to write applications that require little or no editing from one SQL implementation to another. SQL module language is included in the ANSI/ISO standard for SQL and thus offers a way to write standard-compliant programs. To identify the Oracle Rdb extensions to the ANSI/ISO standard in an SQL module, use the FLAG_NONSTANDARD or –std command line qualifier when compiling a module file.

Module language support is not implemented in many vendor implementations of SQL. You should determine what level of module language support a particular implementation offers before writing SQL module language procedures that you may want to port to that implementation.

3.5 Finding More Information About the SQL Module Processor

The following sources provide more information about using the SQL module processor:

- Chapter 4 discusses how to write SQL module procedures, including procedures that include single SQL statements and compound statements in multistatement procedures.
- Chapter 5 discusses how to compile and link SQL modules and host language programs, and how to use context files to improve portability.
- Chapter 7 discusses how to link SQL and host language object modules into images.
- The samples directory provides online examples of SQL modules and their associated calling host language programs. For more information about the sample programs, see the online file, about_sql_examples.txt, in the samples directory.

- The *Oracle Rdb7 SQL Reference Manual* provides reference information on SQL module language.

4

Writing Module SQL Procedures

This chapter discusses how to write SQL module procedures. Refer to Chapter 3 for introductory information about the SQL module language and for a discussion of how to create SQL modules and the module header information. The sections that follow explain:

- The components of SQL module procedures and the different types of procedures
- How to specify the common elements of SQL module procedures, such as procedure names and parameters and their data types,
- How to specify a single SQL statement in a procedure
- How to specify compound statements in a procedure
- The restrictions of the SQL module language

4.1 Introducing SQL Module Procedures

A procedure in an SQL module is a mechanism for executing a set of SQL statements through a call from a host language module.

An SQL module procedure consists of a set of common procedure elements that include the following:

- The keyword PROCEDURE to introduce the procedure
- The procedure name
- A status parameter for error handling
- One or more parameter declarations
- A single SQL statement or one or more SQL statements in a compound statement.

Section 4.2 describes the common procedure elements.

You can define two types of procedures:

- Simple statement procedure

  A **simple statement procedure** consists of the common procedure elements and a single executable SQL statement only. The following simple statement procedure starts a read/write transaction when the SET_TRANS procedure is executed:

```
PROCEDURE SET_TRANS
    (SQLSTATE);

    SET TRANSACTION READ WRITE;
```

  Section 4.3 describes how to write simple statement procedures.

- Multistatement procedure

  A **multistatement procedure** consists of the common procedure elements and a compound statement that includes one or more SQL statements. You can use flow-control statements, such as the SQL IF and FOR statements, in compound statements.

  The following multistatement procedure contains two INSERT statements:

```
PROCEDURE INSERT_DATA
-- Declare the parameters by which the values are passed between the SQL
-- module and the host language.
   (SQLSTATE,
   :P_EMPLOYEE_ID    CHAR(5),
   :P_LAST_NAME      CHAR(14),
   :P_FIRST_NAME     CHAR(10),
   :P_JOB_CODE       CHAR(4),
   :P_DEPARTMENT_CODE CHAR(4));

BEGIN

   INSERT INTO EMPLOYEES
       (EMPLOYEE_ID, LAST_NAME, FIRST_NAME)
       VALUES
       (:P_EMPLOYEE_ID, :P_LAST_NAME, :P_FIRST_NAME);

   INSERT INTO JOB_HISTORY
       (EMPLOYEE_ID, JOB_CODE, JOB_START, DEPARTMENT_CODE)
       VALUES
       (:P_EMPLOYEE_ID, :P_JOB_CODE, CURRENT_TIMESTAMP, :P_DEPARTMENT_CODE);
END;
```

  Section 4.4 describes how to write multistatement procedures and Chapter 12 provides tutorial information on writing compound statements.

4.2 Specifying the Common Elements of SQL Module Procedures

Whether you use a simple statement procedure or a multistatement procedure, a procedure definition:

- Starts with the word PROCEDURE
- Specifies a procedure name
- Declares parameters by which values are exchanged between the SQL module and the calling host language module
- Specifies that the parameter is passed by descriptor (optional)
- Specifies that SQL return a run-time error if the calling module is not passing a parameter by descriptor (optional for parameter declarations that include a BY DESCRIPTOR clause)

4.2.1 Naming a Procedure

An SQL module procedure must begin with the PROCEDURE keyword and be followed by a name for the procedure. Procedure names within an executable image must be unique. If they are not unique within an SQL module, you encounter a compile-time error from the SQL module processor.

Furthermore, if you combine multiple SQL modules in one executable or shareable image, the names of procedures must be unique within the image. In this case, procedure names that are not unique cause an error regarding ambiguous global symbols when you link object files into an image.

Avoid choosing names that are used by other procedures linked with the applications, including system procedures and procedures in any library used by the application.

Digital UNIX

Because the Digital UNIX environment is case sensitive, and because on Digital UNIX the SQL module processor converts procedure names to uppercase by default, you need to take care in using module procedure names.

For example, if you write calls to SQL module language procedures from a C program and invoke those procedures using lowercase characters, you get an error when you link the object files. The C language retains the lowercase characters while SQL converts the names to uppercase.

To eliminate this problem, use one of the following methods:

- From the host language, invoke the SQL procedures using uppercase names.

- Use the –lc_proc command line qualifier to force the names of the module language procedures to lowercase. ♦

4.2.2 Declaring Procedure Parameters

You declare a parameter for every value that the SQL module and host language module exchange when the SQL statement in the procedure executes. Parameter declarations are required for the following kinds of values:

- A value to indicate execution status of the procedure

  You must specify one of the following status parameters in every procedure:

  - SQLCODE
  - SQLCA
  - SQLSTATE

  Oracle Rdb recommends that you use the SQLSTATE status parameter for any new application. SQLSTATE supports more standard status codes than SQLCODE and is the preferred mechanism in the ANSI/ISO SQL standard. The SQLCA status parameter is an extension to the ANSI/ISO SQL standard.

  When you declare a status parameter, you do not specify a data type because the data type is implicit.

  See Chapter 10 for a discussion of these parameters.

- Any values on which table searches are based

  These parameters are required to evaluate most WHERE or HAVING clauses that you specify in the executable statement of the procedure. They are also required in the declaration of a cursor.

- Any column value being retrieved or stored by the procedure

  The data types of parameters for column values are discussed in Section 4.2.5.

- Indicator values for handling null values, if any, in columns

  See Section 8.10 for information about handling column values and nulls.

- The keyword SQLDA

  The **SQLDA** (SQL Descriptor Area) is a data structure that provides information about dynamic SQL statements. You must declare the SQLDA in host language modules for use with dynamic SQL statements whose select list parameters and parameter markers are not known until run time. See Chapter 11 for more information about the SQLDA.

- The keyword SQLDA2

  The **SQLDA2** (SQL Descriptor Area 2) is an extended version of the SQLDA structure. The SQLDA2 data structure provides additional field and field size information about dynamic SQL statements. You should declare the SQLDA2 instead of the SQLDA when any of the following apply to the parameter markers or select list items:

  - The length of the column name is greater than 30 octets (8 bits).
  - The data type of the column is a date-time data type.
  - The data type is CHAR, CHAR VARYING, CHARACTER, CHARACTER VARYING, VARCHAR, or LONG VARCHAR, and either or both of the following is true: the character set is not the default 8-bit character set or the maximum length in octets exceeds 65,535.

  You declare the SQLDA2 in host language modules for use with dynamic SQL statements whose select list parameters and parameter markers are not known until run time.

  See Chapter 11 for more information about the SQLDA2.

When you declare the parameters supplied by SQL (SQLSTATE, SQLCODE, SQLCA, SQLDA, and SQLDA2), do not declare a data type; however, when you declare a user-defined parameter, you must declare a data type for the parameter.

To comply with the ANSI/ISO standard for SQL and to improve readability and avoid ambiguity, Oracle Rdb recommends that you prefix each parameter with a colon (:), separate parameters with a comma (,), enclose all parameters with one set of parentheses, and terminate the list of parameter declarations with a single semicolon (;).

4.2.3 Parameters Required for Different Kinds of Procedures

You need to declare certain kinds of parameters in all module procedures. The parameters that you declare depend upon the SQL statement executed by the procedure, as follows:

- If the SQL module contains a DECLARE ALIAS statement that includes a program parameter for the database specification, you must declare that parameter in every procedure in the SQL module. In addition, you must specify that parameter in every host language call to the procedures in the SQL module. The DECLARE ALIAS statement contains a program parameter whenever the host language module accepts a file name or repository path name value at run time to identify the location of the database.

Example 4–1 shows how the sql_insert_degrees_module.sqlmod module file is modified to enable the host language module, or a user, to enter a file specification for the database at run time.

Example 4–1 Writing SQL Modules to Accept the Database Name at Run Time

```
-- This SQL module accepts the database name at run time.
-----------------------------------------------------------------
-- Header Information Section
-----------------------------------------------------------------
MODULE         sql_insert_degrees_mod_param -- Module name
LANGUAGE       COBOL                        -- Language of calling program
AUTHORIZATION  RDB$DBHANDLE                 -- Authorization identifier
PARAMETER COLONS                            -- Use colons
-----------------------------------------------------------------
-- DECLARE statement section
-----------------------------------------------------------------
DECLARE ALIAS FOR COMPILETIME FILENAME personnel
                  RUNTIME FILENAME :DB_NAME
-----------------------------------------------------------------
-- PROCEDURE SECTION
-----------------------------------------------------------------
PROCEDURE INSERT_DEGREES
    (:DB_NAME         CHAR(80),
     SQLCODE,
    :P_EMPLOYEE_ID   CHAR(5),
    :P_COLLEGE_CODE  CHAR(4),
    :P_YEAR_GIVEN    SMALLINT,
    :P_DEGREE        CHAR(3),
    :P_DEGREE_FIELD  CHAR(15));

  INSERT INTO DEGREES
        (EMPLOYEE_ID, COLLEGE_CODE, YEAR_GIVEN,
         DEGREE, DEGREE_FIELD)
      VALUES
        (:P_EMPLOYEE_ID, :P_COLLEGE_CODE, :P_YEAR_GIVEN,
         :P_DEGREE, :P_DEGREE_FIELD);

PROCEDURE START_TRANS
    (:DB_NAME CHAR(80),
    SQLCODE);

    SET TRANSACTION READ WRITE;
```

(continued on next page)

Example 4–1 (Cont.) Writing SQL Modules to Accept the Database Name at Run Time

```
PROCEDURE COMMIT
    (:DB_NAME CHAR(80),
     SQLCODE);

    COMMIT;
```

You must modify the host language program to pass the database name to all the procedures in the SQL module. Example 4–2 shows an excerpt from the sql_insert_degrees_program.cob program, modified to pass the database name to all the procedures in the SQL module.

Example 4–2 Passing the Database Name to an SQL Module

```
   .
   .
   .
* Declare the database name.
01    P_DB_NAME       PIC X(80).
   .
   .
   .
GET_DATABASE.
    DISPLAY " Enter the database name".
    ACCEPT P_DB_NAME.
   .
   .
   .
    CALL "START_TRANS" USING P_DB_NAME, SQLCODE.
    CALL "INSERT_DEGREES" USING P_DB_NAME, SQLCODE, P_EMPLOYEE_ID,
          P_COLLEGE_CODE, P_YEAR_GIVEN, P_DEGREE, P_DEGREE_FIELD.
    CALL "COMMIT" USING P_DB_NAME, SQLCODE.
```

- For data definition, COMMIT, and ROLLBACK statements, you need to declare only those parameters required for all procedures. If the DECLARE ALIAS statement contains no parameters, you need to specify only SQLCODE, SQLCA, or SQLSTATE, as shown in the following example:

```
PROCEDURE COMMIT_TRANSACTION
          (SQLCODE);

          COMMIT;
```

- If a DECLARE CURSOR statement contains parameters, pass the parameters to it by declaring them in the procedure that contains the OPEN statement. In addition, you must specify the parameter in the host language call to the procedure that contains the OPEN statement. Because the DECLARE CURSOR statement appears in the declaration section of an SQL module, not a procedure, you cannot pass the parameters directly to the DECLARE CURSOR statement.

  See Example 4–3 for an example of using parameters with cursors.

- For an OPEN statement, declare all parameters required for all procedures in addition to the parameters included in the DECLARE CURSOR statement that correspond to the cursor that you are opening. Example 4–3 shows how to declare and use parameters in a cursor.

Example 4–3 Using Parameters with Cursors

```
DECLARE ID_AND_NAME CURSOR FOR
        SELECT EMPLOYEE_ID, FIRST_NAME, MIDDLE_INITIAL, LAST_NAME
          FROM EMPLOYEES
          WHERE EMPLOYEE_ID = :INPUT_ID

PROCEDURE OPEN_ID_AND_NAME
          (SQLCODE,
          :INPUT_ID    CHAR(5));

        OPEN ID_AND_NAME;
```

  Example 4–3 shows a parameter only in a WHERE clause. You can include parameters in other clauses of the DECLARE CURSOR statement; however, if you do, you must also declare those parameters in the procedure that opens the cursor.

  You can specify only one OPEN statement procedure for each DECLARE CURSOR statement in an SQL module.

- For a FETCH statement, declare all parameters specified for the INTO clause in addition to the parameters required for all procedures. For example:

```
PROCEDURE FETCH_ID_AND_NAME
          (SQLCODE,
          :ID_VAR            CHAR(5),
          :FIRST_NAME_VAR    CHAR(10),
          :MIDDLE_INIT_VAR   CHAR(1),
          :MIDDLE_INIT_IND   SMALLINT),
          :LAST_NAME_VAR     CHAR(14);
```

```
    FETCH EMP_ROW_CURSOR INTO
          :ID_VAR,
          :FIRST_NAME_VAR,
          :MIDDLE_INIT_VAR :MIDDLE_INIT_IND,
          :LAST_NAME_VAR;
```

- For the data manipulation statements INSERT, UPDATE, or DELETE, you must declare parameters needed for all procedures. In addition, you declare parameters that store column values or indicator values, or that are used in value comparisons. For example:

```
PROCEDURE UPDATE_JH
          (SQLSTATE,
           :JOB_END_DATE_BIN   DATE,
           :INPUT_EMP_ID       CHAR(5));

    UPDATE JOB_HISTORY
          SET JOB_END = :JOB_END_DATE_BIN
          WHERE EMPLOYEE_ID = :INPUT_EMP_ID;

PROCEDURE DELETE_JH
          (:INPUT_ID       CHAR(5),
           :INPUT_DB_FILE  CHAR(20),
           SQLSTATE);

   DELETE FROM JOB_HISTORY
          WHERE EMPLOYEE_ID = :INPUT_ID;
```

The sql_all_datatypes_ada.sqlmod module in the samples directory shows how to declare parameters for FETCH and UPDATE statements.

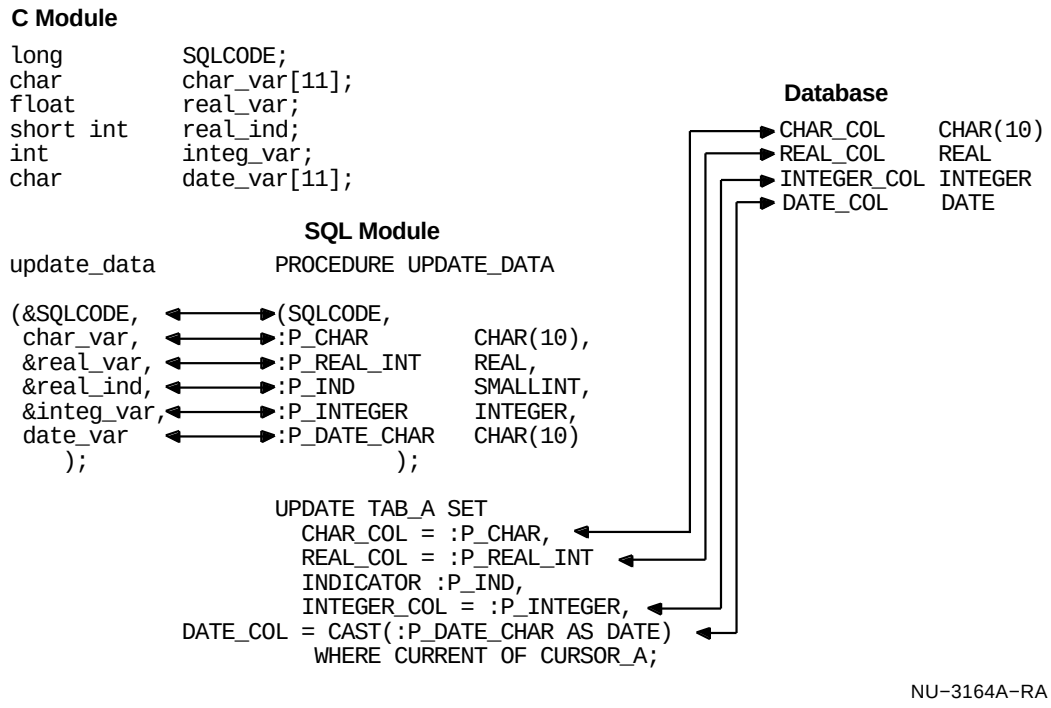
4.2.4 Associating Procedure Parameters and Actual Parameters

Each **procedure parameter**, a formal parameter in an SQL module procedure, corresponds to an **actual parameter**, which you declare in a host language module and then specify in the host language call to the procedure in the SQL module. For some host languages, the parameter in an SQL module also corresponds to a formal parameter identified in a declaration of an external procedure, routine, or function.

The correspondence between parameters in the SQL module and actual parameters in the host language module is established by order of specification. The first parameter declared in the SQL module procedure corresponds to the first parameter specified in the host language call, the second parameter declared in the SQL module procedure corresponds to the second parameter specified in the host language call, and so forth.

Figure 4–1 illustrates the correspondence between SQL module procedure parameters, actual parameters, host language declarations, and the database. In the figure, C is the language used in the calling module; however, the correspondence principles are the same for all languages.

Figure 4–1 Correspondence Between Actual Parameters, Procedure Parameters, and Columns



NU–3164A–RA

Note that, in Figure 4–1, char_var and date_var are declared to be one character longer than their corresponding procedure parameters. This difference in size is only when you use the C language (see Section 4.5 for more information.)

Consistent order is significant only in the correspondence between procedure parameters in the SQL module and actual parameters in the host language module. (In Figure 4–1, the parallel specification of column definitions and column references exists only to keep correspondence lines neat enough to follow.)

Furthermore, it is not important that the names of corresponding actual and procedure parameters match or that you specify one kind of parameter (for example, column) in a certain order with respect to another kind of parameter (for example, indicator). However, if you use similar names for corresponding procedure and actual parameters, and if you use a consistent ordering strategy from one procedure to the next, you are less likely to make parameter correspondence mistakes.

When you specify actual parameters in a different order than you specify procedure parameters (or specify a different number of actual parameters as compared to the procedure), you cause run-time errors that may be difficult to debug. The BY DESCRIPTOR CHECK clause can help detect this kind of error. See Section 4.2.8 for more information on the BY DESCRIPTOR CHECK clause.

4.2.5 Specifying Parameter Data Types

You can choose from a number of data types for procedure parameters in an SQL module. For a detailed description of these data types, read the section in the *Oracle Rdb7 SQL Reference Manual* on data types in the chapter about language and syntax elements. You can also specify the name of a domain definition in place of a data type keyword.

The following excerpt shows how to declare three parameters:

```
(SQLSTATE,                    -- Status parameter.  No data type specified.
 :LAST_NAME_VAR   CHAR(14),   -- Character data type.
 :EMP_ID          ID_DOM)     -- Data type based on the domain ID_DOM.
```

When deciding on parameter data types, you must consider data type correspondence for all of the following:

- Column definition
- Parameter declaration in the SQL module
- Parameter declaration in the host language module

In most cases, it is possible to maintain consistency of data storage formats between the database and programs. However, if either of the following conditions is true, you must declare parameters in the SQL module and host language module to be of a data type that is different from the one specified in the column definition:

- The language in which you write the calling module does not support the data storage format specified in the table column definition, and your program must work with values stored in that column.

- Your program writes to a database from a file (or writes to a file from the database) and the record field associated with the column value has a different storage format than the column.

Usually, you can solve either problem by instructing SQL to convert between data storage formats. This strategy for converting between a character data type and the Oracle Rdb data type DATE is illustrated in Figure 4–1. This figure also shows the declarations of date_var and P_DATE_CHAR and the column definition of DATE_COL.

Alternatively, your host programming language may support the conversion that you want between data storage formats. Consult the documentation for the host language that you are using to determine what capabilities are available in your host language for data type conversions.

Reference Reading

The *Oracle Rdb7 SQL Reference Manual* has a section about data type declarations supported for various host languages in the chapter on the SQL module language.

4.2.6 Effect of the LANGUAGE Clause on the Parameter Data Type

The procedures you include in an SQL module contain declarations of parameters whose data types must normally correspond to data types of actual parameters declared in the calling module. See Section 4.2.2 for more information about procedure parameters.

If you specify a data type for a procedure parameter that is unsupported by the language compiler that you identify in the LANGUAGE clause of the SQL module header, the SQL module processor returns a compile-time warning. For example, SQL supports the BIGINT data type but the PL/I language compiler does not. By returning this warning, the module processor assumes that you intend to store a value in a calling module parameter that your programming language will not allow you to declare. (In most cases, procedure parameters and actual parameters should have the same data type.)

You may choose to ignore this compile-time warning in certain cases. For example:

- Suppose a column is defined as BIGINT, a data type that your programming language does not support. However, this data type is supported by a forms product that you intend to call from your program to interpret the value. In this case, you may choose to define an 8-byte actual parameter any way your programming language allows you to (perhaps as

an 8-character text string) and define both the procedure parameter in the SQL module and the associated form field as BIGINT. At run time, SQL and the forms product correctly retrieve and store the BIGINT value stored in the 8-byte storage area. The run-time routines of the programming language never process the value.

- Suppose that your programming language allows you to create a user-defined data type that matches a data type for which the run-time routines of the language provide no support. In this case, you can ignore the compile-time warning if you plan to create your own routines to correctly interpret and process a value in a particular storage area.

In other cases, however, you should declare parameters in an SQL module using a data type that is supported by the programming language that calls that procedure. Therefore, for the majority of cases, the SQL module processor warning that a programming language does not support a data type indicates either that something is wrong with the declaration of your procedure parameter, or that you specified the wrong language in the LANGUAGE clause. See Chapter 8 for more information about data type declarations and conversions supported for various languages. In addition, see the tables in the *Oracle Rdb7 SQL Reference Manual* that show the correspondence between SQL data types and data types of host languages.

If you specify GENERAL as a language identifier, the SQL module processor allows any data type considered valid by SQL. In this case, you receive no warnings at compile time that you are declaring parameters in a way that may cause run-time errors or unexpected results. This is a disadvantage if you do not fully understand the capabilities and restrictions of the host language you are using.

4.2.7 Effect of the LANGUAGE Clause on the Parameter-Passing Mechanism

The language identifier implicitly specifies the mechanism SQL expects for passing parameter values to and from the host language module at run time. For all language identifiers except GENERAL, SQL expects parameters to be passed from the calling module using the defaults for that language. When you specify GENERAL, SQL expects all parameters to be passed by reference from the calling module. The advantage of identifying the language that will call procedures in the SQL module is that, for the most part, the programmer writing a calling module does not have to worry about identifying the parameter-passing mechanism expected by the called module.

Usually, default passing mechanisms for most languages vary from one data type to another. Your host language documentation for an external procedure call contains information about default passing mechanisms. You need to study this information if either of the following statements is true:

- In the LANGUAGE clause of the SQL module being called, you specify a language other than the one in which the calling module is written, or you specify GENERAL.
- Your call to a procedure in an SQL module specifies host language parameters that use language-specific subtypes of a generic data type, and SQL supports only the generic data type.

Note

Some languages include subtypes for a given data type and may assign different default passing mechanisms to each subtype. When host language parameter declarations specify these subtypes, you may need to override default passing mechanisms for those subtypes in either the SQL or host language module.

To illustrate the point in the preceding note, when you use the text string data type, Ada allows you to declare strings that include the number of characters (for which the default passing mechanism is by reference) as well as strings that do not include the number of characters (for which the default passing mechanism is by descriptor).

In SQL modules, however, you cannot declare text string data types without specifying the number of characters. Therefore, when you specify LANGUAGE ADA in an SQL module, SQL expects text string parameters to be passed always by reference. This means that if you declare a formal parameter as a string in your Ada declaration of the SQL procedure, and you do not specify the number of characters for that string, you must override the passing mechanism default for the parameter either in your Ada module or in your SQL module.

SQL does not support the by-value passing mechanism. When calling procedures in an SQL module, you *must* pass parameters either by reference or by descriptor.

See Section 4.2.8 for more information about parameter-passing mechanisms.

4.2.8 Overriding the Default Passing Mechanism for a Procedure Parameter

If you want a parameter value to be passed by descriptor, you can specify the BY DESCRIPTOR clause as part of a procedure parameter declaration in an SQL module. For example:

```
SIZE_PARAMETER   INTEGER  BY DESCRIPTOR
```

If you do not specify the BY DESCRIPTOR clause, the passing mechanism for a parameter depends on the language specified in the LANGUAGE clause of the module. If the LANGUAGE clause identifies the calling language, passing mechanism defaults are the same for the calling host language module and the called SQL module.

Note the following reasons to specify the BY DESCRIPTOR clause in a procedure parameter declaration in an SQL module:

- You plan to override a by-reference default when you specify an actual parameter in the host language module. In this case, you must explicitly specify a by-descriptor passing mechanism for the corresponding parameter in the SQL module as well.
- You want to specify the CHECK keyword in a declaration of a procedure parameter to ensure that the calling module is passing a parameter as expected by the SQL module. Section 4.2.9 discusses run-time checking of parameter-passing mechanisms.
- SQL syntax has an exact counterpart of only one subtype of a generic data type (such as text string), and you declare a formal parameter using a subtype whose default passing mechanism is different from the one that you can match in SQL syntax. In this case, you must override the default passing mechanism for the parameter in either the SQL or the host language module. (If the SQL default is by descriptor, you can change the passing mechanism default only in the host language module.)

4.2.9 Requesting a Run-Time Check of Parameters Used by the Calling Module

When the host language module declares actual parameters that do not match the data type and size of the corresponding procedure parameters in an SQL module, run-time results are undefined. When you are debugging your program, it is useful to receive a run-time error to inform you when the procedure and actual parameters do not agree.

To enable return of this run-time error, specify the BY DESCRIPTOR CHECK clause in the parameter declaration in the SQL module whose corresponding host language parameter you want SQL to check. You must include the BY DESCRIPTOR CHECK clause even if the default passing mechanism is by descriptor. For example:

```
SIZE_PARAMETER INTEGER BY DESCRIPTOR CHECK
```

If the by-descriptor mechanism is not the default passing mechanism, specify the by-descriptor passing mechanism for the parameter in the host language module as well.

4.3 Using Single SQL Statements in Procedures

An SQL module procedure that includes only one executable SQL statement is called a simple statement procedure. In a simple statement procedure, you enter a semicolon (;) twice: the first occurrence follows the last parameter declaration and the second occurrence is at the end of the executable SQL statement. The following example of a simple statement procedure fetches data from an open cursor:

```
PROCEDURE FETCH_REPORT_RECORD       -- Procedure name
    (                               -- Beginning parenthesis
    SQLSTATE,                       -- Status parameter with no data type
    :P_EMPLOYEE_ID      CHAR(5),    -- Parameters prefixed by colons
    :P_LAST_NAME        CHAR(14),   -- and separated by commas
    :P_FIRST_NAME       CHAR(10),
    :P_JOB_CODE         CHAR(4),
    :P_DEPARTMENT_CODE  CHAR(4),
    :P_SALARY_AMOUNT    REAL
    );                               -- Ending parenthesis and semicolon

    FETCH REPORT_CURSOR INTO         -- Simple statement
    :P_EMPLOYEE_ID, :P_LAST_NAME,
    :P_FIRST_NAME, :P_JOB_CODE,
    :P_DEPARTMENT_CODE, :P_SALARY_AMOUNT; -- Ending semicolon
```

4.4 Using Compound Statements in Multistatement Procedures

An SQL module procedure that includes one or more SQL statements in a compound statement is called a multistatement procedure.

Simple statement procedures restrict database access to a single statement at a time. By using a multistatement procedure and only one host language program call, you can perform comprehensive business transactions, complex program control, and structured error handling all within the database environment. In many cases, you can achieve function abstraction, which is when one physical module performs one logical database function.

In a multistatement procedure, you enter a semicolon (;) following the last parameter declaration, at the end of each SQL statement contained in the compound statement, and at the end of the compound statement itself.

The following example shows a multistatement procedure that contains a FOR loop with a nested IF statement. The procedure increases the minimum salary for some jobs and then, if the maximum salary is less than the minimum salary, it increases the maximum salary. In addition, it counts the number of rows updated for each category.

```
PROCEDURE salary_inc                    -- Procedure name
  (                                     -- Beginning parenthesis
   SQLSTATE,                            -- Status parameter with no data type
   :inc REAL,                           -- Parameters prefixed by colons
   :min_count SMALLINT,                 -- and separated by commas
   :max_count SMALLINT
   );                                   -- Ending parenthesis and semicolon

BEGIN                                   -- Beginning of compound statement

   SET :min_count = 0;                  -- Assignment statements
   SET :max_count = 0;

-- FOR statement.
-- The :jobrec variable represents a record that holds columns from the
-- selected row.
   FOR :jobrec
       AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR

-- The select expression.
          SELECT MINIMUM_SALARY, MAXIMUM_SALARY FROM JOBS
                 WHERE MINIMUM_SALARY < 20000
       DO
-- Update the current row in the JOB_CURSOR
             UPDATE JOBS
                 SET MINIMUM_SALARY = MINIMUM_SALARY + (MINIMUM_SALARY * :inc)
                 WHERE CURRENT OF JOB_CURSOR;
             SET :min_count = :min_count +1;

-- Nested IF statement.
-- If the minimum salary is now greater than the maximum salary, increase the
-- maximum salary.
          IF :jobrec.MINIMUM_SALARY  >  :jobrec.MAXIMUM_SALARY
          THEN
             UPDATE JOBS
                 SET MAXIMUM_SALARY = MAXIMUM_SALARY * :inc
                 WHERE CURRENT OF JOB_CURSOR;
             SET :max_count = :max_count +1;

          END IF;              -- End of IF statement, terminated by semicolon

   END FOR;                    -- End of FOR statement, terminated by semicolon

END;                           -- End of compound statement, terminated by
                               -- semicolon
```

For information about writing compound statements, see Chapter 12.

4.5 Understanding the Restrictions of the SQL Module Language

When you use SQL module language, SQL enforces the following restrictions:

- You cannot continue a keyword, user-defined name, or literal (such as a quoted string) from one line to the next. You must enter them in their entirety on one line of your SQL module file.
- You cannot specify a WHENEVER statement. Furthermore, you cannot specify the WHENEVER statement in a host language source file and expect it to apply to your calls to procedures in SQL modules. The WHENEVER statement is supported only by the SQL precompiler, which can identify only SQL statements embedded in a host language source file. The SQL precompiler does not recognize user-defined procedure calls as being related to SQL operations.

  Handle errors returned by a call to a procedure in an SQL module in the calling host language module. Use a host language conditional statement to evaluate the SQL status parameter, such as SQLSTATE or SQLCODE, immediately following the call.
- If the module contains a CREATE DATABASE statement, it should appear lexically before any DECLARE TABLE statement. If the DECLARE TABLE statement appears before the CREATE DATABASE statement, SQL tries to attach to the default database (such as SQL$DATABASE).
- Data definition statements cannot refer to database objects that do not precede the data definition statement or are not defined in the compile time database. For example, because a CREATE STORAGE MAP statement refers to a table, the table must exist in the compile-time database or the SQL module must create it in a statement that lexically precedes the CREATE STORAGE MAP statement.
- You cannot use the SQL INCLUDE statement to copy host language declarations from the repository or text source file. However, you can use the FROM path-name clause in an SQL module procedure to copy record definitions from the repository, and you can use a host language INCLUDE or COPY statement to copy host language definitions into a host language source file.
- When you specify the language as C in the LANGUAGE clause, SQL translates all C character strings as null-terminated strings. This means that when SQL passes these character strings from the database to the program, it reserves space at the end of the string for the null character. When a program passes a character string to the database for input, SQL looks for the null character to determine how many characters to store

in the database. SQL stores only those characters that precede the null character; it does not store the null character itself.

Because of the way SQL translates C character strings, you may encounter problems with applications that pass binary data in C character strings to and from the database. The binary data, for example, might contain null characters that would cause SQL to prematurely truncate the data. To avoid problems of this type when you use the SQL module language with a C host language program, you might want to consider specifying the module language as GENERAL. In most cases, however, you do not need to use GENERAL in place of C as the module language.

Some of these restrictions are also discussed in Section 8.6, Section 8.12.2, and Section 10.2.

5

Processing SQL Modules and Host Language Files

This chapter describes how to process SQL modules and host language files. In the sections that follow, you will become familiar with how to:

- Invoke the SQL module processor
- Process SQL modules and host language modules
- Bypass parameter checking for faster compilation
- Improve the performance of the SQL module processor when you access remote databases
- Use context files with SQL module language
- Decide on the scope of an SQL module

5.1 Invoking the SQL Module Processor

The SQL module processor is an executable image that checks and processes statements in an SQL module source file. If the module processor encounters no fatal errors, it produces an object file. To use the module processor, you invoke the module processor and specify the SQL module source file as the input filename. The default file extension for the SQL module source file is .sqlmod.

Digital UNIX

On Digital UNIX, invoke the module processor using the `sqlmod` command and specify the input filename. For example, to process the SQL module file my_program.sqlmod, use the following command:

```
$ sqlmod   my_program.sqlmod
```

By default, the SQL module processor produces an object file with the file extension .o as input to the linker. ♦

OpenVMS VAX OpenVMS Alpha

On OpenVMS, run the SQL module processor by defining the following symbol in your login command file or at the DCL command level:

```
$ SQLMOD :== $SQL$MOD
```

When you define SQLMOD as the symbol shown in the preceding example, you can specify the input file on the command line or have the SQL module processor prompt you for it. For example:

```
$ SQLMOD my_program.sqlmod
```

```
$ SQLMOD
INPUT FILE> my_program
```

The SQL module processor produces an object file with the file extension .obj as input to the OpenVMS Linker utility. ♦

In addition to the input filename, you can specify an optional context file on the SQL module processor command line. (See Section 5.5 for information about context files.)

The SQL module processor command line can also include a number of qualifiers, all of which are described fully in the *Oracle Rdb7 SQL Reference Manual*.

Digital UNIX

The following example shows how to use the –o qualifier to specify the name of the resulting object file and the –list qualifier to specify the name of the listing file on a Digital UNIX system:

```
$ sqlmod -list my_program.lis -o my_mod_program.obj
```
♦

If the SQL module processor encounters a fatal error, the compilation stops. To find multiple fatal errors, you must perform multiple compilations.

OpenVMS Alpha

When an SQL module language program compiles with errors on OpenVMS Alpha, the .obj files are not deleted, as they are on OpenVMS VAX and Digital UNIX. ♦

5.2 Processing SQL and Host Language Modules

Processing a program that calls procedures in SQL modules involves four steps:

1. Processing (compiling) the SQL module file into an object file

   Use the SQL module processor to compile SQL source files, as described in Section 5.1.

2. Processing (compiling) the host language source file into an object file

   Use the appropriate host language compiler to compile host language source files.

3. Linking object files to create an executable or shareable image

   Link the object files by processing them with the linker, which is called by the host language compiler. You must link the object files with the SQL libraries.

4. Running the executable image or (if necessary) installing the shareable image

Digital UNIX

The following example creates an executable image on Digital UNIX, using the Bourne shell:

```
$ # Invoke the SQL module processor.
$ sqlmod my_sql_module
$
$ # Define a global symbol for the SQL libraries.
$ SQLLIBS='-lrdbsql -lrdbshr -lcosi -lots'
$ export SQLLIBS
$
$ # Invoke the Digital C compiler and link together the SQL module object
$ # file, the host language object file, and the SQL libraries.
$ cc -o my_calling_module my_calling_module.c my_sql_module ${SQLLIBS}
$ # Run the application.
$ my_calling_module
```

♦

OpenVMS VAX OpenVMS Alpha

The following example creates an executable image on OpenVMS:

```
$ ! Define a symbol to invoke the module processor.
$ SQLMOD :== $SQL$MOD
$ ! Invoke the SQL module processor.
$ SQLMOD my_sql_module
$ ! Invoke the Pascal compiler.
$ PASCAL my_calling_module
$ ! Link the files.
$ LINK  my_calling_module, my_sql_module
$ ! Run the application.
$ RUN my_calling_module
```

The preceding example assumes you have defined the logical name LNK$LIBRARY to be SQL$USER.OLB. ♦

The following sections discuss compiling SQL source files. See your host language documentation for instructions on compiling host language source files. Note that Ada source files are not compiled in the way that the preceding example indicates. See the Ada documentation set for information about compiling Ada source files.

See Chapter 7 for information about the final phases of program development.

5.3 Bypassing Parameter Checking for Faster Compilation

OpenVMS VAX OpenVMS Alpha

The [NO]PARAMETER_CHECK qualifier determines whether or not the SQL module processor compares the number of formal parameters declared for a procedure with the number of parameters specified in the SQL statement of the procedure during compilation. The qualifiers are as follows:

- PARAMETER_CHECK (default)

  Checks that parameter counts match and generates an error at run time (not compile time) when they do not.

- NOPARAMETER_CHECK

  Suspends checking of parameters to improve module compilation time. Consider using the NOPARAMETER_CHECK qualifier after you have debugged your SQL module. ♦

5.4 Improving SQL Module Processor Performance for Remote Databases

The SQL module processor must access the definitions of tables and views to which you refer in programs. By default, the module processor accesses database system tables to retrieve table and view definitions. When a database is remote, compiling the SQL module may require more time and additional resources to retrieve definitions than when a database is local.

When accessing a remote database, you have three options for improving the performance of the SQL module processor. Read Section 15.1.3 for a complete discussion of the first two options. The third option is discussed here.

- Use the COMPILETIME option of the DECLARE ALIAS statement to specify a path name of a node created in a local repository for the database.
- Use the COMPILETIME option of the DECLARE ALIAS statement to specify a local Oracle Rdb database file.
- Use DECLARE TABLE statements in your programs to specify definitions for the tables and views that you name in SQL statements.

  The SQL module processor does not access the database for table definitions that you declare in this way. If your DECLARE TABLE statements include constraints that refer to other tables, be sure to declare those tables as well. In addition to being an option for improving compile-time performance, DECLARE TABLE statements provide a way to compile programs that will access tables not yet created.

Reference Reading

For detailed information about the DECLARE TABLE statement, see the DECLARE TABLE section of the chapter on SQL statements in the *Oracle Rdb7 SQL Reference Manual*.

For a discussion of the compile-time options for database attachment, see the DECLARE ALIAS section of the chapter on SQL statements in the *Oracle Rdb7 SQL Reference Manual*.

5.5 Using Context Files with SQL Module Language

You can use SQL context files with SQL module language. A **context file** is an SQL command procedure containing DECLARE statements that you want to apply when your program compiles and executes. Context files can help you improve the portability of compiled source files.

With one exception, the format of a context file used with SQL module language is the same as that used for precompiled SQL. It is not necessary to end the DECLARE statements with a semicolon (;) when you use a context file with SQL module language. However, if you include the semicolon (;), you can use the context file with both SQL module language and precompiled SQL.

Assume that an application contains a module that must be compiled using different SQL dialects. Rather than having two copies of the module and the problem of maintaining them in parallel, you can have one module and two context files. The module contains all of the code and each context file contains the dialect declaration statement.

For example, the module test must be compiled using the dialects SQL92 and MIA. The context file test-sql92 contains the following DECLARE MODULE statement:

```
DECLARE MODULE
   DIALECT SQL92
```

The context file test-mia contains the following DECLARE MODULE statement:

```
DECLARE MODULE
   DIALECT MIA
```

The dialect is controlled by compiling the module test with the appropriate context file. For the module test to use the SQL92 semantics, compile the module test using the test-sql92 context file. Specify the context file as the second parameter, as shown in the following example:

```
$ SQLMOD
INPUT FILE> test test-sql92
```

For the module test to use the MIA semantics, compile it using the test-mia context file, as shown in the following example:

```
$ SQLMOD
INPUT FILE> test test-mia
```

5.6 Deciding on the Scope of an SQL Module

An executable or shareable image can be assembled from more than one host language module and more than one SQL language module. You can organize SQL operations into SQL source files in the following ways:

- You can associate SQL modules with executable images.

  In this case, you create one SQL source module to be called by any of the host language source modules that contribute to the image. This strategy is easiest for a single application. However, if calling modules are written in a variety of programming languages, the strategy has the disadvantage of establishing SQL passing mechanisms for one, several, or all data types that may not be the same as the calling module defaults.

  If the default passing mechanism for a given data type in a calling module is different from the default passing mechanism expected by the SQL module, you must override the default in one of those modules. (See Section 3.2.5, Section 4.2.7, and Section 4.2.8 for information on controlling how parameters are passed between modules.)

- You can associate each SQL module with only one kind of task.

  For example, you might want to associate only one SQL module with the update of a set of columns contained in one table or in several tables that are related by a constraint. In this case, if the task in question also defines the scope of a single calling host language source file, both the SQL module and the host language module that calls it have a one-to-one correspondence with a specific kind of task. As the number of database applications increases, this strategy makes it easier to keep track of how modules relate to one another and simplifies program maintenance.

6
Using Precompiled SQL

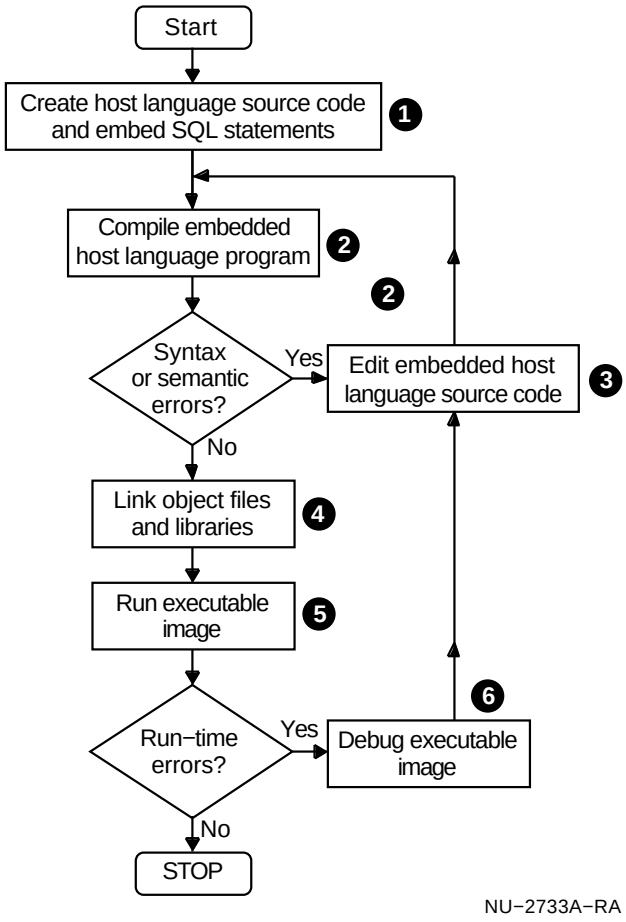
This chapter describes both general and language-specific guidelines for embedding SQL statements in host language source files. You will become familiar with:

- The process you follow when writing programs that contain SQL statements
- Writing a host language program that contains embedded SQL statements
- Invoking the SQL precompiler
- Finding compilation errors
- Improving performance for remote databases
- Specifying compile-time and run-time characteristics
- Embedding SQL statements in each language supported by the SQL precompiler

6.1 Understanding the Precompiler Process

The SQL precompiler lets you embed SQL statements in host language programs. The flowchart in Figure 6–1 illustrates the steps you follow in creating a program with precompiled SQL statements. The callouts are explained in the list following the figure.

Figure 6–1 Application Program Development with the SQL Precompiler



The callouts contained in Figure 6–1 are described as follows:

❶ Create the host language program and embed SQL statements in the code, prefixing each SQL statement with the EXEC SQL flag.

❷ Invoke the SQL precompiler to process the precompiled SQL statements and the host language program.

❸ If compilation errors occur, edit the source code. You return to this step if errors occur (step ❻) from the run phase. After editing the source code, return to the compilation phase (step ❷).

❹ Link the object files and libraries to create an executable image or a shareable image or shared module. (You link shareable images or shared modules with other programs.) Chapter 7 explains both options.

❺ Test the executable image created from the link phase (step ❹) to verify that the program works properly. You may need to install the shareable image or link with other modules.

❻ If you discover any run-time errors, debug the executable image and return to the edit phase (step ❸) to correct the host language source code or the embedded SQL statements.

Digital UNIX

The following example creates an executable image, using the C language source file called my_program.sc, on Digital UNIX using the Bourne shell:

```
$ # Invoke the SQL precompiler and specify the language as C.
$ sqlpre -l cc my_program.sc
$
$ # Define a global symbol for linking.
$ SQLLIBS='-lrdbsql -lrdbshr -lcosi -lots'
$ export SQLLIBS
$
$ # Invoke the DEC C compiler and link the object file.
$ cc -o my_program my_program.o ${SQLLIBS}
$ # Run the application.
$ my_program                                          ♦
```

OpenVMS VAX OpenVMS Alpha

The following example creates an executable image, using the C language source file called my_program.sc, on OpenVMS:

```
$ ! Invoke the SQL precompiler and specify the language as C.
$ SQLPRE
INPUT FILE> my_program /CC
$ ! Link the file.
$ LINK my_program
$ ! Run the application.
$ RUN my_program
```

On OpenVMS, the logical name LNK$LIBRARY must be defined as SQL$USER before you invoke the linker. ♦

Reference Reading

For information about linking object modules into images and running executable images, see Chapter 7.

6.2 Embedding SQL Statements in Host Programs

The SQL precompiler allows you to embed both simple SQL statements and compound statements in your host language program:

- Simple SQL statements

  A simple SQL statement consists of a single SQL statement. You embed a simple SQL statement by preceding the statement with the EXEC SQL flag and terminating it with the language-specific termination character, as shown in the following excerpt from a C precompiled program:

```
EXEC SQL UPDATE DEPARTMENTS
                SET MANAGER_ID = :mgrid
                WHERE DEPARTMENT_CODE = 'SALE';
```

- Compound statements

  A compound statement consists of one or more SQL statements delineated with the keywords BEGIN and END. You can include flow-control statements, such as the SQL IF and FOR statements, in compound statements. As you do with simple SQL statements, you embed a compound statement by preceding the statement with the EXEC SQL flag and terminating the entire compound statement with the language-specific termination character. In addition, you end each SQL statement within the compound statement with a semicolon (;).

  Chapter 12 explains how to use compound statements with the SQL precompiler.

The SQL precompiler processes programming language source files and imposes certain rules on how to include SQL statements in those files. These rules apply to any precompiled file:

- Begin each SQL statement with the EXEC SQL flag.

  The EXEC SQL flag identifies the beginning of SQL statements to the SQL precompiler. The EXEC SQL flag can occur only at the beginning of the first line of a multiline SQL statement. (The need for host language continuation characters for multiline SQL statements is a language-specific need. Follow the rules of the programming language you are using to continue SQL statements from one line to the next.)

- The EXEC SQL flag can occur on the same line as other host language elements. For example, an SQL statement might come before a label for a section in your program. The EXEC SQL flag can also follow a language statement (an IF statement, for example) on the same line.

- To continue multiline literals in SQL statements from one line to another, follow the host language rules for continuation of literals.

- Specify INCLUDE statements that copy executable host language statements, executable SQL statements, or both, in a part of your program that allows executable statements. The INCLUDE SQLCA, INCLUDE FROM repository, and INCLUDE statements that copy host language declarations must appear in a part of your program that allows variable declarations.

  Within a variable declaration, you cannot specify INCLUDE statements that copy partial host language declarations. For example, assume that FILE.DAT contains the following lines:

```
05  FIELD1              PIC X(10).
05  FIELD2              PIC X(10).
```

  In this situation, the precompiler does not accept the INCLUDE statement contained in the following section of a COBOL program:

```
WORKING-STORAGE SECTION.
01  DEPT_REC                       PIC X(24).
01  COMMAREA
EXEC SQL INCLUDE 'FILE.DAT'    END-EXEC.
```

- You can embed DECLARE statements in any part of your program. However, the DECLARE ALIAS, DECLARE TABLE, and DECLARE TRANSACTION statements must come before statements that depend on the information those DECLARE statements contain. For example, if you explicitly declare databases and specify a transaction in your source file, a DECLARE ALIAS or DECLARE TABLE statement must precede a DECLARE (or SET) TRANSACTION statement, which must precede an OPEN statement in a source file. A DECLARE CURSOR statement, however, does not have to precede an associated OPEN statement in a source file.
- You must embed any CREATE DATABASE statements so that they appear lexically before a DECLARE TABLE statement. If the DECLARE TABLE statement appears before the CREATE DATABASE statement, SQL tries to attach to the default database (such as SQL$DATABASE).
- Data definition statements cannot refer to database objects that are not defined in the compile-time database or do not precede the data definition statement. For example, because a CREATE STORAGE MAP statement refers to a table, the table must exist in the compile-time database or the embedded SQL program must create it in a statement that lexically precedes the CREATE STORAGE MAP statement.

- The SQL precompiler follows language-specific rules for processing parameter names. For example, when processing C source files, the SQL precompiler treats uppercase and lowercase characters as different characters in C parameter names.

  However, the SQL precompiler always applies SQL rules to names of database entities and to SQL keywords other than those that name parameters.

  Therefore, regardless of the rules you apply to parameter names, you:

  - Must specify underscores (_) rather than hyphens when entering SQL keywords and names of database entities

    The END-EXEC flag used in COBOL is an exception to this rule. To comply with the ANSI/ISO SQL standard, specify the END-EXEC flag. The END_EXEC flag compiles but is not compliant with the standard.

  - Can specify either uppercase or lowercase letters when entering SQL keywords and names of database entities.

    However, SQLCODE, SQLSTATE, SQLCA, SQLDA, and SQLDA2 are keywords that refer to parameters processed by the host language as well as by SQL. If you are writing in a language, such as C, that applies case-sensitive rules to parameters, then you must specify SQLCODE, SQLSTATE, SQLCA, SQLDA, and SQLDA2 in uppercase characters.

- End each statement as required by your host language. Table 6–1 lists the rules for ending SQL statements in each host language.

Table 6–1 Ending SQL Statements in Precompiled Host Language Source Files

Language	Ending SQL Statements
Ada	End SQL statements with a semicolon (;).
C	End SQL statements with a semicolon (;).
COBOL	End SQL statements with the END-EXEC flag[1]. You can insert at least one space and then the END-EXEC flag after the last word in the SQL statement, or you can place the END-EXEC flag as the only word on the line immediately following the SQL statement.

[1] Depending on where the SQL statement occurs in the program structure, you may need to add a period (.) immediately following the END-EXEC flag. Add a period if a COBOL statement in the same position would require one.

(continued on next page)

Table 6–1 (Cont.) Ending SQL Statements in Precompiled Host Language Source Files

Language	Ending SQL Statements
FORTRAN	End SQL statements as you would FORTRAN statements. In other words, SQL statements end at a line terminator unless you specify a continuation character on the following line.
Pascal	End SQL statements with a semicolon (;), even those within a Pascal IF-THEN-ELSE statement. For example, if you embed an SQL statement before the ELSE clause, you must surround the SQL statement with a BEGIN-END block and the SQL statement ends with a semicolon (;).
PL/I	End SQL statements with a semicolon (;).

Section 6.7 provides additional rules and guidelines for each language supported by the SQL precompiler.

6.3 Invoking the Precompiler

To use the SQL precompiler, you invoke the precompiler and specify the input file name and the host language in which the source file is written.

Digital UNIX

On Digital UNIX, invoke the precompiler using the `sqlpre` command. In addition, specify the input file name, the `-l` qualifier and a language-specific qualifier. For example, to process a C host language source file, you specify the `-l` qualifier, along with the `cc` language-specific qualifier, as shown in the following example:

```
$ sqlpre -l cc my_program.sc
```

You can pass strings to the host language compiler by enclosing the string in quotes ('), as shown in the following example:

```
-l fortran='-extend_source'
```

SQL does not check that the string you pass is valid. ♦

OpenVMS VAX OpenVMS Alpha

On OpenVMS, to invoke the SQL precompiler, define a symbol in your login command procedure or at the DCL command level. You can define the following symbol to invoke the SQL precompiler:

```
$ SQLPRE :== $SQL$PRE
```

To invoke the SQL precompiler for processing a C language source file, specify a file name and the /CC language-specific qualifier with the SQLPRE symbol, as the following example shows:

```
$ SQLPRE
INPUT_FILE>  my_program.sc /CC                    ♦
```

Table 6–2 shows the default input file types and language identification qualifiers you can use when you precompile a program. The figure also shows the default file types used by the precompiler when it passes the program to a host language compiler.

Table 6–2 Language Identifiers and Default File Extensions Used in Precompiling Programs

	SQL Precompiler				Host Language Compiler		
Host Language	**Input File**	**OpenVMS Qualifier**	**Digital UNIX Qualifier**	**Output File**	**Input File**	**OpenVMS Output File**	**Digital UNIX Output File**
Ada	.sqlada	/ADA	n/a	.ada	.ada	.obj	n/a
C	.sc	/CC	-l cc	.c	.c	.obj	.o
COBOL	.sco	/COBOL	-l cobol	.cob	.cob	.obj	.o
FORTRAN	.sfo	/FORTRAN	-l fortran	.for	.for	.obj	.o
Pascal	.spa	/PASCAL	-l pascal	.pas	.pas	.obj	.o
PL/I	.spl	/PLI	n/a	.pli	.pli	.obj	n/a

OpenVMS VAX OpenVMS Alpha

On OpenVMS, you can also define the symbol that invokes the precompiler to be language-specific. In this case, the symbol definition also includes the language qualifier. The following example shows a sample symbol definition for the SQL precompiler and the COBOL language:

```
$ SCOB     :== $SQL$PRE/COB
```

SQL invokes the language compiler with any symbol you define for that compiler, provided that the symbols do not execute command procedures. The symbol COBOL is supported, for example, if it means COBOL/DEBUG, but is not supported if it means @cob_proc.

(Ada programs require that you create and enable an Ada library before you precompile them. See Section 6.7.2.)

You cannot pass the [NO]OBJECT or [NO]G_FLOAT qualifiers through a symbol you define to invoke the language compiler. These qualifiers must be specified at precompile time. For instance, if you use the following symbol definitions, the precompiler will not recognize the G_FLOAT qualifier:

```
$ MYC :== CC /GFLOAT
$ MYSQL :== SQLPRE/CC
$ MYSQL FILE_A
```

Either specify the [NO]OBJECT and [NO]G_FLOAT qualifiers as part of the definition of the symbol you use to invoke the SQL precompiler or include them on the command line when you use a symbol to invoke the SQL precompiler. ♦

Digital UNIX

On Digital UNIX, the SQL precompiler does not pass the following qualifiers to the host language compilers:

- `-list`
- `-match`
- `-form ansi`

You must explicitly specify the appropriate qualifier in the compiler switch list. For example, for the COBOL host language compiler, you must specify `-ansi` in order to use the `-form ansi` qualifier:

```
$ sqlpre -l cobol="-ansi" -form ansi
```

See your compiler documentation for information about the appropriate qualifiers to use in the compiler switch list. ♦

The command you enter to invoke the SQL precompiler can include two parameters:

- The file specification for a host language source file (required)

  The host language file must contain both host language code and precompiled SQL statements. The default file type for the source file depends on the host language specified in the language qualifier, as shown in Table 6–2.

- The file specification of an SQL context file (optional)

  SQL defines a **context file** as a special-purpose command procedure containing declarations that you want to apply when your program precompiles and executes. You need to use context files only if you are creating precompiled source files that you expect to port from one implementation of SQL to another. For a discussion of context files, refer to Section 6.6.2.

The SQL precompiler command line includes both precompiler and host language compiler qualifiers:

- SQL precompiler qualifiers

  You must include a precompiler qualifier to specify the host language in which the source is written. In addition to the language identification qualifier, you can specify a number of other qualifiers on the SQL precompiler command line. These qualifiers are documented fully in the SQL precompiler chapter of the *Oracle Rdb7 SQL Reference Manual*.

- Host language compiler qualifiers

  You can specify language compiler qualifiers that you want the precompiler to use when it submits a preprocessed source file to the language compiler. Refer to your host language documentation for a description of the language compiler qualifiers you can use on the precompiler command line.

The precompiler command line cannot exceed 255 characters.

Caution

Do not edit the language source module created by the SQL precompiler and then use the host language compiler to compile that source module. This rule applies even if you want to make source changes that do not affect SQL statements, because the next time you precompile the source file, the original precompiled SQL module will write over any changes that you made to the temporary language source module generated by the precompiler.

OpenVMS VAX

On OpenVMS VAX, the precompiler:

1. Creates two source files from the file you submit to it:
   - A macro module (file type .mar) that translates SQL statements into macro procedures.
   - A host language module that a language compiler can process. In this host language module, the precompiler replaces the SQL statements with calls to the external procedures in the macro module.
2. Invokes the macro assembler to create an object file (file type .mob) for the macro module and submits the preprocessed language source file to the appropriate language compiler.

3. Combines into one file both the object file output by the language compiler and the object file for the macro module. The precompiler then deletes the .mar and .mob files. (The precompiler does not perform this concatenation of object files when it processes Ada programs. See Section 6.7.2 for information about precompiling Ada programs.) ♦

OpenVMS Alpha

On OpenVMS Alpha, the precompiler creates a single output file, which it submits to the appropriate language compiler. The precompiler does not create a macro module on OpenVMS Alpha. ♦

Digital UNIX

On Digital UNIX, the precompiler creates a single output file, which it submits to the appropriate language compiler. The precompiler does not create a macro module on Digital UNIX. ♦

6.4 Finding Precompile-Time and Compile-Time Errors

If the SQL precompiler finds an error while it is processing your program, it displays the error on your terminal or in a log file and writes the error to the host language source file. Refer to the error message appendix in the *Oracle Rdb7 SQL Reference Manual* for ways you can access supplementary information for error messages.

If the host language compiler finds an error, the compiler writes the error to a location that the SQL precompiler can access. The SQL precompiler retrieves the error message, displays it on your terminal or in the log file, and writes it to a log file named sqlerr.log. If you specify the LIST or `-list` qualifier, the host language compiler also writes the error message to the listing file.

Note

The SQL precompiler produces the file sqlerr.log only after it submits precompiled source code to the host language compiler. Depending on the errors it encounters, the precompiler may not submit source code to the host language compiler, and therefore, may not produce the file sqlerr.log.

When the precompiler does not produce an sqlerr.log file, it produces a host language source file that you can display or edit to review errors. For example, if you submit a file named myprog.sfo to the precompiler, you can type or edit the file myprog.for to review precompile-time errors.

Consult your host language documentation for supplementary information about compile-time errors.

6.5 Improving Precompiler Performance for Remote Databases

The SQL precompiler must access definitions of the tables and views your program will access at run time. By default, the precompiler accesses system tables in a database file to retrieve table and view definitions. When a database is remote, precompiling may take longer and use more resources to retrieve definitions than when a database is local.

When accessing a remote database, you have three options for improving precompiler performance. Read Section 15.1.3 for a complete discussion of the first two options. The third option is discussed here.

- Use the COMPILETIME option of the DECLARE ALIAS statement to specify a a path name of a node created in a local repository for the database.
- Use the COMPILETIME option of the DECLARE ALIAS statement to specify a local Oracle Rdb database file.
- Use the DECLARE TABLE statement in your programs to specify definitions for the tables and views you name in SQL statements.

  Because the definitions are included as part of the program, the SQL precompiler does not need to access the database. To reduce maintenance of multiple source files when using DECLARE TABLE statements, specify those statements in one text file. Use the SQL INCLUDE file-spec statement to copy that text file into your programs.

  If your DECLARE TABLE statements include constraint definitions that refer to other tables, be sure to declare those tables as well.

Reference Reading

For detailed information about the DECLARE TABLE and DECLARE ALIAS statements, see the chapter on SQL statements in the *Oracle Rdb7 SQL Reference Manual*.

6.6 Specifying Compile-Time and Run-Time Options

SQL allows you to control many settings through precompiler qualifiers and the DECLARE MODULE statement. These settings affect characteristics such as the character set of the data you will be processing and whether the precompiler should flag syntax that does not conform to the ANSI/ISO SQL standard.

Refer to the *Oracle Rdb7 SQL Reference Manual* for a complete list of precompiler qualifiers and the rules for their use. The following sections describe some of the ways you can use these settings.

6.6.1 Using the DECLARE MODULE Statement

You can affect a variety of compile-time and run-time SQL settings for precompiled programs by including a DECLARE MODULE statement directly in a host language program or in a context file. Example 6–1 shows a DECLARE MODULE statement that can be included in a host language program.

Example 6–1 Changing Compile-Time and Run-Time Settings with the DECLARE MODULE Statement

```
EXEC SQL DECLARE MODULE employee_module
    DIALECT SQL92
    NAMES ARE DEC_KANJI
    NATIONAL CHARACTER SET KANJI
    CATALOG ADMINISTRATION
    SCHEMA ACCOUNTING
    DEFAULT CHARACTER SET DEC_KANJI
    AUTHORIZATION RDB$DBHANDLE
    CHARACTER LENGTH OCTETS
    ALIAS RDB$DBHANDLE;
```

You can specify the following categories of options with the DECLARE MODULE statement:

- Dialect options

  The DIALECT clause controls in one clause the settings for several other clauses, similar to throwing a single electrical switch to control a bank of lights. Refer to the *Oracle Rdb7 SQL Reference Manual* for a listing of the settings that the DIALECT clause affects.

- Character set options

  Character set options control the default, identifier, national, and literal character sets for a module.

- Module language options

  Module language options consist of a number of clauses, such as ALIAS and RIGHTS, several of which can be set with the DIALECT clause.

- Authorization identifier and multischema options

These options specify the default authorization identifier, schema, and catalog.

The *Oracle Rdb7 SQL Reference Manual* describes the DECLARE MODULE statement in detail and the clauses you can specify with it.

6.6.2 Including Declarations in an SQL Context File

Context files can make it easier to move your precompiled SQL programs to other platforms. A **context file** is an SQL command procedure containing DECLARE statements that you want to apply when your program compiles and executes.

For example, on OpenVMS to precompile a COBOL source file named test.sco and apply to it DECLARE statements in the test_declares.sql file, use the following command:

```
$ SQLPRE/COBOL test test_declares
```

Example 6–2 shows the context file test_declares.sql, which contains declarations for precompiling a source file.

Example 6–2 Context File for Precompiled SQL Compilation

```
DECLARE ALIAS FILENAME pers;

DECLARE TRANSACTION READ WRITE
        RESERVING EMPLOYEES FOR PROTECTED WRITE,
                  JOB_HISTORY FOR PROTECTED WRITE,
                  DEPARTMENTS FOR SHARED READ,
                  JOBS FOR SHARED READ;

DECLARE MODULE employee_module
    DIALECT SQL92
    NAMES ARE DEC_KANJI
    NATIONAL CHARACTER SET KANJI
    CATALOG ADMINISTRATION
    SCHEMA ACCOUNTING
    DEFAULT CHARACTER SET DEC_KANJI
    AUTHORIZATION RDB$DBHANDLE
    CHARACTER LENGTH OCTETS
    ALIAS RDB$DBHANDLE;
```

You can use the same context file with more than one source file, or you can use different context files for each source file, or you can use different context files with one source file.

Assume that an application contains a module that must be compiled using different SQL dialects. Rather than having two copies of the module and the problem of maintaining them in parallel, you can have one module and two context files. The module contains all of the code and each context file contains the dialect declaration statement.

The dialect is controlled by compiling the module test with the appropriate context file.

The format of a context file is the same as any SQL command procedure; DECLARE statements in the file are not preceded by the EXEC SQL flag (as they are in SQL statements embedded in a source file) and they always end with a semicolon (;).

You can include other DECLARE statements, such as DECLARE TABLE and DECLARE STATEMENT, in a context file to improve program portability.

If you use a context file, you should note in your host language source file that DECLARE statements needed for precompiling the program are included in a context file.

6.7 Language-Specific Guidelines for Using the SQL Precompiler

This section contains language-specific guidelines for the languages supported by the SQL precompiler. Those languages are Ada, C, COBOL, FORTRAN, Pascal, and PL/I.

6.7.1 Embedding SQL Statements in Ada Source Files

This section contains information that applies only to Ada programs. Note that Ada programs require special considerations for precompiling and linking (see Section 6.7.2 and Section 7.2.3). Refer also to Section 8.12.1 for information about declaring and using parameters specific to the Ada language.

Limiting the Length of File Names

Limit the length of the file name of an Ada precompiler file to 27 characters. The Ada compiler limits file names to 31 characters; however, the SQL precompiler adds the prefix "SQL_" to the file name to create a package name.

Named Literals or Ranges

The Ada precompiler does not support the use of named literals or ranges. To avoid this restriction, use the SQL module language.

Embedding SQL Statements in Ada Files That Contain Multiple Procedures

The SQL precompiler supports block structure in Ada programs. As a result, you can declare parameters to which SQL statements refer, such as SQLCODE, in multiple procedures in the same Ada source file, and the precompiler will recognize them as independent parameters.

Overloading of Subprograms

The SQL Ada precompiler does not support the overloading of subprograms. It treats all of the overloaded programs as a single name space.

One workaround is to make sure that all names used in SQL statements are unique in all the overloaded procedures. To conform to the ANSI/ISO standard, the names of all host language variables must be unique in the entire program.

6.7.2 Precompiling Ada Programs

Because of differences between the Ada compiler and other host language compilers supported by the SQL precompiler, there are differences both in how the precompiler handles Ada source files and in how you precompile them. Refer to Table 6–3 for a summary of the files associated with the processing of Ada source files.

The differences between the Ada compiler and other compilers supported by the SQL precompiler follow:

You Must Create and Enable an Ada Library

Before you can precompile an Ada program that includes SQL statements, you must first create and enable an Ada library. To do this, use the DEC Ada program library manager ACS commands CREATE LIBRARY and SET LIBRARY.

See the Ada documentation set for more information on ACS.

The SQL Precompiler Generates a Second .ada File

When it processes an Ada program, the precompiler generates a second .ada file that is an Ada package specification for the SQL statement procedures. The file name for the package specification file is the name of the input file preceded by the acronym SQL and an underscore (_), and the file type is .ada. The Ada compiler uses the package specification, which is the package definition created by the precompiler.

You Must Copy an Intermediate File to the Ada Library

For languages other than Ada, the precompiler concatenates the object file it creates from processing SQL statements with the object file generated by the host language compiler, and then deletes the intermediate object files.

For Ada programs, however, the precompiler does not concatenate or delete the object files. The file name for the object file that the precompiler creates from processing embedded SQL statements is the name of the input file preceded by the acronym SQL and an underscore (SQL_). The file extension is .obj. However, the precompiler replaces hyphens (-) or dollar signs ($) with underscores (_) and reduces two or more underscores in a row to one underscore. You must copy that file to the Ada library with the ACS COPY FOREIGN command before you link the program, or explicitly name the file in the ACS LINK command.

Table 6–3 summarizes the files used by and created during the processing of Ada source modules.

Table 6–3 Files Related to Precompiling Ada Source Modules

File Name	Purpose
filename.sqlada	The source Ada language file that contains embedded SQL statements submitted to the SQL precompiler.
filename.ada	An intermediate Ada source file created by the precompiler and submitted to the Ada compiler.
filename.obj	The object file created by the Ada compiler from filename.ada. This file resides in the Ada library directory, not in the same directory as the .sqlada file.
sql_filename.mar[1]	An intermediate macro source file created by the SQL precompiler containing procedures that implement the SQL statements embedded in the .sqlada file. The precompiler submits the file to the macro processor and then deletes it unless the logical name SQL$KEEP_PREP_FILES is defined.
sql_filename.obj	The object file created by the macro processor from the .mar file. Unlike corresponding object files for other host languages, the precompiler does not concatenate this file with filename.obj and delete it. You must copy this file to the Ada library before linking, or else explicitly specify it in the ACS LINK command.

[1]On OpenVMS VAX only.

(continued on next page)

Table 6–3 (Cont.) Files Related to Precompiling Ada Source Modules

File Name	Purpose
sql_filename.ada	An Ada package specification created by the precompiler for the SQL statement procedures implemented in the .mar file.

Example 6–3 shows the steps in precompiling an Ada file called my_program.sqlada.

Example 6–3 Precompiling Ada Files

```
$ ! Create an Ada library:
$ ACS CREATE LIBRARY [.ADALIB]
$ ! Enable the library:
$ ACS SET LIBRARY [.ADALIB]
%ACS-I-CL_LIBIS, Current program library is DVD15:[PROGRAMS.ADALIB]
$ ! Invoke the SQL precompiler:
$ SQLPRE
INPUT FILE> my_program.sqlada/ADA
```

See Section 7.2.3 for an example of linking Ada programs.

6.7.3 Embedding SQL Statements in C Source Files

This section contains information that applies only to C programs. Refer also to Section 8.12.2 for information about declaring and using parameters specific to the C language.

Using Character Strings in C

When you use the SQL C precompiler, SQL translates all C character strings as null-terminated strings. This means that when SQL passes these character strings from the database to the program, it reserves space at the end of the string for the null character. When a program passes a character string to the database for input, SQL looks for the null character to determine how many characters to store in the database. SQL stores only those characters that precede the null character; it does not store the null character itself.

Because of the way SQL translates C character strings, you may encounter problems with applications that pass binary data to and from the database. To avoid these problems when you use the SQL C precompiler, use the $SQL_VARCHAR data type that SQL provides. Declaring the $SQL_VARCHAR data type lets you store and pass binary data.

Embedding SQL Statements in C Files That Contain Multiple Procedures

The SQL precompiler supports block structure in C programs. As a result, you can declare parameters to which SQL statements refer, such as SQLCODE, in multiple procedures in the same C source file, and the precompiler recognizes them as independent parameters.

Floating-Point Data in C

OpenVMS VAX OpenVMS Alpha

If your C program uses the D-floating format for floating-point data, you should compile with the NOG_FLOAT qualifier. This forces SQL to convert the double-precision data in the database from the G-floating format used by Oracle Rdb to the D-floating format used by C, and vice versa. ♦

OpenVMS VAX

On OpenVMS VAX, callable images must be linked against VAXCRTLG. ♦

Using C Pointers

The SQL precompiler for C does not support the use of pointers in passing parameters in SQL statements.

Declaring SQL Routines Using an Include File

When you call certain SQL routines in C programs, the programs can generate compilation informational messages. Oracle Rdb provides an include file, sql_rdb_headers.h, that eliminates the messages by explicitly providing prototypes for explicitly called SQL functions. For more information, see Section 10.3.4.

VAX C Extensions

Digital UNIX

On Digital UNIX, to compile programs that contain VAX C extensions, use the following option to the sqlpre command line:

```
-l cc="migrate -vaxc"
```
♦

6.7.4 Embedding SQL Statements in COBOL Source Files

This section provides information that applies only to COBOL programs.

Continuing Multiline Literals

For statements embedded in COBOL programs, SQL lets you continue multiline literals in two ways:

– To comply with the ANSI/ISO SQL standard, use a single quotation mark (') as the starting quotation mark character, put a hyphen (-) in the indicator area, and use a double quotation mark (") as the continuation character.

– If your program does not need to comply with the ANSI/ISO SQL standard, you can use the COBOL rule, which requires that the continuation character match the quotation mark character that started the literal.

Ensuring That COBOL Source Files Are in the Correct Format

The SQL precompiler lets you specify whether COBOL source files are in terminal format or ANSI/ISO format on the command line. The default is terminal format. If your source file is in ANSI/ISO format, specify the ANSI_FORMAT or –`form ansi` qualifier on the command line.

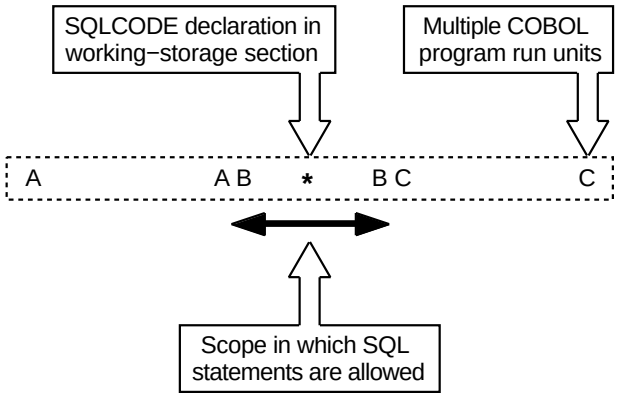
Embedding SQL Statements in COBOL Files That Contain Multiple Programs

If your precompiled COBOL source file contains multiple programs, all your SQL statements must execute within the scope of one of those programs.

Because the SQL precompiler does not support block structure in COBOL programs, you can specify only one SQLCODE declaration in a precompiled source file, even when the source file contains multiple programs. Either declare SQLCODE directly in one working-storage section of the source file or declare it indirectly by specifying the INCLUDE SQLCA statement in one working-storage section of the file. Whether you declare SQLCODE directly or indirectly, the location of the SQLCODE parameter determines the program that the SQL precompiler will process. At run time, SQL updates the SQLCODE parameter to reflect the execution status of SQL statements. Therefore, executable SQL statements must be in the same program where the SQLCODE parameter is declared. The precompiler assumes that all SQL statements are in the same program.

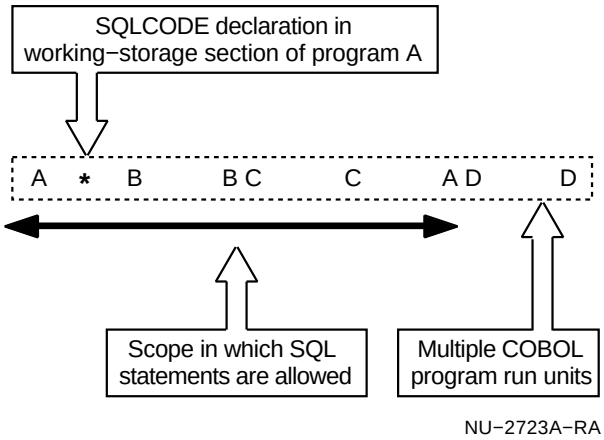
In Figure 6–2, the asterisk (*) marks the location of the SQLCODE declaration. Matched pairs of uppercase letters delimit each program in the file. The arrows note the section of the file where SQL statements can be specified. Precompiled SQL statements that occur after the END PROGRAM statement for program B generate host language compiler errors.

Figure 6–2 Scope of SQLCODE Declaration (COBOL)



Because contained programs are nested, SQL statements may span programs contained by the program in which the SQLCODE declaration occurs. For example, in Figure 6–3, program A contains programs B and C. The SQLCODE declaration located in the working-storage section of program A therefore allows SQL statements to be included in programs A, B, and C. SQL statements that occur in program D generate host language compiler errors because the statements occur after the END PROGRAM statement for program A.

Figure 6–3 Scope in Which SQL Statements Are Allowed (COBOL)



If your executable program image contains several COBOL programs that are not nested, and you want to execute SQL statements in more than one program, organize those programs into separate source files.

Declaring Structures

In COBOL, a structure defined as a word followed by a string is treated as a single variable. The type equates to VARCHAR(*n*) in SQL. The following example shows a structure that is treated as a single variable:

```
VARCHAR(32)   01 STRUCTURE.
                  49 STRUCT_LENGTH PIC S9(4) COMP.
                  49 STRUCT_STRING PIC X(32).
```

6.7.5 Embedding SQL Statements in FORTRAN Source Files

This section provides information that applies only to precompiled FORTRAN programs.

Limiting the Number of Characters Per Line

In precompiled FORTRAN programs, the SQL precompiler adheres to a restriction of 72 characters per line, unless you use the FORTRAN option EXTEND_SOURCE or –extend_source. When you do, the source code can contain up to 132 characters per line. If a statement is longer than the maximum, enter a continuation character in column 6 and continue the statement on the next line.

Digital UNIX

On Digital UNIX, you must use not only the –extend_source qualifier, but you must also specify it as a compiler option, as shown in the following example:

```
sqlpre -l fortran='-extend_source' -extend_source test.sfo
```
♦

Limiting the Number of Continuation Lines

OpenVMS VAX

The VAX FORTRAN compiler lets you specify a maximum number of continuation lines (up to 99) in a statement if you use the CONTINUATIONS qualifier. The default number of continuation lines is 19. The default for the DEC FORTRAN compiler is 99.

If a program uses a record definition, the SQL precompiler unpacks the record into individual elements and places each one on a separate line. If the number of elements in the record is greater than the maximum number of continuation lines, the FORTRAN compiler generates an error.

If this happens, increase the number of continuation lines by using the CONTINUATIONS qualifier to the FORTRAN command line. If the record contains more elements than the maximum allowed by FORTRAN (99 elements), you can edit the intermediate file (.for) to place more than one element on a line. ♦

Limiting the Length of FORTRAN Field Names Beginning with SQL

In precompiled FORTRAN programs, the SQL precompiler adheres to a 6-character restriction on the length of parameter names beginning with SQL that SQL declares as the result of an INCLUDE SQLCA statement. For example, the SQL precompiler automatically declares the SQLCODE parameter as SQLCOD rather than SQLCODE and SQLERR rather than SQLERRD.

This manual uses the complete name for fields in the SQLCA. However, use only the first 6 characters of the SQLCA names that begin with SQL when you specify them in FORTRAN statements. Note that parameter names for the RDB$MESSAGE_VECTOR and the fields it contains are *not* affected by this restriction and must be specified in their entirety.

If you substitute an explicit parameter declaration for the INCLUDE SQLCA statement, the parameter name must be SQLCOD in precompiled FORTRAN programs.

Embedding SQL Statements in FORTRAN IF Statements

Embed SQL statements only in block IF (IF . . . THEN . . . END_IF) and arithmetic IF statements in FORTRAN. Do not embed SQL statements in logical IF statements (IF <condition> <statement>). When SQL statements are embedded in logical IF statements, you may not be able to evaluate the execution status of the SQL statement.

Embedding SQL Statements in FORTRAN DO Loops

An SQL statement cannot be a labeled statement that ends a FORTRAN DO loop. The precompiler converts SQL WHENEVER statements to one or more host language IF statements that follow each executable SQL statement. IF statements cannot end a FORTRAN DO loop. (If you are not using SQL WHENEVER statements to monitor the execution status of SQL statements, you should be entering these final IF statements yourself to handle conditions and errors.)

If an SQL statement performs the last operation in a DO loop, end the loop either with an END DO statement or with a label and the CONTINUE statement.

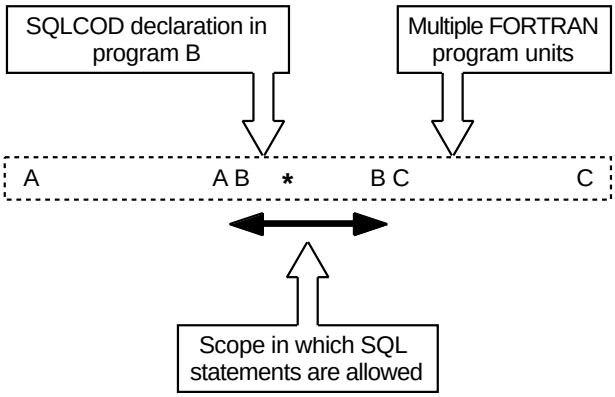
Embedding SQL Statements in FORTRAN Files That Contain Multiple END Statements

If your FORTRAN source file contains multiple program units (a main program and called subroutines), all your SQL statements must execute within the scope of one of those program units.

The SQL precompiler does not support block structure in FORTRAN programs. This means you can specify only one SQLCOD declaration in a source file. Either declare SQLCOD directly in one program unit of the source file, or declare it indirectly by specifying the INCLUDE SQLCA statement in one program unit of the source file. Whether you declare SQLCOD directly or indirectly, the location of the SQLCOD parameter determines the program unit that the SQL precompiler will process. At run time, SQL updates the SQLCOD parameter to reflect the execution status of SQL statements. Therefore, executable SQL statements must be in the program unit where the SQLCOD parameter is declared. The precompiler assumes that all SQL statements are in the same unit.

In Figure 6–4, the asterisk (*) marks the location of the SQLCOD declaration. Matched pairs of uppercase letters delimit each program unit in the file. The arrows note the section of the file where SQL statements can be specified. In this file, precompiled SQL statements that occur after the END statement for program unit B generates errors in the host language compiler.

Figure 6–4 Scope in Which SQL Statements Are Allowed (FORTRAN)



To include SQL statements in more than one unit of a program, organize the units into separate source files.

Declaring Structures

In FORTRAN, a structure defined as a word followed by a string is treated as a single variable. The type equates to VARCHAR(*n*) in SQL. The following example shows a structure that is treated as a single variable:

```
STRUCTURE /struct_name
        INTEGER*2           length
        CHARACTER*32        string
END STRUCTURE
```

6.7.6 Embedding SQL Statements in Pascal Source Files

This section contains information that applies only to Pascal programs.

Including Data from Dictionary

The precompiler does not support the use of the INCLUDE FROM DICTIONARY statement.

Embedding SQL Statements in Pascal Files That Contain Multiple Procedures

The SQL precompiler supports block structure in Pascal programs. As a result, you can declare parameters to which SQL statements refer, such as SQLCODE, in multiple procedures in the same Pascal source file, and the precompiler recognizes them as independent parameters.

Modifying Pascal Source Files to Meet SQL Restrictions

The SQL precompiler recognizes most Pascal rules and statements. However, you should be aware of the following restrictions:

- The SQL precompiler recognizes only the HIDDEN attribute for Pascal host variables.
- The SQL precompiler supports only one level of pointers.
- Because the SQL precompiler reads lines of code sequentially, the order of statements in your program is important. For example, a variable declaration must precede an assignment to that variable.

6.7.7 Embedding SQL Statements in PL/I Source Files

This section contains information that applies only to PL/I programs.

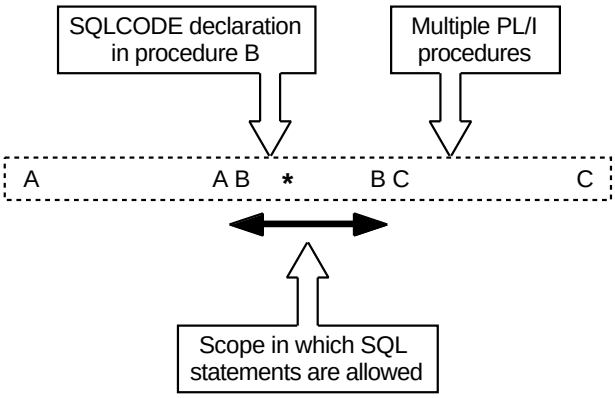
Embedding SQL Statements in PL/I Files That Contain Multiple Procedures

If your precompiled PL/I source file contains more than one procedure or function, your SQL statements must execute within the scope of the SQLCODE declaration.

The SQL precompiler does not support block structure in PL/I programs. As a result, you can specify only one SQLCODE declaration (or INCLUDE SQLCA statement) in a precompiled source file, even when the source file contains multiple procedures. At run time, SQL updates the SQLCODE parameter to reflect the execution status of SQL statements. Executable statements must therefore be in the procedure where the value of SQLCODE can be returned.

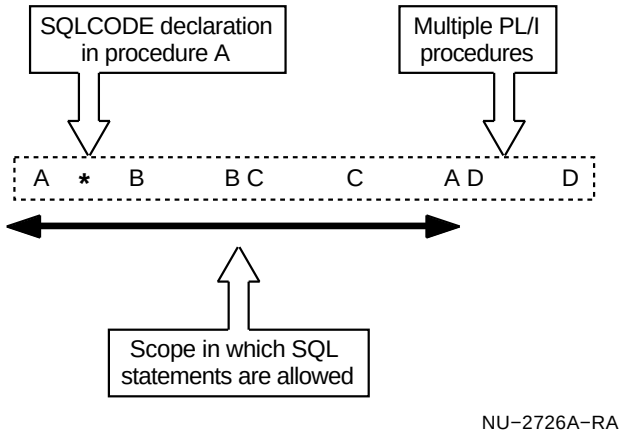
In Figure 6–5, the asterisk (*) marks the location of the SQLCODE declaration. Matched pairs of uppercase letters delimit procedures in the file. The arrows note the section of the file where SQL statements can be specified. Precompiled SQL statements that occur after the END statement for program unit B generate host language compiler errors.

Figure 6–5 Scope of SQLCODE Declaration (PL/I)



If procedures are nested, SQL statements may span procedures contained by the procedure in which the SQLCODE declaration occurs. For example, in Figure 6–6, procedure A contains procedures B and C. The SQLCODE declaration in procedure A therefore allows SQL statements in procedures A, B, and C. However, the SQL precompiler does not process SQL statements that occur in procedure D because these occur after the END statement for procedure A.

Figure 6–6 Scope in Which SQL Statements Are Allowed (PL/I)



7

Creating Images for Program Execution

After completing the compile phase of the application development process, you link one or more object modules into an executable image. Only after you create an executable image with the linker does the operating system let you run an application program. When the linker binds one or more object modules, it often must link with other necessary information, such as shareable images.

This chapter describes:

- The concepts of executable images, shareable images, and shared objects
- Linking object modules on OpenVMS
- Creating a shareable image file on OpenVMS to conserve system resources and improve program performance
- Installing a shareable image
- Linking object modules on Digital UNIX
- Inserting precompiled SQL modules in object libraries and archives
- Running the executable image created during the link phase
- Debugging the host language, SQL code, or both if the executable image produces run-time errors

Reference Reading

See the operating system and compiler documentation for additional information about linking object modules and creating shareable images or shared objects, running an executable image, and debugging programs.

7.1 Understanding Executable and Shareable Images

After you generate object modules for each source file, you link them together to create an executable image or a shareable image or shared object.

An **executable image** is an image that runs in a process. When run, an executable image is read from a file for execution in a process.

A **shareable image** is a collection of procedures that are called by executable images or other shareable images. A shareable image cannot be executed directly, but it can be linked with one or more other images to produce executable images.

If you recompile one module with a different version of Oracle Rdb, recompile all modules before you link them together.

7.2 Using the OpenVMS Linker

OpenVMS VAX OpenVMS Alpha

Ada Language Note

If you are creating an executable image from objects generated by the Ada compiler, you must follow procedures different from those described in this section. See Section 7.2.3 for details about linking Ada object modules.

To link object modules that include SQL statements, you specify:

- All the object files that your program image requires
- The SQL$USER library file

You can include the specifications directly in a command line or indirectly, in an options file. The command line to link a program without using an options file looks something like this:

```
$ LINK main_sql_mod, SQL$USER/LIBRARY
```

The /LIBRARY qualifier tells the linker that the file is a library file.

Make sure you link with the SQL$USER logical name, not with SYS$SHARE:SQL$USER.OLB.

Your system manager may have defined SQL$USER as a user default library by defining the LNK$LIBRARY logical name (or appropriate follow-on logical name such as LNK$LIBRARY_1) after installing Oracle Rdb. If SQL$USER is defined as a user default library, you do not need to specify the SQL library in your LINK command.

This example links one object module when SQL$USER is defined:

```
$ LINK my_program
```

This example links three object modules to form a single executable image, named my_main_program.exe:

```
$ LINK my_main_program, my_called_program, prog_messages
```

Reference Reading

The host language you use with SQL may require linking with language-specific support libraries or images. Refer to the section on linking programs in your host language documentation to see if your program must be linked with files that are specific to your host language.

See the OpenVMS Linker utility documentation for detailed information about image creation.

♦

7.2.1 Linking Programs Compiled with the Digital C Compiler on OpenVMS VAX Systems

OpenVMS VAX

When you link programs compiled with the Digital C compiler on an OpenVMS VAX system, you may need to set up some program section (PSECT) attributes. This restriction applies to both the SQL precompiler and SQL module processor. If you do not, you may receive a LINK-W-MULPSC, conflicting attributes error.

To avoid these errors, include the PSECT_ATTR on the link command as shown:

```
$ LINK/EXE=SQL$IVPC -
     SQL$IVPC, -
     SQL$USER/LIB, -
     SYS$INPUT/OPT
     PSECT_ATTR = RDB$MESSAGE_VECTOR,NOSHR,NOPIC
```

♦

7.2.2 Creating an Executable Image That Links with a Shareable Image

OpenVMS VAX OpenVMS Alpha

To identify as input a shareable image that is not in a library, you must use an options file. The /SHAREABLE positional qualifier, which is used to identify an input file as a shareable image file, can be used only in an options file; otherwise, the linker interprets it as a command qualifier rather than a positional qualifier. Example 7–1 shows a file named project3.opt containing both input file specifications and link options.

Example 7–1 Using an Options File to Link with a Shareable Image

```
MOD1,MOD7,LIB3/LIBRARY,-
LIB4/LIBRARY/INCLUDE=(MODX,MODY,MODZ),-
MOD12/SELECTIVE_SEARCH
STACK=75
SYMBOL=JOBCODE,5
```

```
$ LINK/MAP/CROSS_REFERENCE PROGA, PROGB, PROGC, project3/OPTIONS
```

If you want the LINK command to be in a command procedure, and you want to specify an options file in the LINK command, specify SYS$INPUT: as the options file. The DCL command interpreter interprets the lines following the LINK command as lines in the options file. For example, a command procedure linkprog.com might contain the following lines:

```
$ LINK, MAIN, SUB1, SUB2, SYS$INPUT:/OPTIONS
MYPROG/SHAREABLE
SYS$LIBRARY:APPLPCKGE/SHAREABLE
STACK=75
```

♦

7.2.3 Linking Ada Objects

OpenVMS VAX OpenVMS Alpha

The process for creating an executable image from objects generated by the Ada compiler is different from that for other languages. (Section 7.2 describes how to link object modules in host language programs other than Ada.) This section details those differences:

- With Ada, you use the ACS LINK command instead of the DCL LINK command.
- You do not specify the file name of the .obj file created by the Ada compiler as a parameter to the ACS LINK command. Instead, you specify the name of the main procedure in the Ada module as the first parameter to the ACS LINK command.

- If the object files are created by precompiling Ada source modules with embedded SQL statements, you must specify to the Ada program library manager that the SQL_filename.obj file (created at precompile time) is the package definition that corresponds to the package specification in the SQL_filename.ada file. There are two ways to do this:

  - Use the ACS COPY FOREIGN command to copy the object file into the Ada library:

    ```
    $ ACS SET LIBRARY [.ADALIB]
    %ACS-I-CL_LIBIS, Current program library is DVD15:[PROGRAMS.ADALIB]
    $ ACS COPY FOREIGN SQL_filename.obj
    ```

  - Explicitly specify the .obj file as the second parameter in the ACS LINK command:

    ```
    $ ACS LINK main_procedure_name SQL_filename.obj
    ```

If the logical name LNK$LIBRARY (or the appropriate follow-on logical name) is not defined for SQL$USER, and if the object files are created by precompiling Ada source modules with embedded SQL statements, then you must specify the SQL library in your LINK command as follows:

```
$ ACS LINK main_procedure_name SQL_filename.obj, SQL$USER
```

The following example links a precompiled Ada program sql_my_program.obj that has a main procedure called MY_PROGRAM. The example does not specify the ACS COPY command; instead, it specifies the .obj file created at precompile time along with the main procedure name:

```
$ ! This LINK command requires that the logical name
$ ! LNK$LIBRARY is defined as SQL$USER.OLB
$ !
$ ! The name MY_PROGRAM, immediately following the word LINK,
$ ! is a procedure name, not a file name.
$ !
$ ACS LINK MY_PROGRAM         sql_my_program.obj
```

The ACS LINK command invokes the OpenVMS Linker utility. Other than the differences outlined in this section, the information in other sections of this chapter applies to linking Ada objects as well. ♦

7.3 Creating Shareable Images

OpenVMS VAX OpenVMS Alpha

The OpenVMS Linker utility can create shareable images. You might consider making part of your application a shareable image for the following reasons:

- You save disk space because many executable images can be linked with a single disk-resident copy of the shareable image.
- Shareable images simplify the implementation of applications where response times are so critical that control variables and data readings must remain in main memory.
- By carefully organizing a shareable image and by using universal symbols (in a transfer vector on OpenVMS VAX or a symbol vector on OpenVMS Alpha) and position-independent coding techniques, you can make significant changes and enhancements to the shareable image without relinking all the images bound to it.

A shareable image is not complete by itself and is therefore not directly executable. To execute a shareable image, you must include it as input in a linking operation that produces an executable image.

To produce a shareable image, specify the /SHAREABLE qualifier on the command line. Like executable images, shareable images must be linked with the SQL$USER library file.

Building a shareable image requires that you specify additional information about how the linker is to treat the program sections (PSECTs). A **program section** represents an area of memory. It has a name, a length, and a set of attributes describing the intended and permitted uses of that part of memory. For instance, certain declarations may be set to disallow writing to that part of memory.

When the linker builds an image from the object file, it combines program sections that have similar attributes to form image sections. **Image sections** specify the size and attributes of a part of system memory. The image activator uses that information to determine the characteristics of physical memory pages—for instance, the nowrite program section mentioned in the previous paragraph would be part of a nowrite image section, and that image section would be mapped to a part of physical memory that did not allow writing.

Section 7.3.1 describes how to create a shareable image when the shareable image and the executable images that link with it do not share database attaches and transactions, or when the shareable image contains SQL modules but the executable image does not. Neither image contains modules compiled with the CONNECT qualifier.

Section 7.3.2 describes how to create a shareable image when the shareable image and the executable images that link with it share database attaches and transactions, or the images use connections (that is, the SQL modules were compiled using the CONNECT qualifier).

Do not attempt to create images where both the executable image and the shareable image contain SQL modules and use connections, but do not share database and transaction handles. This combination is not supported and may not work as expected. If the images share database and transaction handles, there is no problem. ♦

7.3.1 Creating Executable and Shareable Images Not Sharing Database Attaches

OpenVMS VAX OpenVMS Alpha

This section describes how to create images that do not share database attaches. This is the simplest way to create shareable images that contain SQL modules. All that is necessary to create this type of image is to ensure that the program section definitions in the shareable image are properly defined. Example 7–2 shows the proper definitions.

Example 7–2 Linking to Create a Shareable Image

```
$ LINK/SHAREABLE=MYSHARE.EXE MYVECTORS, MYSHARE, SQL$USER/LIB, SYS$INPUT/OPTIONS
PSECT_ATTR=RDB$DBHANDLE,           NOSHR,LCL
PSECT_ATTR=MYALIAS,                NOSHR,LCL
PSECT_ATTR=RDB$MESSAGE_VECTOR,     NOSHR,LCL
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR,LCL
PSECT_ATTR=SQLCA,                  NOSHR,LCL
PSECT_ATTR=SQLDA,                  NOSHR,LCL
PSECT_ATTR=SQL$CALLER_PC,          NOSHR,LCL
$
```

Every alias declared in your programs must be set to NOSHR,LCL. NOSHR means these data structures are not shared; that is, your program uses a different part of memory than the shareable image does. If both the shareable image and the executable image attach to a database and start a transaction, you have two database attachments and two separate (and potentially competing) transactions.

If one transaction encounters a lock conflict with the other, you receive a deadlock error. ♦

7.3.2 Creating Executable and Shareable Images Sharing Database Attaches

OpenVMS VAX OpenVMS Alpha

When images must share database attachments, the executable image needs to write to SQL data structures that reside within the shareable image. For this to happen correctly, the data structures to be written must reside at a fixed offset within that image and they must not move if the image is later relinked. Positioning the SQL data structures within the shareable image is similar in concept to placing the transfer vector for the shareable image. (See the OpenVMS Linker documentation for more information on transfer vectors.)

This section describes how to direct the linker to properly place the affected SQL program sections (PSECT) within a shareable image and how to initialize aliases so that modules can share the aliases.

When you share aliases across multiple images, you must have one and *only one* definition of each shared alias. To define a shared alias, use the GLOBAL keyword in the DECLARE ALIAS statement, as shown in the following example:

```
DECLARE MYALIAS GLOBAL ALIAS  FOR FILENAME mf_personnel
```

In other modules, you can have many references to the defined alias, but you must declare the alias using the EXTERNAL keyword.

Compile the module containing the defining DECLARE ALIAS statement with the NOEXTERNAL_GLOBAL or SQLOPTIONS=NOEXTERNAL_GLOBAL command line qualifier. These qualifiers specify that those aliases defined as GLOBAL be treated as GLOBAL and those defined as EXTERNAL be treated as external.

The image into which you link this module cannot be linked against any other image that defines this alias.

For example, suppose you want to create a shareable image named myshare.exe. The myshare.sc file is an SQL precompiled program that contains SQL routines and the DECLARE ALIAS statement that defines MYALIAS using the GLOBAL keyword. Compile the SQL module using the NOEXTERNAL_GLOBAL qualifier. The resulting object module is named myshare.obj. ♦

OpenVMS VAX

On OpenVMS VAX, you must create an object module containing the transfer vector, such as myvectors.obj. Then, create the shareable image by using the command in Example 7–3.

Example 7–3 Linking Shareable Images That Share Handles on OpenVMS VAX

```
$ LINK/SHAREABLE=myshare.exe MYSHARE, SQL$USER/LIB, SYS$INPUT/OPTIONS
CLUSTER=TRANSFER_VECTOR,,,MYVECTORS
CLUSTER=SQL_PSECTS
COLLECT=SQL_PSECTS, RDB$DBHANDLE, MYALIAS, RDB$MESSAGE_VECTOR,-
    RDB$TRANSACTION_HANDLE, SQL$CALLER_PC, SQL$TRANSACTION_PTR,-
    SQLCA, SQLDA
PSECT_ATTR=RDB$DBHANDLE,           NOSHR,GBL
PSECT_ATTR=MYALIAS,                NOSHR,GBL
PSECT_ATTR=RDB$MESSAGE_VECTOR,     NOSHR,GBL
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR,GBL
PSECT_ATTR=SQL$CALLER_PC,          NOSHR,GBL
PSECT_ATTR=SQL$TRANSACTION_PTR,    NOSHR,GBL
PSECT_ATTR=SQLCA,                  NOSHR,GBL
PSECT_ATTR=SQLDA,                  NOSHR,GBL
$                                                    ♦
```

OpenVMS Alpha

On OpenVMS Alpha, create the shareable image by using the command in Example 7–4. This command includes the symbol vector.

Example 7–4 Linking Shareable Images That Share Handles on OpenVMS Alpha

```
$ LINK/SHAREABLE=myshare.exe MYSHARE, SQL$USER/LIB, SYS$INPUT/OPTIONS
CLUSTER=SQL_PSECTS
COLLECT=SQL_PSECTS, RDB$DBHANDLE, MYALIAS, RDB$MESSAGE_VECTOR,-
    RDB$TRANSACTION_HANDLE, SQL$CALLER_PC, SQL$TRANSACTION_PTR,-
    SQLCA, SQLDA
PSECT_ATTR=RDB$DBHANDLE,           NOSHR,GBL
PSECT_ATTR=MYALIAS,                NOSHR,GBL
PSECT_ATTR=RDB$MESSAGE_VECTOR,     NOSHR,GBL
PSECT_ATTR=RDB$TRANSACTION_HANDLE, NOSHR,GBL
PSECT_ATTR=SQL$CALLER_PC,          NOSHR,GBL
PSECT_ATTR=SQL$TRANSACTION_PTR.    NOSHR,GBL
PSECT_ATTR=SQLCA,                  NOSHR,GBL
PSECT_ATTR=SQLDA,                  NOSHR,GBL
```

(continued on next page)

Example 7–4 (Cont.) Linking Shareable Images That Share Handles on OpenVMS Alpha

```
SYMBOL_VECTOR   =       (               -
   RDB$DBHANDLE=PSECT,                  -
   MYALIAS=PSECT,                       -
   RDB$MESSAGE_VECTOR=PSECT,            -
   RDB$TRANSACTION_HANDLE=PSECT,        -
   SQLCA=PSECT,                         -
   SQLDA=PSECT,                         -
   MYSHAREDPROC=PROCEDURE               -
                        )                                   ♦
```

OpenVMS VAX OpenVMS Alpha

If you relink the shareable image and the data structures move, then you must relink any executable and shareable images linked against the shareable image that changed. Be careful that the lengths of the program sections collected in the SQL_PSECTS cluster do not change. For example, if your program contains an SQLDA but does not use an SQLCA, and you later add one, the offset for the SQLDA program section changes. Similarly, if you later add another database alias to your shareable image, you need to relink the executable images.

Similarly, if the length of your transfer vector's or symbol vector's program section changes, then you may need to relink your executable image. It is advisable to allocate space in your transfer vector or symbol vector for future use so that you do not need to relink all executable images if you add entries to the transfer vector or symbol vector.

If you use database aliases, the alias names must be collected in the SQL_PSECT, and the program section attributes must be set to NOSHR GBL. ♦

OpenVMS VAX

For example, if you declare a database with the alias ANOTHER_DB, you need to add ANOTHER_DB to the COLLECT=SQL_PSECTS command, and you need to add the following line to the linker options:

```
PSECT_ATTR=ANOTHER_DB,          NOSHR,GBL                   ♦
```

OpenVMS Alpha

On OpenVMS Alpha, you must also add the following line to the symbol vector:

```
ANOTHER_DB=PSECT                                                    ♦
```

After you create the shareable image, create the executable images that use the shareable image. For the executable image to work properly, you must:

- Declare the aliases as EXTERNAL if they refer to the shared alias.

  Remember that the shared alias is the alias declared as GLOBAL in the shareable image. As mentioned previously, you can have only one definition of each shared alias. However, you can have any number of references to the shared alias, but those references must be declared as EXTERNAL.

- Compile the SQL modules using the NOEXTERNAL_GLOBAL option.
- Define the program section attributes to match the program section attributes in the shareable image.

For example, you compile an embedded SQL and COBOL program as shown in the following example:

```
$ SQLPRE
INPUT_FILE> mymain.sco /COB /SQLOPTIONS=(NOEXTERNAL_GLOBAL)
```

Then, you link mymain using the commands shown in Example 7–5.

Example 7–5 Linking an Executable Image That Uses a Shareable Image

```
$ LINK mymain, SQL$USER/LIB, SYS$INPUT/OPTIONS
PSECT_ATTR=RDB$DBHANDLE,            NOSHR,GBL
PSECT_ATTR=MYALIAS,                 NOSHR,GBL
PSECT_ATTR=RDB$MESSAGE_VECTOR,      NOSHR,GBL
PSECT_ATTR=RDB$TRANSACTION_HANDLE,  NOSHR,GBL
PSECT_ATTR=SQLCA,                   NOSHR,GBL
PSECT_ATTR=SQLDA,                   NOSHR,GBL
MYSHARE/SHARE
$                                                                   ♦
```

7.4 Installing Shareable Images

OpenVMS VAX OpenVMS Alpha

In order for shareable images to be used by multiple processes, they often must be installed to make them known to the system. To have all processes share the same copy of a shareable image in memory, thus conserving main physical memory, install the image using the INSTALL command of the OpenVMS Install utility, and modify the file specification of the shareable image with the /SHARE qualifier.

For information about installing shareable images, see the OpenVMS Install utility documentation. For information about the shared images that Oracle Rdb installs and those that your system manager must install manually, refer to the *Oracle Rdb7 Installation and Configuration Guide.* ♦

7.5 Linking Modules on Digital UNIX

Digital UNIX

On Digital UNIX, you link the object files by processing them with the host language compiler. To link object modules that include SQL statements, specify:

- All the object files that your program requires, such as the SQL module language object file and host language object file, or the SQL precompiler object file
- The SQL libraries, which are located in the SQL directory tree

To simplify linking, define a global symbol, SQLLIBS, to translate to the SQL libraries. Define the symbol to equate to the following:

```
-lrdbsql -lrdbshr -lcosi -lots
```

For example in the Bourne shell, enter the following commands:

```
$ SQLLIBS='-lrdbsql -lrdbshr -lcosi -lots'
$ export SQLLIBS
```

When you link an SQL module language program, you must first process the SQL module with the SQL module processor. Then, in one step, you can compile the host language module and link object files of the host language module and the SQL module with the SQL libraries.

The following example shows how to link the SQL module object file test_mod.o with the C language program test_h.c:

```
$ cc -o test_h test_h.c test_mod.o ${SQLLIBS}
```

For more information on compiling SQL module language programs, see Section 5.2.

When you link a precompiled SQL program, you must first process the program with the SQL precompiler. Then, you use the host language compiler to link the object file with the SQL libraries.

The following example shows how to link the object file, test.o, from a precompiled SQL program written in the C language:

```
$ cc -o test test.o ${SQLLIBS}
```

For more information on compiling SQL precompiled programs, see Section 6.3.

Oracle Rdb provides a file, /usr/lib/sqllibs.make, that you can include in your makefile to define SQLLIBS. The sqllibs.make file contains the four required shared object libraries, but you may find it necessary to add libraries to resolve the additional symbols your application is using. ♦

7.5.1 Building Applications with Multiple Modules

Digital UNIX

There are a number of internal variables, such as aliases, that SQL generates in the object files it creates. These variables must be properly initialized to ensure proper statement execution. By default, SQL generates code to initialize these variables.

When linking two or more SQL object files in an application, the Digital UNIX linker requires that only one module initialize the variable and that all other modules generate references to that variable.

If more than one module initializes these variables, the Digital UNIX linker generates an error indicating that the symbols are multiply defined. If no module initializes these variables, the SQL statement behaves erratically, generates Oracle Rdb errors about bad handles, or both.

You must choose one module for each of these objects to act as the definer for these variables. For example, you must have one and *only* one definition of each shared alias. To define a shared alias, use the GLOBAL keyword in the DECLARE ALIAS statement, as shown in the following example:

```
DECLARE MYALIAS GLOBAL ALIAS FOR FILENAME mf_personnel
```

In other modules, you can have many references to the defined alias, but you must declare the alias using the EXTERNAL keyword.

Compile all modules with the –noextern or –s'–noextern' option (the default).

This option specifies that those aliases defined as GLOBAL be treated as GLOBAL and those defined as EXTERNAL be treated as external. ♦

7.6 Inserting Precompiled SQL Modules in Object Libraries and Archives

You can insert SQL precompiler object files into OpenVMS object libraries or Digital UNIX archives. Note that SQL precompiler object files contain two distinct modules. One module is the host language program and the other is the generated SQL module. The generated SQL module is named RDB$<module> where <module> is either the name declared in the SQL DECLARE MODULE statement or the name of the host module (the default).

7.7 Running a Program

OpenVMS VAX OpenVMS Alpha

To execute a program on OpenVMS, enter the RUN command and the program name at the DCL prompt:

```
$ RUN MYPROGRAM
```

♦

Digital UNIX

To execute a program on Digital UNIX, enter the program name at the command line prompt:

```
$ myprogram
```

♦

When you run the program to test input and output, you may encounter error conditions that your program does not handle correctly or at all. For information about handling errors in programs, see Chapter 10.

Reference Reading

For information about error messages, refer to the *Oracle Rdb7 SQL Reference Manual*. The error message appendix in the reference manual supplies the online location of files with explanations and user actions for messages from relational database facilities.

7.8 Debugging SQL Statements and Program Code

To debug a program, you often need to look at your SQL statements and host language code, as follows:

- SQL code debugging

  Use interactive SQL to find and eliminate syntax errors in SQL statements that are in an SQL module file or in an embedded program.

- Host language code debugging

  If you cannot readily solve a problem, debugger utilities provide ways for you to monitor the execution of your program at run time.

With many debugger utilities, you can:

- Step through the program one statement at a time
- Examine and modify statements and data values
- Stop program execution at specified points
- Display messages at specified points

OpenVMS VAX OpenVMS Alpha

To use the OpenVMS Debugger, include the /DEBUG qualifier when you compile and link program files.

Use the /NOOPTIMIZE compile qualifier with the /DEBUG qualifier for FORTRAN, Pascal, and Ada programs. By default, FORTRAN, Pascal, and Ada include optimizations in programs. Optimized code can make program errors difficult to find.

Debugging global routines in a shareable image requires commands (SET IMAGE, for example) that you do not need to use when debugging similar routines in an executable image. If you are debugging routines in a shareable image, be sure to read about these commands in the OpenVMS Debugger utility documentation. ♦

Note

When you include SQL statements in precompiled programs, you should enter all changes, including changes to host language statements, in the source file processed by the SQL precompiler, not the host language file.

8
Declaring and Using Parameters

This chapter discusses the topic of ensuring data type compatibility when transferring column values to and from a program. For the most part, the discussion applies to your program regardless of the SQL processor you are using at compile time.

This chapter describes the following:

- The differences between how the module processor and the SQL precompiler handle parameters
- The terminology used to describe parameters
- The function of parameters and options in declaring them
- Declaring the data types of parameters
- Copying parameter declarations from an outside source
- Declaring and using main parameters
- Declaring and using indicator parameters
- Avoiding common mistakes when declaring and using parameters
- Declaring and using parameters with specific host languages

For information on declaring parameters for stored routines, see Chapter 13.

8.1 Overview of Declaring and Using Parameters

The approach you take in declaring and using parameters differs between the SQL precompiler and the SQL module processor:

- Specifying host language variables with the SQL precompiler

  If you use the SQL precompiler, you embed SQL statements in a host language source file and include direct references to host language variables in the SQL statements. At compile time, the SQL precompiler replaces the SQL statements in the host language source file with calls to procedures implemented by the processor. The host language variables

originally specified in SQL statements by the programmer are specified as parameters in host language calls.

- Specifying module parameters with the SQL module processor

  When you are working with the SQL module processor you create the source file that contains the procedures to execute SQL operations. These procedures contain both SQL statements and the declarations of parameters to which the statements refer. Then, in the host language source file, you declare an additional set of parameters and specify them in calls to the external procedures.

  Figure 4–1 illustrates the correspondence between actual parameters declared using host language syntax and procedure parameters declared using SQL syntax.

Whether you use the SQL precompiler or the SQL module processor, you can use parameters in value expressions in data manipulation statements, but you cannot use them in data definition statements.

Your options for declaring parameters in your host language source file:

- Depend on the function of the parameter
- Are affected by the method you use to specify and process SQL statements in programs
- Are limited by the types of parameter declarations that SQL supports in your host language

Reference Reading

For detailed information about parameter declarations, refer to the chapter on language and syntax elements in the *Oracle Rdb7 SQL Reference Manual*. That chapter has sections on the following topics: main parameters and indicator parameters, structures and indicator arrays, data types, supported host language parameter declarations, and data type conversions.

In addition, the chapter on language and syntax elements contains rules for entering SQL keywords and user-defined names. Information related to the interpretation of letter case, hyphens (-), and underscores (_) is particularly important in the programming environment.

8.2 Understanding Terminology

This chapter contains terminology that you must understand to declare and use parameters effectively. These terms include the following:

- Variable and parameter

  Because of the need to address both SQL precompiler users and SQL module processor users, the term *parameter*, rather than the term variable, is used most often to describe a program declaration to which an SQL statement refers. Parameter is a term appropriate in the context of calls and procedures.

  If you are using the SQL module processor, parameter refers to the formal parameters that you specify in SQL module language procedures.

  If you are using the SQL precompiler, parameter refers to the declared host language variables that you include in embedded SQL statements. If you use the SQL precompiler, and therefore do not directly implement SQL operations as procedures, you may prefer to substitute the term field or variable for the term parameter.

  The term *variable* refers to local variables, such as those found in compound statements.

- Structure and record

  The term **structure** refers to a parameter that contains other fields. (Unlike the case of repeating items that make up one dimension in an array, the basic elements in a structure do not need to be identically defined.) In some host languages, the term record and the term structure refer to the same declaration. In other languages, such as FORTRAN, you declare structures and records separately. If the language you are using declares records separately from structures, you should specify record names in precompiled SQL statements. In this case, the fields listed or implied by the record declaration constitute the structure to SQL.

8.3 Understanding Parameter Function and Declaration Options

You have different options for declaring parameters depending upon the function of the parameters. The following list describes your options for declaring parameters used in SQL statements. Unless specifically noted otherwise, you can use the described options with either the SQL precompiler or SQL module processor.

- **Main parameters**, which are parameters that contain values for selecting, manipulating, or storing columns, can be declared in the following ways:
  - Directly, by using host language statements in programs
  - Indirectly, by using the SQL INCLUDE statement to copy parameter definitions from the repository or from a text file into your host language source file (SQL precompiler only)
  - Indirectly, by using the FROM path-name clause in an SQL module procedure to copy record definitions from the repository
  - Indirectly, by using a host language INCLUDE or COPY statement to copy parameter definitions from another source into your host language source file
- Individually declared **indicator parameters** and **indicator arrays**, both of which are parameters to handle null values, can be declared in the following ways:
  - Directly, by entering declarations in your host language source file
  - Indirectly, by using the SQL INCLUDE statement to copy a set of declarations for indicator parameters from the repository or a text file into your host language source file (SQL precompiler only)
  - Indirectly, by using a host language COPY or INCLUDE statement to copy a set of declarations for indicator parameters from another source into your host language source file
- Parameters for storing the execution status of SQL statements can be declared in the following ways:
  - Directly, by entering declarations in your host language source file to define the SQLSTATE parameter

    See Chapter 10 and the appendix on SQLSTATE in the *Oracle Rdb7 SQL Reference Manual* for information on the SQLSTATE parameter.
  - Directly, by entering declarations in your host language source file to define the SQLCODE parameter (SQLCOD in FORTRAN programs) and perhaps other fields that you want to use
  - Indirectly, by using the INCLUDE SQLCA statement in a section of your host language source file where parameter declarations are valid (SQL precompiler only)

    This statement includes a set of parameter declarations. Among the defined parameters are SQLCODE and other parameters useful in error-handling procedures.

- Indirectly, by using a host language COPY or INCLUDE statement to copy declarations from another source into your host language source file (SQL module processor only)

The appendix on the SQL Communications Area (SQLCA) in the *Oracle Rdb7 SQL Reference Manual* includes language-specific declarations of SQLCODE and other fields that make up the SQLCA. This appendix shows the declarations SQL automatically makes if you use the INCLUDE SQLCA statement. If you use host language statements to define parameters needed for error handling, the declarations provide a useful reference.

For more information about declaring and using parameters associated with error handling, see Section 8.6 and Chapter 10.

- Parameters that describe the number and data types of input and output values in dynamically executed SQL statements can be declared in the following ways:
  - Directly, by entering declarations in your host language source file
  - Indirectly, by using the INCLUDE SQLDA or INCLUDE SQLDA2 statement in a section of your host language source file where parameter declarations are valid (SQL precompiler only)
  - Indirectly, by using host language COPY or INCLUDE statements to copy declarations from another source into your host language source file (SQL module processor only)
  - Indirectly, by using the SQL_SQLDA2.H header file in C programs. For more information on using this header file, see Section 11.3.2.

Among these parameters are fields representing keywords, and table and column names in SQL statements that are unknown until run time.

Reference Reading

Chapter 11 describes when to declare parameters to process dynamic SQL statements, and provides examples. In addition, the *Oracle Rdb7 SQL Reference Manual* contains an appendix on the SQL Descriptor Areas (SQLDA and SQLDA2) that describes parameters you may need to supply if you cannot use the INCLUDE SQLDA or INCLUDE SQLDA2 statement.

Read Section 8.6 for more details about using the SQL INCLUDE statement or equivalent host language statements.

Your options for declaring parameters used in SQL statements are limited to the types of declarations that SQL supports in your host language. For example, in host language programs you can specify a structure (record) or array element only in the places where you specify a parameter name in SQL statements. In addition, the SQL precompiler does not support all constructs that are specific to a host language (implicit declarations in FORTRAN, for example).

When you declare parameters, make sure that you use data types that are compatible with SQL data types.

8.4 Declaring the Data Types of Parameters

Whether you use the SQL precompiler or the SQL module processor to process SQL statements, a parameter declaration in SQL statements is *invalid* when both the following conditions are true:

- The data type in the parameter declaration is not equivalent to an SQL data type.
- SQL cannot convert the parameter's value:
  - To the data type defined for the column (for transfer of data to or from a column)
  - To the data type for the value expression to which the host parameter or associated parameter is compared (for evaluation of a condition)

Data type support issues are not limited to whether the language or the database system supports general types of storage formats (packed decimal, date, floating-point, text, or fixed-point binary, for example). For numeric data, there may be issues related to support for signed or unsigned data, scale computations, size, and interpretation of double-precision floating-point values (D-Floating or G-Floating).

In some cases, you may need to define intermediate host language variables that are related to an input or output operation.

You may need additional variables, for example, to contain values input from the terminal or data file when those values require intermediate processing by the program before they can be used in UPDATE or INSERT statements. Conversely, you may need additional variables when values returned by a FETCH or SELECT statement require processing by the host language before output to a terminal or data file. The kinds of values most likely to require additional variables for temporary program storage are: dates, times, timestamps, intervals (assuming the source or target column is defined using

the date-time data types), and text strings (usually to convert strings to all uppercase or mixed case, as appropriate).

One of the main advantages of precompiler support for your host language is that the precompiler generates errors or warnings to alert you to declarations that SQL considers invalid. Therefore, the precompiler prevents many of the run-time errors and unexpected results that unsupported or inappropriate declarations can cause when data is transferred between a program and the database. However, you cannot detect at precompile time all the run-time results that you may consider a problem when you transfer data between your program and a database.

Oracle Rdb considers truncation of character values and rounding of numeric values to be appropriate operations in certain situations. Oracle Rdb also considers some conversions among unlike data types to be valid because these operations are sometimes appropriate and useful to the programmer. Therefore, it is possible for you to declare a parameter that is the wrong data type or the incorrect size for what you want to do, but not discover the problem until run time.

In addition, the SQL precompiler does not check declarations of parameters used only for internal program processing (parameters not specified in SQL statements). These parameters are checked only by the host language compiler after it receives processed source code from the SQL precompiler.

Note

The preceding paragraph does not apply to all parameters in precompiled programs. If you declare any parameters as double-precision floating-point (even if those parameters are not used in precompiled SQL statements), read the information about the SQL precompiler qualifiers for G-Floating and D-Floating numbers in the *Oracle Rdb7 SQL Reference Manual*.

The SQL module processor does not check declarations of parameters in the host language module at all. Therefore, when you create a host language module that calls procedures in an SQL language module, it is your responsibility to ensure that parameters in the host language file are correctly declared when you intend to use them in a call to an SQL procedure. Specifically, it is your responsibility to make sure that the storage format and specification order of the actual parameter in your host language source file exactly match the storage format and specification order of the corresponding procedure parameter in an SQL source file. Also, you must be careful to match

the passing mechanism (by reference or by descriptor) of corresponding values in the called and calling modules.

Note

In SQL modules, when the storage format of a host language parameter cannot match the storage format of a column, the storage format of an associated procedure parameter should match that of the host language parameter rather than the column.

There is one exception to the previous rule. When you want to transfer DATE VMS values in *binary* format between a host language program and the database, always define the associated procedure parameter as a DATE data type even though most host languages do not support this data type. For binary interpretation of DATE values, SQL expects an 8-byte host language parameter (into which it stores a quadword value) and is not particular about the data type or passing mechanism you define for that parameter in the host language program. On the other hand, the conversion of DATE values from or to character-string format *does* follow the general rule about matching procedure parameters with their counterpart host language parameters. For character-string interpretation of DATE values, both host language parameters and procedure parameters are defined as 8- to 16-byte character fields.

See Chapter 9 for more information about date-time data types.

SQL module language provides the run-time CHECK option to help you determine if procedure and actual parameters correctly match with respect to the passing mechanism. See Section 4.2.4 for more information about the correct correspondence between actual parameters in a host language program and procedure parameters in SQL.

See Section 4.2.2 through Section 4.2.9 for more information about declaring parameters in SQL module procedures.

Language-specific sections presented later in this chapter provide more information about parameter use. Note, however, that the *Oracle Rdb7 SQL Reference Manual* is the only place where you will find a complete list of the data conversion rules applied by Oracle Rdb.

8.5 Copying Parameter Declarations from a Source Outside Your Program

Depending on whether SQL statements are processed by the SQL precompiler or the SQL module processor, you can use the SQL INCLUDE statement, a comparable host language statement, or the FROM path-name clause in a procedure declaration. These statements and clauses allow you to develop and use standard sets of declarations that programmers at your site can copy into applications. Maintaining one set of standard declarations at your site reduces:

- The number of programmer errors due to incorrectly typing declarations in source files
- Program development time
- The amount of editing that source files may require if table, column, or view definitions change after a program is in use

8.6 Using the SQL INCLUDE Statement

You can use the SQL INCLUDE statement only when you embed SQL statements in a host language source file. The SQL precompiler must access declarations of all parameters you specify in SQL statements and does not process host language COPY or INCLUDE statements.

However, you cannot use the SQL INCLUDE statement in SQL modules. Declarations for host language program parameters must be copied into host language source files. Unlike the SQL precompiler, the SQL module processor *does not* process your host language source files. Your host language compiler, which *does* process these files, recognizes only the host language COPY or INCLUDE statement.

The SQL INCLUDE statement has the following variations:

OpenVMS VAX OpenVMS Alpha

- The SQL INCLUDE FROM DICTIONARY statement copies the definition for a table, view, or record from a repository node. ♦
- The INCLUDE file-spec statement copies code from text files that contain language source code to be shared by programs.
- The INCLUDE SQLCA statement declares parameters for error handling.
- The INCLUDE SQLDA or INCLUDE SQLDA2 statement declares parameters used in dynamic SQL statements.

OpenVMS VAX OpenVMS Alpha The SQL INCLUDE statement does not support files created by the OpenVMS Librarian utility. Therefore, you cannot copy from library files into your program when these files contain parameter declarations used in embedded SQL statements. ♦

Reference Reading

The chapter on SQL statements in the *Oracle Rdb7 SQL Reference Manual* contains a section on the INCLUDE statement. The section provides additional discussion of statement variations and a complete program example.

8.6.1 Using the SQL INCLUDE FROM DICTIONARY Statement

OpenVMS VAX OpenVMS Alpha Use the INCLUDE FROM DICTIONARY statement in SQL precompiled programs to copy the definition of:

- A table or view from the repository node created and maintained for a database by the CREATE DATABASE or INTEGRATE statements
- A record from a repository node other than the one created or maintained for the database

If the host language you are using with SQL supports all data types defined for columns in a table, a quick way to define program parameters that will match columns in the tables or views you use is to include definitions from the repository node that stores a database.

For example, to include column definitions for the DEPARTMENTS table from the repository node that was created in directory WARREN for the personnel database, use the following statement, making sure to enclose the repository path name in single quotation marks ('):

```
EXEC SQL INCLUDE FROM DICTIONARY
     'DISK3:[DEPT32.CDD]WARREN.PERSONNEL.RDB$RELATIONS.DEPARTMENTS'
```

The preceding example illustrates that you specify RDB$RELATIONS as part of the path name for a table when the repository node from which you are copying definitions was created by the CREATE DATABASE or INTEGRATE statements.

For tables that contain data types not supported by a host language, you may create and maintain repository record definitions with data types that are compatible with the ones in the database. You can use the INCLUDE FROM DICTIONARY statement to copy these as well. To include a repository record definition that is not generated and stored by database operations, use the

following statement, making sure to enclose the path name in single quotation marks ('):

```
EXEC SQL INCLUDE FROM DICTIONARY
        'DISK3:[DEPT32.CDD]WARREN.PERSONNEL.DEPARTMENTS'
```

When you use the SQL precompiler for the C language, SQL translates character string fields in the record as null-terminated strings. See Section 8.12.2 for more information about how SQL handles character strings in the C language. ♦

8.6.2 Using the INCLUDE SQLCA Statement

The INCLUDE SQLCA statement provides declarations that are coded into the SQL precompiler. This statement declares parameters for error handling. In addition to declarations of SQLCODE and the RDB$MESSAGE_VECTOR array, the INCLUDE SQLCA statement declares the SQLERRD array.

Declare the SQLCA as shown in the following example;

```
EXEC SQL INCLUDE SQLCA
```

In SQL precompiled C programs, if you have multiple modules that use the INCLUDE SQLCA statement, you can add the EXTERNAL keyword to all but one of them.

The INCLUDE SQLCA statement defines a globally visible instance of the SQLCA structure; the EXTERNAL keyword lets you declare an external reference to the SQLCA structure. On OpenVMS, if your application shares the SQLCA among multiple images, one image must define the SQLCA while all other images must reference the SQLCA. Use the INCLUDE SQLCA EXTERNAL statement to reference the SQLCA. On Digital UNIX, you must have at most one definition of the SQLCA, regardless of whether or not your application uses multiple images.

The second and third elements of the SQLERRD array are of most interest to programmers.

The second element of the SQLERRD array returns information about whether or not a dynamic SQL statement is a SELECT statement. To use the information from the SQLERRD array in dynamic SQL, you must refer to the SQLCA in both the PREPARE statement and in the EXECUTE statement.

The third element of the SQLERRD array is updated after execution of every FETCH, INSERT, UPDATE, and DELETE statement to contain the count of rows processed by the statement or, in the case of the FETCH statement, the ordinal position of the fetched row in a cursor. You may want to display or log the information in this parameter as part of your program.

If you do not plan to use the row count value, you may prefer not to include the SQLERRD declaration in your program. In this case, you can substitute an explicit declaration of SQLCODE in place of the INCLUDE SQLCA statement.

If you do not specify the INCLUDE SQLCA statement and plan to refer to the RDB$MESSAGE_VECTOR array, you must also explicitly declare the RDB$MESSAGE_VECTOR array.

See Chapter 10 for more information about the INCLUDE SQLCA statement and declaring and using SQLCODE and RDB$MESSAGE_VECTOR.

8.6.3 Using the INCLUDE SQLDA or SQLDA2 Statement

The INCLUDE SQLDA and INCLUDE SQLDA2 statements declare parameters that are coded into the SQL precompiler and are used in dynamic SQL.

Declare the SQLDA or SQLDA2 structure as shown in the following example:

```
EXEC SQL INCLUDE SQLDA
EXEC SQL INCLUDE SQLDA2
```

For more information on declaring and using the SQLDA or SQLDA2, see the Section 11.3.2.

8.6.4 Using the INCLUDE File Statement

If you maintain text files that contain language source code shared by programs for database access, you can use the INCLUDE file-spec statement to copy that code. The text file may contain SQL statements, declarations of parameters, or both. Some host languages, such as COBOL, require that declarations and executable statements be separated into different program sections. If your programming language has this requirement, specify more than one INCLUDE statement to copy in the declarations and executable statements separately. In one section of your program, you can copy a text file that contains declarations (and perhaps nonexecutable SQL statements). In another section of your program, you can copy a text file that contains executable SQL and host language statements.

To include a text file of declarations, use a statement such as the following:

```
EXEC SQL INCLUDE dcls_for_depts_table.cob
```

8.7 Using the SQL Module Language FROM path-name Clause

OpenVMS VAX OpenVMS Alpha

When you use the SQL module language, you can specify the FROM path-name clause to copy a record definition from the repository to procedures in an SQL module. The FROM path-name clause lets you create and maintain repository record definitions using data types compatible with the ones in the database. It lets you quickly define parameters for a procedure, ensuring that the parameters match columns in tables or views.

The following example shows how to declare a record in an SQL module procedure and copy the record from the repository:

```
PROCEDURE store_proc
    (SQLCODE,
     emp_rec RECORD
            FROM cddplus.records.employees  END RECORD);
```

If you use the FROM path-name clause in the SQL module, you make it easier to ensure that the parameters you declare in the SQL module match the parameters in the host language program that copies its parameters from the repository. Section 8.8 describes how to use a host language COPY or INCLUDE statement in the host language program.

Note

When you use the SQL module processor and specify the module language as C, SQL translates character string fields in the record as null-terminated strings. See Section 8.12.2 for more information about how SQL handles character strings in the C language. ♦

8.8 Using Host Language COPY or INCLUDE Statements

When you use the SQL module language, you can specify a host language COPY or INCLUDE statement in your host language program to add parameter declarations of used in calls to procedures in an SQL module.

You cannot specify a host language COPY or INCLUDE statement to add parameter declarations used in SQL statements that are embedded in a host language source file. Unlike the SQL module processor, the SQL precompiler must access definitions of parameters to which SQL statements refer. The precompiler does not recognize a host language COPY or INCLUDE statement.

See your host language documentation for information about using host language COPY or INCLUDE statements.

8.9 Declaring and Using Main Parameters

In a precompiled program, usually you substitute names of host language parameters for the literals (constants) or other value expressions that you use with interactive SQL. You must precede all parameter names in precompiled SQL statements with a colon (:).

In an SQL module, rather than specifying host language parameters in SQL statements, you specify procedure parameters whose data types match those of the host language parameters used to call the procedure. When you write SQL statements in SQL modules, the use of colons in parameter names is optional. However, note the following implications:

- Using colons

  Oracle Rdb recommends that you precede each parameter name with a colon (:), although the use of colons is optional. When you use the PARAMETER COLONS clause in the module header, SQL requires that all parameters be prefixed with a colon. When prefixing parameters with colons, use colons in parameter definitions and in all references to the parameter. You must use the colons consistently throughout the module. Use of colons as prefixes to parameters will become the default behavior in a future release of Oracle Rdb.

- Not using colons

  If you do not use the PARAMETER COLONS clause in the module header, the use of colons in definitions of the parameter and in all references to the parameter is invalid. If you omit colons as a prefix, do so consistently throughout the module.

Main parameters are parameters you declare that contain values for selecting, manipulating, or storing columns. In SQL statements, main parameters usually appear:

- In the VALUES clause of the INSERT statement
- In the SET clause of the UPDATE statement
- As a WHERE clause condition value that selects rows in the:
  - UPDATE statement
  - DELETE statement
  - Singleton SELECT statement that retrieves one row
  - Column SELECT expression

  - SELECT expression of a cursor declaration that is later opened to retrieve rows you plan to update or delete, one-by-one

- In the INTO clause of SELECT or FETCH statements

Main parameters usually contain values retrieved from the database, read from a record in a file, or accepted from the terminal after the program prompts a user for input. In some cases, parameters may contain literals or values returned from a system routine.

8.9.1 Declaring Main Parameters

If you do not copy parameter declarations from a source outside your program, you must declare the parameters before you use them. Your options for declaring parameters used in SQL statements are limited to the types of declarations supported by your host language. If your host language allows it, you can declare main parameters individually or as structures, whether you use the SQL precompiler or the SQL module processor.

If you use the SQL precompiler, declare main parameters in the source file as you would any other declaration. For example, to declare the EMPLOYEE_RECORD structure in a precompiled COBOL program, use the following code:

```
01  EMPLOYEE_RECORD.
    05  EMP_ID_P               PIC X(5).
    05  L_NAME_P               PIC X(15).
    05  F_NAME_P               PIC X(11).
    05  M_INIT_P               PIC X(1).
    05  ADDRESS_1_P            PIC X(25).
    05  ADDRESS_2_P            PIC X(20).
    05  CITY_P                 PIC X(20).
    05  STATE_P                PIC X(2).
    05  POSTAL_CODE_P          PIC X(5).
    05  SEX_P                  PIC X(1).
    05  BIRTH_DATE_P           PIC X(23).
    05  STATUS_P               PIC X(1).
```

If you use the SQL module processor, declare the parameters in the host language program and in the module procedure. To declare the EMPLOYEE_RECORD structure in the host language program, use the same code as the preceding example. To declare the structure in the module procedure, use the following code:

```
PROCEDURE fetch_record
    (:emp_rec RECORD
            emp_id_p  char(5),
            l_name_p char(15),
            f_name_p char(11),
            m_init_p char(1),
```

```
            address_1_p char(25),
            address_2_p char(20),
            city_p char(20),
            state_p char(2),
            postal_code_p char(5),
            sex_p char(1),
            birth_date_p char(23),
            status_p char(1)
            END RECORD
    SQLCODE);
```

You can prefix the record name with a colon, just as you can individual parameter names.

In addition to declaring main parameters within a structure, you can declare main parameters individually. The following example shows how to declare main parameters in a COBOL host language program:

```
01  EMP_ID_P                PIC S9(4) COMP.
01  L_NAME_P                PIC S9(4) COMP.
01  F_NAME_P                PIC S9(4) COMP.
01  M_INIT_P                PIC S9(4) COMP.
01  ADDRESS_1_P             PIC S9(4) COMP.
01  ADDRESS_2_P             PIC S9(4) COMP.
01  CITY_P                  PIC S9(4) COMP.
01  STATE_P                 PIC S9(4) COMP.
01  POSTAL_CODE_P           PIC S9(4) COMP.
01  SEX_P                   PIC S9(4) COMP.
01  BIRTH_DATE_P            PIC S9(4) COMP.
01  STATUS_P                PIC S9(4) COMP.
```

Refer to Section 4.2.3 for more information about declaring parameters in SQL module procedures.

8.9.2 Using Main Parameters

This section contains examples of different ways to use parameters when you retrieve or update data. For these examples, assume that your program has declared the EMPLOYEE_RECORD structure as described in Section 8.9.1. This structure contains main parameters that correspond to all columns in the EMPLOYEES table and consists of the following fields:

```
EMPLOYEE_RECORD (name of structure)

EMP_ID_P       (1st field)
L_NAME_P       (2nd field)
F_NAME_P       (3rd field)
M_INIT_P       (4th field)
ADDRESS_1_P    (5th field)
ADDRESS_2_P    (6th field)
CITY_P         (7th field)
STATE_P        (8th field)
POSTAL_CODE_P  (9th field)
SEX_P          (10th field)
BIRTH_DATE_P   (11th field)
STATUS_P       (12th field)
```

The following examples show how to use parameters in SQL statements in SQL module language and precompiled SQL.

In the following examples, assume that columns always store values (or, in some cases, null values) for all rows being processed. In practice, some statements in these examples would produce errors when used with the sample personnel database because certain columns in the EMPLOYEES table contain actual values in some rows and null values in other rows. Section 8.10 discusses how to declare and use indicator parameters to allow conditional handling of null values. The following examples show the use of main parameters only:

- To display a row or include it in a report, the program usually fetches the row from a cursor and stores its column values in parameters that can be manipulated by host language statements.

  In the following example, the FETCH statement specifies the EMPLOYEE_RECORD structure:

```
DECLARE EMPLOYEES_CURSOR CURSOR FOR
        SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
               ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE,
               POSTAL_CODE, SEX, BIRTHDAY, STATUS_CODE
          FROM EMPLOYEES
         WHERE STATUS_CODE = 1

OPEN EMPLOYEES_CURSOR

FETCH EMPLOYEES_CURSOR INTO :EMPLOYEE_RECORD
```

  Instead of specifying the EMPLOYEE_RECORD structure, you can declare the variables individually and then use the following FETCH statement, which lists each parameter:

```
FETCH EMPLOYEES_CURSOR INTO :EMP_ID_P, :L_NAME_P, :F_NAME_P, :M_INIT_P,
                            :ADDRESS_1_P, :ADDRESS_2_P, :CITY_P, :STATE_P,
                            :POSTAL_CODE_P, :SEX_P, :BIRTH_DATE_P, :STATUS_P
```

See Chapter 18 for information about declaring and using cursors.

- To retrieve a single row, the program can use a singleton select statement rather than a cursor. The singleton select statement includes both parameters, which you might otherwise include in a cursor declaration, and select list items, which you might otherwise include in a FETCH statement.

  In the following example, the singleton select statement specifies the EMPLOYEE_RECORD structure:

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
       ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE, POSTAL_CODE,
       SEX, BIRTHDAY, STATUS_CODE
       INTO :EMPLOYEE_RECORD
   FROM EMPLOYEES
   WHERE EMPLOYEE_ID = :INPUT_ID
```

  In the following example, the singleton select statement lists each parameter:

```
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
       ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE, POSTAL_CODE,
       SEX, BIRTHDAY, STATUS_CODE
       INTO
       :EMP_ID_P, :L_NAME_P, :F_NAME_P, :M_INIT_P,
       :ADDRESS_1_P, :ADDRESS_2_P, :CITY_P, :STATE_P, :POSTAL_CODE_P,
       :SEX_P, :BIRTH_DATE_P, :STATUS_P
   FROM EMPLOYEES
   WHERE EMPLOYEE_ID = :INPUT_ID
```

  See the *Oracle Rdb7 Introduction to SQL* and the *Oracle Rdb7 SQL Reference Manual* for information about singleton select statements.

- To insert a row, the host language puts values into parameters, and then SQL stores the set of values as a row, as the following example shows:

```
INSERT INTO EMPLOYEES (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
                       ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE,
                       POSTAL_CODE, SEX, BIRTHDAY, STATUS_CODE)
               VALUES
                      (:EMPLOYEE_RECORD)
```

  If the program is storing values in only some columns of a row, the INSERT statement lists only the columns and the associated parameters being used, as the following example shows:

```
INSERT INTO EMPLOYEES (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL)
               VALUES
                      (:EMP_ID_P, :L_NAME_P, :F_NAME_P, :M_INIT_P)
```

When you insert a row, but store values in only some of the columns of the row, columns that are in the table but not specified in the INSERT statement will contain null values.

- To update or delete a row, the host language puts the values on which the row selection is based into parameters.

  In the example that follows, INPUT_ID is a parameter that accepts an employee ID number from the terminal:

```
DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = :INPUT_ID
```

- To update a row, the host language stores new values in parameters. SQL uses these values to replace existing column values in the row. The following example shows how to use cursors with the parameter INPUT_ID to update a row:

```
DECLARE FIND_EMPLOYEE CURSOR FOR
        SELECT LAST_NAME, ADDRESS_DATA_1, ADDRESS_DATA_2,
               CITY, STATE, POSTAL_CODE
        FROM  EMPLOYEES
        WHERE EMPLOYEE_ID = :INPUT_ID

OPEN FIND_EMPLOYEE

FETCH FIND_EMPLOYEE INTO :L_NAME_P, :ADDRESS_1_P, :ADDRESS_2_P,
                         :CITY_P, :STATE_P, :POSTAL_CODE_P

UPDATE EMPLOYEES
   SET
        LAST_NAME      = :L_NAME_P,
        ADDRESS_DATA_1 = :ADDRESS_1_P,
        ADDRESS_DATA_2 = :ADDRESS_2_P,
        CITY           = :CITY_P,
        STATE          = :STATE_P,
        POSTAL_CODE    = :POSTAL_CODE_P
   WHERE CURRENT OF FIND_EMPLOYEE
```

  In SQL modules, in addition to using the parameter in the DECLARE CURSOR, FETCH, and UPDATE statements, you must declare INPUT_ID as a parameter for the procedure that opens the cursor.

Refer to Section 4.2.3 for more information about using parameters in SQL module procedures. Section 8.10 provides examples similar to the ones in this section, but adds indicator parameters to allow conditional handling of null values.

8.10 Declaring and Using Indicator Parameters

An **indicator parameter** is a parameter whose value specifies whether or not its associated main parameter has been assigned a null value. If you expect that a main parameter will sometimes receive from or pass to the database null values, you must specify an indicator parameter with the name of the main parameter in an SQL statement. If you do not use an indicator parameter, and a column contains a null value, SQL generates an error, aborts the execution of the statement, and returns an error. If this happens, you cannot rely on the values passed to any of the parameters.

An indicator parameter also specifies if a text string passed from a database has been truncated. If the text string was truncated, SQL assigns the length of the string in the database to the indicator parameter. SQL also returns a warning message in SQLSTATE if you specify SQL92 as the dialect.

Remember that virtual columns usually contain null values if SQL processes no rows when evaluating the expression that returns a value to the column. Null results are possible when a column in a result table contains the result of an arithmetic calculation or the functions MIN, MAX, AVG, and SUM. Therefore, define indicator parameters for these columns.

You do not need an indicator parameter when retrieving results returned by the COUNT function because SQL returns a zero, rather than a null value, when it finds no rows that meet the criteria specified for processing by that function.

8.10.1 Declaring Indicator Parameters

You can declare indicator parameters directly in your host language program or copy declarations from a source outside your program. Although indicator parameters are not defined in a database repository node, you can use a repository utility to create a record at another node to define indicator parameters for one or more tables. You can then copy that record into programs.

In most languages, you declare an indicator parameter as a signed longword or as some element of a signed longword array. When you refer to individually named main parameters, always declare and refer to individually named indicator parameters. When you use a host language structure instead of individually named main parameters, always declare an indicator array. In other words, when you use a structure (record) to contain main parameters, you must use an indicator array to contain indicator parameters. For example, if you use the EMPLOYEE_RECORD structure set up in Section 8.9.1, declare an indicator array with 12 elements of a field named IND_ITEM. (You need

at least as many elements of an item in an array as there are fields in the structure to which you refer.)

The following example shows how to declare an indicator array in a precompiled COBOL program:

```
01  IND_ARRAY.
    05  IND_ITEM                OCCURS 12 TIMES
                                 PIC S9(9) COMP.
```

If you use the SQL module language, not only do you declare an indicator array in the host language program, you must declare indicator arrays in the SQL module procedures, using code similar to the following:

```
PROCEDURE proc_1
    (:ind_array RECORD
          INDICATOR ARRAY OF 12 INTEGER
      END RECORD,
      SQLCODE);
```

Because you cannot explicitly refer to an element of an indicator array, you must declare and refer to individually named indicator parameters when you refer to individually named main parameters.

For example, in a precompiled COBOL program or a COBOL host language program that calls SQL module procedures, use the following code to declare indicator parameters to correspond to the 12 fields in the EMPLOYEE_ RECORD structure:

```
01  EMP_ID_IND               PIC S9(9) COMP.
01  L_NAME_IND               PIC S9(9) COMP.
01  F_NAME_IND               PIC S9(9) COMP.
01  M_INIT_IND               PIC S9(9) COMP.
01  ADDRESS_1_IND            PIC S9(9) COMP.
01  ADDRESS_2_IND            PIC S9(9) COMP.
01  CITY_IND                 PIC S9(9) COMP.
01  STATE_IND                PIC S9(9) COMP.
01  POSTAL_CODE_IND          PIC S9(9) COMP.
01  SEX_IND                  PIC S9(9) COMP.
01  BIRTH_DATE_IND           PIC S9(9) COMP.
01  STATUS_IND               PIC S9(9) COMP.
```

If you use the SQL module language, you must also declare individually named indicator parameters in the SQL module procedures using code similar to the following:

```
PROCEDURE  proc_1
     (:EMP_ID             CHAR(5),
      :EMP_ID_IND         INTEGER,
      :L_NAME             CHAR(15),
      :L_NAME_IND         INTEGER,
      :F_NAME             CHAR(11),
      :F_NAME_IND         INTEGER,
      :M_INIT             CHAR(1),
      :M_INIT_IND         INTEGER,
      :ADDRESS_1          CHAR(25),
      :ADDRESS_1_IND      INTEGER,
      :ADDRESS_2          CHAR(20)
      :ADDRESS_2_IND      INTEGER,
      :CITY               CHAR(20),
      :CITY_IND           INTEGER,
      :STATE              CHAR(2),
      :STATE_IND          INTEGER,
      :POSTAL_CODE        CHAR(5),
      :POSTAL_CODE_IND    INTEGER,
      :SEX                CHAR(1),
      :SEX_IND            INTEGER,
      :BIRTH_DATE         CHAR(23),
      :BIRTH_DATE_IND     INTEGER,
      :STATUS             CHAR(1),
      :STATUS_IND         INTEGER);
```

8.10.2 Using Indicator Parameters

As Section 8.10.1 explains, always use an indicator array when you use host language structures. Always use individually named indicator parameters when you use individually named main parameters. In addition, note that you cannot use indicator arrays or host language structures, only individually named indicator parameters, in the SQL UPDATE statement or the WHERE clause.

You append the indicator parameter name to the main parameter name, using colons with indicator parameters just as you do with main parameters, as the following example shows:

```
INSERT INTO EMPLOYEES
        VALUES (:EMPLOYEE_RECORD INDICATOR :IND_ITEM);
```

You cannot explicitly refer to an element of an indicator array.

Using the example of EMPLOYEE_RECORD, which contains fields that correspond with each of the columns in the EMPLOYEES table, you declare and refer to an indicator parameter whenever you refer to a field that may sometimes receive or transfer a null value.

For example, the MIDDLE_INITIAL column of the EMPLOYEES table is likely to contain a null value in some rows because some employees do not have middle names. A reference to M_INIT_P (in EMPLOYEE_RECORD) requires a reference to an indicator parameter as well.

If you declare an indicator parameter named M_INIT_IND, append the indicator parameter name to the main parameter name, as the following example shows:

```
UPDATE EMPLOYEES
     SET MIDDLE_INITIAL = :M_INIT_P INDICATOR :M_INIT_IND
```

In the preceding example, the indicator parameter M_INIT_IND could correspond either to a host language actual parameter that is individually declared as a signed longword or to some element of a signed longword array. The array element for an actual parameter that stores an indicator parameter value is not mapped to any record by an SQL processor. Therefore, you can declare an indicator array for use with a host language structure to contain only as many elements as there are columns that can contain null values, and you decide how elements in the array should correspond to indicator parameters in the SQL statement.

In the previous examples, the parameter M_INIT_P, and other parameters that correspond to columns in the EMPLOYEES table, store a value being retrieved from or moved to the database. The indicator parameter M_INIT_IND or an occurrence of IND_ITEM contains a number that tells the program (or SQL) if it should access the value in M_INIT_P. If the indicator parameter contains a 0 or any positive integer, then the program (or SQL) knows that the main parameter M_INIT_P has a value. If the indicator parameter contains a negative integer, then the program (or SQL) knows that M_INIT_P has a null value.

How the program uses indicator parameters depends on whether it is using parameters to retrieve from or write to the database.

8.10.3 Using Indicator Parameters When Retrieving Values

When the program retrieves values from the database, it performs the following operations:

1. The program fetches or selects a row into individually named parameters and corresponding indicator parameters, or into a host language structure and indicator array.

For example, the following statement could appear in precompiled programs or in SQL module procedures:

```
FETCH EMPLOYEES_CURSOR INTO :EMP_ID_P      :EMP_ID_IND,
                            :L_NAME_P      :L_NAME_IND,
                            :F_NAME_P      :F_NAME_IND,
                            :M_INIT_P      :M_INIT_IND,
                            :ADDRESS_1_P   :ADDRESS_1_IND,
                            :ADDRESS_2_P   :ADDRESS_2_IND,
                            :CITY_P        :CITY_IND,
                            :STATE_P       :STATE_IND,
                            :POSTAL_CODE_P :POSTAL_CODE_IND,
                            :SEX_P         :SEX_IND,
                            :BIRTH_DATE_P  :BIRTH_DATE_IND,
                            :STATUS_P      :STATUS_IND
```

The following example could appear in precompiled programs or in SQL module procedures:

```
FETCH EMPLOYEES_CURSOR
      INTO :EMPLOYEE_RECORD :IND_ITEM
```

2. The program checks the values of each indicator parameter or array element to determine which, if any, columns have null values.

   In a FETCH or SELECT statement, SQL sets the value of indicator parameters corresponding to each column in the row being fetched. SQL sets the value of an indicator parameter to –1 if the column has a null value, or to 0 or a positive integer if the column contains a value.

   If the column contains a truncated text string, SQL sets the value of the indicator parameter to the length of the source string. However, if another data type was converted to a text string and that string was truncated, SQL sets the indicator parameter to the full length of the converted text string, not the original length. For example, if you assign a date field to a character host language variable, SQL converts the date to text. Because a full text representation of a date is 16 characters, if the host language variable is less than 16 characters, SQL sets the value of the indicator parameter to 16.

   If the column contains a value other than a truncated text string, SQL sets the value of the indicator parameter to 0.

3. The program uses values only from host language parameters whose corresponding indicator parameters store 0 or a positive integer.

   If the program displays or writes values from parameters whose indicator parameters store –1, the program is working with values belonging to a previously fetched row rather than the current row.

8.10.4 Using Indicator Parameters When Storing Values

When the program uses a parameter to transfer values to the database, it:

1. Initializes host language indicator parameters or array elements to 0 (for stored value default)
2. Moves values to main parameters to assign values that are not null
3. Moves –1 (for null) to the indicator parameter or indicator array element associated with any main parameter to which the program has not moved a value
4. Executes an UPDATE or INSERT statement to process the row

The following INSERT statement assumes the following:

- The fourth column, MIDDLE_INITIAL, varies between storing null and actual values. As a result, it uses an indicator parameter for this column.
- All other columns store actual values. As a result, the statement lists all other columns without indicator parameters.
- The sixth column, ADDRESS_DATA_2, always has a null value because the INSERT statement does not specify ADDRESS_DATA_2 in the list of main parameters.

```
INSERT INTO EMPLOYEES
           (
           EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
           ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE,
           SEX, BIRTHDAY, STATUS_CODE
           )
       VALUES
           (
           :EMPLOYEE_ID_P, :LAST_NAME_P, :FIRST_NAME_P,
           :MIDDLE_INITIAL_P :INDICATOR :MIDDLE_INITIAL_IND,
           :ADDRESS_1_P, :CITY_P, :STATE_P, :POSTAL_CODE_P,
           :SEX_P, :BIRTH_DATE_P, :STATUS_P
           )
```

The INSERT statement can also use a structure to store parameters. In this case, the host language program initializes all indicator parameters in the indicator array IND_ITEM, except the sixth, to 0 (for stored value), initializes the sixth indicator array element to –1 (for null), and depending on whether the value of MIDDLE_INITIAL_P is an actual value or a null value, sets the fourth element of the structure to 0 or –1.

The following SQL precompiler example uses the EMPLOYEE_RECORD structure to store parameters:

```
INSERT INTO EMPLOYEES
            (
            EMPLOYEE_ID, LAST_NAME, FIRST_NAME,
            MIDDLE_INITIAL, ADDRESS_DATA_1,
            ADDRESS_DATA_2, CITY, STATE,
            POSTAL_CODE, SEX, BIRTHDAY, STATUS_CODE
            )
       VALUES
            ( :EMPLOYEE_RECORD:IND_ITEM )
```

In precompiled programs, SQL stores values only from main parameters whose associated indicator parameters or indicator array elements are set to 0 or a positive integer. If any indicator parameter is set to a negative integer, SQL does not store a value from the associated main parameter whether or not the program moved a value to that parameter.

In SQL modules, SQL stores values only from main parameters whose associated indicator parameters are set to 0 or to a positive integer in the calling host language module. If any indicator parameter is set to a negative integer in the calling host language module, SQL does not store a value from the procedure parameter for the column value whether or not the program moved a value to the main parameter that matches the column.

Note

Indicator parameters in Boolean expressions may not behave as you expect. For example, A=NULL is never true even if A is, in fact, NULL.

8.11 Avoiding Mistakes When Declaring and Using Parameters

This section summarizes some points made in preceding sections of this chapter about declaring and using parameters to which SQL statements refer.

Whether you are using the SQL precompiler or the SQL module processor, remember the following rules:

- Do not declare parameters using declaration methods or data types that SQL does not support.
- Do not forget how SQL expands structures when associating columns with main parameters.

Suppose you include in your program a structure definition that contains a field (ADDRESS_2_P) for the ADDRESS_DATA_2 column in the EMPLOYEES table. Assume you want this column to always have a null value. In this case, you cannot omit the ADDRESS_DATA_2 column from the list of columns in your SQL statement and still refer to the structure name when you insert or fetch rows.

If you omit the ADDRESS_DATA_2 column from the columns that you list, but use the structure definition (with ADDRESS_2_P included), the SQL precompiler and SQL module processor incorrectly associate columns with fields in the structure. In other words, they assume that you want the value in the ADDRESS_2_P field stored in the column CITY, the value in the CITY_P field stored in the column STATE, and so forth.

When you specify a structure name, compile-time errors are generated by:

- An invalid conversion between the storage format defined for a column and the storage format defined for the associated field in the structure
- A structure that contains more fields than the number of columns specified by the SQL statement
- A structure that contains fewer fields than the number of columns specified by the SQL statement

However, inadvertently mismatching a column list with a structure does not always produce a compile-time error. The mistake may produce only a warning (perhaps a warning about truncation of character strings) or no indication at compile time that anything is wrong. This situation might arise if you substitute another column name for the one you intend and both columns have compatible definitions. It can arise also if someone deletes and then redefines a table (containing columns with similar data types) to rearrange the columns, and your SQL statement uses the asterisk (*) to specify columns rather than listing the names of all columns. You should explicitly use column names rather than asterisks.

- When you use an indicator array, make sure that you have sufficient array elements for the number of fields in the structure.

  If you have more fields in a structure than there are elements in the indicator array, SQL returns a warning.

- Use individually named indicator parameters and indicator arrays appropriately.

If your SQL statement refers to an individually named main parameter associated with a column, you must update, evaluate, and refer to an individually named indicator parameter. If your SQL statement refers to a structure, you must update, evaluate, and refer to an indicator array element.

If you confuse the two types of indicator parameters in an SQL statement, you encounter a compile-time error. If you confuse the two types of indicator parameters in host language statements that evaluate or update the indicator parameters (assuming you are using both types of indicator parameters in the same program), your program will not recognize or store actual and null values correctly at run time.

- Do not forget how SQL associates columns with fields when a structure contains subordinate structures.

  SQL expands all structures to a list of elementary fields in the structure, and this list includes elementary fields for any subordinate structures. Then SQL associates a column with each elementary field in the structure. This means that, in some situations, you create a structure with more elementary fields than there are columns.

  For example, suppose you define a structure that contains the elementary fields YEAR_P, MONTH_P, and DAY_P, and you refer to the structure using the name DATE_GROUP. Even though DATE_GROUP may be logically associated with only one column in a table, DATE_GROUP is a structure that contributes three elementary fields to the record. Therefore, SQL assumes that the three fields in DATE_GROUP are associated with three different columns.

  If you forget this information, you can encounter various unexpected run-time problems for which solutions are not intuitively obvious.

  There is an easy solution to this problem if your host language lets you redefine the same storage area. The solution is possible because SQL expands structures based on the first definition of a given storage area in the structure. Therefore, if you are mapping multiple field names to an area that should be associated with only one column, make sure that your first definition of the area specifies an elementary field. Your second (and any subsequent) definitions of the same area can specify a group of fields.

  You can find examples of this solution in the sample programs in the samples directory. The sample programs do not nest structures in records. However, the programs do illustrate the general principle of first defining a given storage area as one elementary field before defining that area as multiple fields. The principle applies whenever you are associating more

than one field with a column, including those times when you want to do this within a record that applies to a row.

For more information, check the sample programs for your host language in the samples directory and read information about supported declarations and data types in the *Oracle Rdb7 SQL Reference Manual*.

8.11.1 Avoiding Mistakes When Using Embedded SQL

Remember the rules listed in Section 8.11 when you use embedded SQL. In addition, do not declare SQLCODE and also specify the INCLUDE SQLCA statement in the same source file. This results in two declarations of SQLCODE, and SQL uses only the SQLCA. Therefore, if you specify the INCLUDE SQLCA statement, do not declare SQLCODE separately.

8.11.2 Avoiding Mistakes When Using SQL Modules

The most important rule for you to remember when using SQL modules is that the data type, size, and passing mechanism for an actual parameter should be identical to the data type, size, and passing mechanism for a parameter in the SQL module procedure being called. Therefore, for both procedure and actual parameters, you are limited to the set of data types that both your host language and SQL support. In some cases, you may want to convert between data types. If so, data conversion is always performed within one of the modules and never when a value is passed between modules.

In addition, remember that SQL implements support for any SQL data type keyword in one of two ways:

- For both a column definition and a parameter declaration
- For a parameter declaration only

For example, CHAR(n) always specifies a character-string storage format of *n* characters whether you are declaring a parameter in an SQL module or you are defining a table column. However, DECIMAL(n) defines a packed decimal field of *n* digits only for a parameter declared in an SQL module procedure. For columns, DECIMAL(n) defines either a fixed-point binary or floating-point binary storage format, depending on the value of *n*. (Oracle Rdb does not implement the packed decimal storage format.) Therefore, if you define a procedure parameter in an SQL module using DECIMAL, SQL converts the data stored in a column corresponding to the procedure parameter to packed decimal before transfer to the host language module. See the *Oracle Rdb7 SQL Reference Manual* for more information about SQL data types.

The following are additional considerations for declaring and specifying parameters when you use SQL module language:

- Oracle Rdb recommends that you use the colon (:) before parameter names in SQL statements, although the use of colons is optional by default. If you use colons, use the PARAMETER COLONS clause in the module header.
- In an SQL statement, if you specify a parameter and a column that have the same name, you must qualify the column name using a table name if the parameter does not use colons. If you use the PARAMETER COLONS clause in the module header, SQL recognizes the column as a column because the parameter must be prefixed by a colon for the SQL statement to be valid.

8.12 Declaring and Using Parameters in Specific Languages

This section provides information about using parameters with particular host languages.

The sql_all_datatypes sample program in the samples directory illustrates most of the declarations and operations discussed in this chapter. This example is available in all languages supported by the SQL precompiler and in SQL module language. In the precompiled programs, the comments in these programs refer to main variables and indicator variables instead of main parameters and indicator parameters.

8.12.1 Declaring and Using Parameters in Ada Source Files

This section describes specific guidelines for declaring and using parameters in Ada language programs. See Section 6.7.1 for additional rules.

Using Ada Packages

SQL lets you declare host language variables either directly or by calling the Ada package SQL_STANDARD. See the *Oracle Rdb7 SQL Reference Manual* for more information about using SQL with Ada packages.

Dollar Signs Not Allowed

Ada does not permit the dollar sign ($) in names you specify in programs. In Ada programs, when you refer to names of parameters and calls that this manual specifies as beginning with SQL$ or RDB$, substitute an underscore (_) for the dollar sign ($). For example, specify RDB_LU_STATUS for RDB$LU_STATUS.

8.12.2 Declaring and Using Parameters in C Source Files

This section describes specific guidelines for declaring and using parameters in C language programs. See Section 6.7.3 for additional rules.

Case Sensitivity of SQLCODE and SQLSTATE

If you are using the SQL precompiler, note that you must use uppercase to declare and refer to the SQLCODE or SQLSTATE parameter. The SQL precompiler converts SQL statements to host language calls. It specifies SQLCODE and SQLSTATE in uppercase when listing parameters on each of these host language calls.

Declaring SQLCODE

When you declare SQLCODE independently, declare it as a signed longword. Note that when you use the C language, the actual length (32 bits or 64 bits) depends on how the host language compiler interprets a longword. SQL supports both 32-bit and 64-bit longwords.

When you use the SQL precompiler, the support is automatic because the precompiler sees the definition of SQLCODE.

When you use the SQL module language, declare SQLCODE as a longword in your host language program and compile the SQL module using the `-lsqlcode` or LONG_SQLCODE switches.

Aligning Data

OpenVMS Alpha

When you use an SQL module called by a C program and the LANGUAGE clause specified in the module header is C, use the ALIGN_RECORDS qualifier.

In particular, you should use this qualifier when you use C and compile the host language program with the FLOAT=D_FLOAT qualifier. ♦

Declaring SQLCA and Alignment

OpenVMS Alpha

On OpenVMS Alpha, when you use the SQL module processor and specify C in the module header language clause or when you use the SQL precompiler for C, SQL aligns fields in structures by default. However, you should not allow member alignment of the SQLCA.

If you explicitly define the SQLCA structure, surround the structure definitions with the C preprocessor directive #pragma nomember_alignment to prevent alignment of the structures.

If you use the SQL INCLUDE statement, you do not need to use the preprocessor directive. ♦

Character Strings

When you use the SQL C precompiler or the SQL module processor and specify C as the module language, SQL translates all C character strings as null-terminated strings. This means that when SQL passes these character strings from the database to the program, it reserves space at the end of the string for the null character. When a program passes a character string to the database for input, SQL looks for the null character to determine how many characters to store in the database. SQL stores only those characters that precede the null character; it does not store the null character itself.

Because of the way SQL translates C character strings, you may encounter problems with applications that pass binary data to and from the database. To avoid these problems when you use the SQL C precompiler, use the $SQL_VARCHAR data type that SQL provides. To avoid these problems when you use the SQL module language with a C language host program, specify the module language as GENERAL.

In addition, the way SQL translates C character strings will affect programs that use the SQL INCLUDE or the SQL FROM DICTIONARY clause to copy record definitions from a repository.

Integer Data Types on Digital UNIX

Digital UNIX

On Digital UNIX, there is a distinction between integer and longword variables that does not exist on OpenVMS. The following table shows the three integer data types for C compilers on Digital UNIX:

Data Type	Size of Data Type
short (int)	16-bits
int	32-bits
long (int)	64-bits

Applications written in C must use the proper declarations when calling SQL module language programs. The ANSI/ISO standard states that the calling program must pass a pointer to a longword. This applies to the mapping of SQL module language formal parameters of type INTEGER and their C actual parameter.

SQL assumes that INTEGER parameters are called with pointers to C “long”. If you use the –int32 command line option, SQL assumes that the parameters are called with pointers to C “int”.

To override the size specified by the –int command line options, specify that an INTEGER parameter is 4 or 8 bytes, as the following example shows:

```
salary_amount INTEGER is 4 BYTES
```

The SQL precompiler interprets the sizes of these data types correctly. ♦

Using the SMALLINT Data Type

OpenVMS VAX

Because VAX C does not support scaled exact numerics, if a column in a table uses the SMALLINT(2) data type, you must have SQL convert the data for you. For example, if you are using a cursor to fetch three columns from a table, use an SQL module language declaration and procedure similar to the following:

```
DECLARE my_cursor CURSOR FOR
    SELECT X, Y, Z
    FROM MYTABLE
    WHERE W = IN_DATA
   .
   .
   .

PROCEDURE FETCH_EM
        SQLCODE
        A       CHAR(5)
        B       CHAR(10)
        C       REAL;   -- Corresponds to column Z of type SMALLINT(2)

    FETCH my_cursor INTO A, B, C;
```

The C language host program calls the procedure FETCH_EM as shown in the following example:

```
   .
   .
   .
static char in[5] = "0001";
static char out_a[6];
static char out_b[11];
static float out_c;
   .
   .
   .
    FETCH_EM(&sqlcode, out_a, out_b, &out_c);
    if (sqlcode == 100)
        break;
    if (sqlcode < 0)
   .
   .
   .
```

Note that the parameter of data type REAL is mapped to a float host language variable. ♦

8.12.3 Declaring and Using Parameters in COBOL Source Files

If you are using the SQL precompiler, see Section 6.7.4, which covers general rules that apply to embedding SQL statements in COBOL source files.

8.12.4 Declaring and Using Parameters in FORTRAN Source Files

If you are using the SQL precompiler, see Section 6.7.5, which covers general rules that apply to embedding SQL statements in FORTRAN source files.

8.12.5 Declaring and Using Parameters in Pascal Source Files

If you are using the SQL precompiler, see Section 6.7.6, which covers general rules that apply to embedding SQL statements in Pascal source files.

8.12.6 Declaring and Using Parameters in PL/I Source Files

If you are using the SQL precompiler, see Section 6.7.7, which covers general rules that apply to embedding SQL statements in PL/I source files.

8.12.7 Declaring and Using Parameters in SQL Modules and Calling Programs

For more information about declaring and using parameters in SQL module language, see Chapter 4. Section 4.2.4 discusses the association of parameters in called and calling programs.

The sql_all_datatypes_ada.sqlmod module file in the samples directory illustrates how to use parameters in an SQL module. The sql_all_ datatypes.ada Ada source program in the samples directory calls procedures in the sql_all_datatypes_ada.sqlmod module.

9

Using Date-Time Data Types

Oracle Rdb provides a set of date-time data types and value functions that adhere to the rules prescribed in the ANSI/ISO SQL standard. These date-time data types and value functions allow programmers and database administrators greater control over date, time, and interval data.

The following sections explain how to:

- Store data in data-time data types
- Use date-time data types in programs
- Plan for portability of date-time data types
- Convert applications and databases to use a different date-time data type
- Handle DATE VMS data types
- Port applications that contain DATE VMS data types
- Use date-time data types with dynamic SQL

This chapter includes a series of examples to help you program with date-time data types and value functions. The examples use the corporate_data sample database.

Reference Reading

If you are not familiar with the date-time data types, refer to the *Oracle Rdb7 Introduction to SQL*. For information about creating tables and domains using these data types, see the *Oracle Rdb7 Guide to Database Design and Definition*.

Refer to the *Oracle Rdb7 SQL Reference Manual* for information such as date-time data type formats, data type conversion rules, date-time literals, value functions, valid operators, and language-specific date-time data type declarations.

9.1 Storing Data in Date-Time Data Types

When you store data in columns with a date-time data type, you must consider the following two points:

- When you use date-time literals and default values, the leading and fractional seconds precision of the default value must match exactly those of the column. They must also use the same interval qualifiers.
- The literal itself does not need leading zeros to match the precision, as long as the leading precision and fractional seconds precision specified is greater than or equal to the value of the literal. For example, the following two interval literal expressions are equivalent and valid:

```
INTERVAL '12:30' HOUR(4) TO MINUTE
INTERVAL '0012:30' HOUR(4) TO MINUTE
```

Example 9–1 shows how to insert date-time data into the DAILY_HOURS table in the corporate_data database.

Example 9–1 Inserting Data-Time Data

```
SQL> INSERT INTO ADMINISTRATION.ACCOUNTING.DAILY_HOURS
cont>              (EMPLOYEE_ID, START_TIME, END_TIME)
cont>       VALUES ('00415',
cont>               TIMESTAMP '1995-07-15 07:30:45.33',
cont>               TIMESTAMP '1995-07-15 17:15:22.18');
1 row inserted
SQL>
SQL> SELECT * FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS;
 EMPLOYEE_ID   START_TIME               END_TIME
 00415         1995-07-15 07:30:45.33   1995-07-15 17:15:22.18
1 row selected
SQL>
```

The following examples show how to store date-time data in your database. Suppose, for example, that you create the following table using date-time data types:

```
SQL>  CREATE TABLE DATE_TABLE
cont>  ( CHAR_VAR   CHAR(20),
cont>    COL_DATE   TIMESTAMP(0)
cont>        CHECK (COL_DATE BETWEEN TIMESTAMP '1995-1-1 00:00:00'
cont>               AND TIMESTAMP '1996-1-1 00:00:00')  NOT DEFERRABLE,
cont>    COL_YEAR_TO_MONTH     INTERVAL YEAR TO MONTH
cont>        CHECK (COL_YEAR_TO_MONTH >= INTERVAL '00-00' YEAR TO MONTH)
cont>               NOT DEFERRABLE,
cont>    COL_DAY_TIME          INTERVAL HOUR TO SECOND);
```

When you use the INSERT statement to store date-time data, you must pay special attention to leading and fractional precision. If you try inserting literal timestamp data of precision TIMESTAMP(2) into the COL_DATE column defined as precision TIMESTAMP(0), SQL issues an error message:

```
SQL> INSERT INTO DATE_TABLE  (COL_DATE)
cont>   VALUES ( TIMESTAMP '1995-6-12 16:12:32.05' );
%SQL-F-UNSDATASS, Unsupported date/time assignment from <Source> to COL_DATE
-SQL-F-DATESCANEQ, Date/time expressions with different fractional seconds
 precision are not comparable
```

The following example fails because the leading precision for the YEAR TO MONTH interval is too small:

```
SQL> INSERT INTO DATE_TABLE   (COL_YEAR_TO_MONTH)
cont>  VALUES (INTERVAL '26512-10' YEAR(4) TO MONTH);
%SQL-F-DATCONERR, Data conversion error for string '26512-10'
-COSI-F-IVTIME, invalid date or time
```

The following example succeeds because the precisions are greater than or equal to the literal representations. This example succeeds with both the original domain definition and the altered domain definitions. As long as the literal is of the same interval type, SQL converts the data to the domain's declared data type.

```
SQL> INSERT INTO DATE_TABLE
cont>       VALUES ('STRING',
cont>               TIMESTAMP '1995-6-12 16:12:32',
cont>               INTERVAL '265-10' YEAR(3) TO MONTH,
cont>               INTERVAL '73:23:34' HOUR(4) TO SECOND);
```

9.2 Using Date-Time Data Types in Programs

The SQL module language and the SQL precompiler handle date-time data types in different ways. This section provides examples of using date-time data types in programs using either processor. It shows how to convert a date-time character string to the date-time format understood by Oracle Rdb and how to convert it back to a character string. Data type conversion of this type is critical to programming successfully with date-time data types.

When you assign a character string literal or variable to a DATE ANSI column, you use the following format where yyyy is the four digits representing the year, nn is the two digits representing the month, and dd is the two digits representing the date.

```
yyyy-nn-dd
```

See the *Oracle Rdb7 SQL Reference Manual* for information about the format of other date-time data types.

To establish an ANSI/ISO environment for the DATE data type and the CURRENT_TIMESTAMP value function, use the DEFAULT DATE FORMAT clause in a DECLARE MODULE statement or the module header.

Note that the CURRENT_TIMESTAMP value function behaves differently depending on the default date format. See the *Oracle Rdb7 SQL Reference Manual* for more information.

9.2.1 Converting Date-Time Data Types for Program Development

In some cases, you need to convert date-time character strings to the date-time format accepted by Oracle Rdb to enable the correct processing of date-time data types. For example, to store complete rows using record structures containing date-time data types, convert these data types to the Oracle Rdb date-time format. Likewise, convert the date-time format back to a character string for displaying in a program.

Example 9–2 shows SQL module procedures that convert character strings to and from the date-time format required by Oracle Rdb. When using SQL module language, host language variables used for binary representation should be declared as quadwords or the language equivalent of quadword.

Example 9–2 SQL Module Segment for Converting Date-Time Data Types

```
-- This procedure converts a character string to the DATE ANSI format.
PROCEDURE CVT_TO_DATETIME
   (SQLCODE,
   :CHAR_VAR     CHAR(23),
   :DT_VAR       DATE ANSI);

   BEGIN
      SET :DT_VAR = CAST(:CHAR_VAR AS DATE ANSI); ❶
   END;

-- This procedure converts a character string to the INTERVAL format.
PROCEDURE CVT_TO_INTERVAL
   (SQLCODE,
   :CHAR_VAR     CHAR(23),
   :DT_VAR       INTERVAL DAY(6) TO SECOND);

   BEGIN
     SET :DT_VAR = CAST(:CHAR_VAR AS INTERVAL DAY(6) TO SECOND); ❷
   END;
```

(continued on next page)

Example 9–2 (Cont.) SQL Module Segment for Converting Date-Time Data Types

```
-- This procedure converts a binary value to a character string.
PROCEDURE CVT_INTERVAL_TO_CHAR
   (SQLCODE,
   :CHAR_VAR     CHAR(23),
   :DT_VAR       INTERVAL DAY(6) TO SECOND);

   BEGIN
     SET :CHAR_VAR = CAST (:DT_VAR AS VARCHAR(23)); ❸
   END;
```

The following callout descriptions are keyed to numbered items in Example 9–2:

❶ The CAST function converts a character string to a data type of DATE ANSI and returns it as an 8-byte binary value.

❷ The CAST function converts a character string to a binary data type of interval DAY TO SECOND.

❸ The CAST function converts an interval DAY TO SECOND to a character string.

You must declare all date-time data types as quadwords (char[8]) in C, as Example 9–3 demonstrates.

Example 9–3 C Program for Converting Date-Time Data Types

```
char    int_ds_string[24],
        date_ansi_string[24];

int     sql_return_status;

struct {
        char            employee_id[6];
        char            last_name[21];
        char            q_date_ansi[8];
        char            q_date_vms[8];
        char            q_time[8];
        char            q_timestamp[8];
        char            q_int_ym[8];
        char            q_int_ds[8];
        }date_rec;
```

(continued on next page)

Example 9–3 (Cont.) C Program for Converting Date-Time Data Types

```
main()
{
    strcpy(date_ansi_string,"1995-9-12\0");
    cvt_to_datetime(&sql_return_status,date_ansi_string,date_rec.q_date_ansi);

    strcpy(int_ds_string,"-12:10:35:59.05\0");
    cvt_to_interval(&sql_return_status,int_ds_string,date_rec.q_int_ds);
}
```

9.2.2 Using Date-Time Data Types with the SQL Precompiler

When you use the SQL precompiler, you can use special SQL-defined data types for variables representing date-time data types.

SQL allows you to fetch DATE, TIME, TIMESTAMP, and INTERVAL data types into binary host language variables, but the internal format in many cases is understood only by Oracle Rdb. You should only use the binary host language variables to receive data from queries, and then to pass that data back to SQL as input to other queries.

To facilitate the definition of host language variables, SQL defines a set of date-time data types for use with the SQL precompiler. For example, in a COBOL program, you can declare variables using the following data types:

```
01 P_DATE           SQL_DATE.
01 P_DATE_V         SQL_DATE_VMS.
01 P_TIME           SQL_TIME(0).
01 P_TIMESTAMP      SQL_TIMESTAMP(2).
01 P_INTER_1        SQL_INTERVAL (YEAR TO MONTH).
01 P_INTER_2        SQL_INTERVAL (DAY TO HOUR).
```

Example 9–4 uses the SQL precompiler data types to declare date-time host variables.

Example 9–4 SQL Precompiler Program Using Date-Time Data Types

```
#include <string.h>

main()
{

char str_var[24];
long SQLCODE;
```

(continued on next page)

Example 9–4 (Cont.) SQL Precompiler Program Using Date-Time Data Types

```
struct {
        char                                    employee_id[6];
        char                                    last_name[21];
        SQL_DATE_ANSI                           my_date_ansi;
        SQL_DATE_VMS                            my_date_vms;   ❶
        SQL_TIME                                my_time;
        SQL_TIMESTAMP(1)                        my_timestamp;
        SQL_INTERVAL(YEAR(3) TO MONTH)          my_int_yr32mo;
        SQL_INTERVAL(DAY(4) TO SECOND (2))      my_int_d42s2;
       }date_rec;

        strcpy(str_var,"33-4\0");

EXEC SQL DECLARE ALIAS FILENAME personnel;

/* Conversion from a string to a date-time format. */
EXEC SQL BEGIN
         SET :date_rec.my_int_yr32mo = CAST(:str_var AS INTERVAL YEAR TO MONTH);
         END;

/* Conversion from date-time format to a string. */
EXEC SQL BEGIN
         SET :str_var = CAST(:date_rec.my_int_yr32mo AS CHAR(24));  ❷
         END;
}
```

The following callout descriptions are keyed to numbered items in Example 9–4:

❶ Use SQL precompiler data types when binary format is required.

❷ Use the CAST function to convert to character strings when you want to display date-time data.

Refer to the *Oracle Rdb7 SQL Reference Manual* for a list of the special data types you can use with the SQL precompiler. The samples directory includes a program, sql_all_datatypes_date, that shows how to use these data types in a variety of languages.

Example 9–5 shows a SQL precompiler C program that computes the weekly wages from the corporate_data sample database. It is comparable to the SQL module program shown in Section 9.2.3.

Example 9–5 C Program Using Date-Time Data Types with the SQL Precompiler

```
#include <stdio.h>
#include <sql_rdb_headers.h>

/* Include the SQLCA. */
    EXEC SQL INCLUDE SQLCA;

/* Declare the alias. */
    EXEC SQL DECLARE ALIAS FILENAME corporate_data;

main()
{
/* Declare return status variable for error handling. */
    int    sql_return_status;

/* Variables for program use. */
    char  start_date[11],
          end_date[11];
    char  emp_id[6],
          last_name[21],
          hours_worked[12];
    int   hours,min;
    float hr_rate,wages,sec;

    printf("Enter START DATE (YYYY-MM-DD):");
    gets(start_date);
    printf("Enter END DATE   (YYYY-MM-DD):");
    gets(end_date);

/* Declare a cursor. */
    EXEC SQL DECLARE WEEKLY_HRS_CURSOR CURSOR FOR
             SELECT E.EMPLOYEE_ID, LAST_NAME, HOURLY_RATE,
                   CAST(SUM(HOURS_WORKED) AS CHAR(12)),
                   EXTRACT(HOUR FROM SUM(HOURS_WORKED)),
                   EXTRACT(MINUTE FROM SUM(HOURS_WORKED)),
                   EXTRACT(SECOND FROM SUM(HOURS_WORKED))

            FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS D,
                 ADMINISTRATION.PERSONNEL.HOURLY_HISTORY H,
                 ADMINISTRATION.PERSONNEL.EMPLOYEES E
            WHERE (H.EMPLOYEE_ID = D.EMPLOYEE_ID)
                  AND  (E.EMPLOYEE_ID = H.EMPLOYEE_ID)
                  AND CAST(START_TIME AS DATE ANSI)
                  BETWEEN CAST('1995-07-01' AS DATE ANSI) AND
                          CAST('1995-07-19' AS DATE ANSI)
            GROUP BY E.EMPLOYEE_ID, LAST_NAME,HOURLY_RATE;

/* Open the cursor. */
    EXEC SQL OPEN WEEKLY_HRS_CURSOR;
```

(continued on next page)

Example 9–5 (Cont.) C Program Using Date-Time Data Types with the SQL Precompiler

```
    printf("%20s  WEEKLY PAYROLL ( %s - %s)\n\n"," ",start_date,end_date);
    printf("Emp Id     Last Name %11s Hours %14s Rate %9s Wages\n\n",
           " "," "," ");

/* Get data for employees until end of file. */

    while (SQLCA.SQLCODE == 0)
    {
        EXEC SQL FETCH WEEKLY_HRS_CURSOR INTO
                :emp_id, :last_name, :hr_rate,
                :hours_worked, :hours, :min, :sec;

        if (SQLCA.SQLCODE != 0 && SQLCA.SQLCODE != 100)
        {
            sql_signal();
            EXEC SQL ROLLBACK;
        }
        else
            if (SQLCA.SQLCODE != 100)
            {
                if (hours < 40)
                    wages = (float)hours * hr_rate +
                            (float)min/60 * hr_rate +
                            (float)sec/3600 * hr_rate;
                else
                    wages = 40.00 * hr_rate +
                            (float)(hours - 40) * hr_rate * 1.5 +
                            (float)min/60 * hr_rate * 1.5 +
                            (float)sec/3600 * hr_rate * 1.5;

                printf("%s %25s %10s %15.2f %15.2f\n",
                        emp_id,last_name,hours_worked,hr_rate,wages);
            }
    }
    EXEC SQL ROLLBACK;
}
```

9.2.3 Using Date-Time Data Types with the SQL Module Language

This section illustrates how to use the CAST and EXTRACT functions to convert date-time data types to and from the date-time format to generate a report of weekly wages based on time and a half after 40 hours.

Example 9–6 shows the SQL module, mod_weekly_pay.sqlmod, that fetches rows using the date-time data types.

Example 9–6 SQL Module Using Date-Time Data Types

```
-------------------------------------------------------------------------
-- The module illustrates how to use SQL module language to use a cursor to
-- fetch columns of date-time data type.
-------------------------------------------------------------------------
-- Header Information Section
-------------------------------------------------------------------------
MODULE          MOD_WEEKLY_PAY        -- Module name
LANGUAGE        C                     -- Language of calling program
ALIAS           RDB$DBHANDLE
PARAMETER       COLONS
-------------------------------------------------------------------------
-- DECLARE Statements Section
-------------------------------------------------------------------------
DECLARE ALIAS FILENAME corporate_data   -- Declaration of the database

DECLARE WEEKLY_HRS_CURSOR CURSOR FOR ❶
       SELECT E.EMPLOYEE_ID, LAST_NAME, HOURLY_RATE,
              CAST(SUM(HOURS_WORKED) AS CHAR(12)), ❷
              EXTRACT(HOUR FROM SUM(HOURS_WORKED)),❸
              EXTRACT(MINUTE FROM SUM(HOURS_WORKED)),
              EXTRACT(SECOND FROM SUM(HOURS_WORKED))
       FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS D,
            ADMINISTRATION.PERSONNEL.HOURLY_HISTORY H,
            ADMINISTRATION.PERSONNEL.EMPLOYEES E
       WHERE ((H.EMPLOYEE_ID = D.EMPLOYEE_ID)
             AND  (E.EMPLOYEE_ID = H.EMPLOYEE_ID)
             AND (CAST(D.START_TIME AS DATE ANSI) ❹
                 BETWEEN CAST(:DATE1 AS DATE ANSI) AND ❺
                       CAST(:DATE2 AS DATE ANSI)))
       GROUP BY E.EMPLOYEE_ID, LAST_NAME, HOURLY_RATE
-------------------------------------------------------------------------
-- Procedure Section
-------------------------------------------------------------------------
-- This procedure opens the cursor that has been declared for the
-- WEEKLY_PAY table.
PROCEDURE OPEN_CURSOR
    (SQLCODE,
    :DATE1     CHAR(10),
    :DATE2      CHAR(10));

    OPEN WEEKLY_HRS_CURSOR;

-- This procedure fetches the data from the opened cursor.
```

(continued on next page)

Example 9–6 (Cont.) SQL Module Using Date-Time Data Types

```
PROCEDURE FETCH_WEEKLY_HOURS
    (SQLCODE,
    :EMP_ID             CHAR(5),
    :LAST_NAME          CHAR(20),
    :HRLY_RATE          REAL,
    :HRS_WORKED         CHAR(11),
    :WEEK_HOURS         INTEGER,
    :WEEK_MIN           INTEGER,
    :WEEK_SEC           REAL);

    FETCH WEEKLY_HRS_CURSOR INTO :EMP_ID, :LAST_NAME, :HRLY_RATE,
                                 :HRS_WORKED, :WEEK_HOURS,
                                 :WEEK_MIN, :WEEK_SEC;

-- This procedure rolls back the transaction.

PROCEDURE ROLLBACK_TRANSACTION
    (SQLCODE);

    ROLLBACK;
```

The following callout descriptions are keyed to numbered items in Example 9–6:

❶ Declares a cursor to create a record stream for each hourly employee's weekly hours.

❷ Uses the CAST function to convert the date from binary format to a character string. The SUM(HOURS_WORKED) expression assumes the data type interval DAY TO SECOND, which must be converted to a character string before displaying.

❸ Uses the EXTRACT function to create numeric values for HOUR, MINUTE, and SECOND. These values are used in the algorithm for calculating weekly wages.

❹ Uses the CAST function to convert the column START_TIME from TIMESTAMP to DATE ANSI. This example prompts the user for two dates, the desired starting and ending dates to use in calculating wages. Because only the date is significant, the CAST operation allows comparison of the date portion of the START_TIME column only.

❺ Uses the CAST function to convert the date character string (YYYY-MM-DD) entered by the user to DATE ANSI.

Example 9–7 shows the C language program, mod_datetime_c.c, used with the mod_weekly_pay.sqlmod module to retrieve date-time data types in programs.

Example 9–7 C Program Using Date-Time Arithmetic

```
/* ABSTRACT:
  This sample C program is used with mod_weekly_pay.sqlmod to demonstrate
  the retrieval of date-time data types. Note the use of the CAST and
  EXTRACT functions to convert to and from a date-time format.

  This application prompts the user for a START_DATE and END_DATE. The
  total hours worked for each employee during this time period are returned
  along with the employee's hourly rate. Weekly pay is calculated assuming
  time and a half after 40 hours.
*/
#include <stdio.h>
#include <sql_rdb_headers.h>
main()
{
/* Declare return status variable for error handling. */
    int    sql_return_status;

/* Variables for program use. */
    char  start_date[11], ❶
          end_date[11];
    char  emp_id[6],
          last_name[21],
          hours_worked[12];
    int   hours,
          min;
    float hr_rate,
          wages,
          sec; ❷
    printf("Enter START DATE (YYYY-MM-DD):");
    gets(start_date);
    printf("Enter END DATE   (YYYY-MM-DD):");
    gets(end_date);

/* Open the cursor. */

    open_cursor(&sql_return_status,start_date,end_date);
    if (sql_return_status < 0)
    {
        sql_signal();
        rollback_transaction(&sql_return_status);
    }
```

(continued on next page)

Example 9–7 (Cont.) C Program Using Date-Time Arithmetic

```
    printf("%20s  WEEKLY PAYROLL ( %s - %s)\n\n"," ",start_date,end_date);
    printf("Emp Id     Last Name %11s Hours %14s Rate %9s Wages\n\n",
           " "," "," ");

/* Get data for employees until end of file. */

    while (sql_return_status == 0)
    {
        fetch_weekly_hours(&sql_return_status,emp_id,last_name,&hr_rate,
                           hours_worked,&hours,&min,&sec);  ❸

        if (sql_return_status != 0 && sql_return_status != 100)
        {
            sql_signal();
            rollback_transaction(&sql_return_status);
        }
        else
            if (sql_return_status != 100)
            {
                if (hours < 40)
                    wages = (float)hours * hr_rate +
                            (float)min/60 * hr_rate +
                            (float)sec/3600 * hr_rate;
                else
                    wages = 40.00 * hr_rate +
                            (float)(hours - 40) * hr_rate * 1.5 +
                            (float)min/60 * hr_rate * 1.5 +
                            (float)sec/3600 * hr_rate * 1.5;

                printf("%s %25s %10s %15.2f %15.2f\n",
                        emp_id,last_name,hours_worked,hr_rate,wages);
            }
    }
    rollback_transaction(&sql_return_status);
    if (sql_return_status < 0)
    {
        sql_signal();
        rollback_transaction(&sql_return_status);
    }
}
```

The following callout descriptions are keyed to numbered items in Example 9–7:

❶ Declares host character variables with length = length + 1 to accommodate the null terminator.

❷ Declares host variable sec as float, corresponding to the module language parameter REAL, because extracting the SECOND field from an interval results in the SQL data type INTEGER(2), which is not supported by C. The variable hr_rate is declared as type float for the same reason.

❸ Fetches the data for each employee until end of stream (SQLCODE = 100) or an unexpected error occurs.

If you compile, link, and run this program, the program displays the following report:

```
Enter START DATE (YYYY-MM-DD):1995-07-15
Enter END DATE   (YYYY-MM-DD):1995-07-19

                    WEEKLY PAYROLL ( 1995-07-15 - 1995-07-19)

Emp Id     Last Name            Hours               Rate          Wages

00415      Mistretta            51:18:16.7          12.50         711.96
00416      Ames                 48:34:16.7          12.50         660.71
```

9.3 Improving Portability When Using Date-Time Data Types

If your applications may need to work with more than one SQL product or you want applications that are compliant with the SQL standard, consider the following guidelines:

- Use the EXTRACT function to retrieve separate fields from date-time data types. For example:

```
SELECT  EXTRACT(YEAR FROM x), EXTRACT(MONTH FROM x)
        INTO    :years, :months
        FROM  DATE_TAB;
```

- Use the CAST function to store separate fields into intervals or evaluate an expression by using separate fields in intervals. For example:

```
UPDATE DATE_TAB
       SET  I_COL = CAST(:hours AS INTERVAL HOUR) +
                    CAST(:minutes AS INTERVAL MINUTE) +
                    CAST(:seconds AS INTERVAL SECOND);

SELECT ID, I_COL FROM   DATE_TAB
       WHERE  I_COL > CAST(:hours AS INTERVAL HOUR) +
                      CAST(:minutes AS INTERVAL MINUTE) +
                      CAST(:seconds AS INTERVAL SECOND);
```

- The INSERT statement accepts expressions in the VALUES clause. Therefore, you can insert host data directly into tables, as follows:

```
INSERT INTO x (LAST_NAME, AGE)
       VALUES  (:last_name, CAST(:age AS INTERVAL YEAR));
```

- Retrieve the date-time fields as text. For example:

```
SELECT CAST(BIRTHDAY AS VARCHAR(30))
       FROM  EMPLOYEES;
```

  The resulting text strings conform to the layout defined for intervals and date-time data types. The leading field for intervals occupies the number of digits specified by the interval leading field precision. There is always provision for a sign character (either a space for 0 or positive intervals, or a minus sign for negative intervals).

9.4 Converting Applications and Databases

You can convert applications and databases to use the date-time data types as follows:

- To change the data type from DATE VMS to DATE ANSI, TIME, or TIMESTAMP, use the ALTER DOMAIN statement. Conversion of data stored in older versions of the row is performed automatically by Oracle Rdb.
- Because assignment of DATE ANSI literal values does not function in exactly the same way as assignment of DATE VMS literal values, you must specify literal values using the ANSI date-time formats, or use the CAST function to convert from DATE VMS to the ANSI data types. For example:

```
CAST(CAST(:host_char AS DATE VMS) AS TIMESTAMP)
```

- If you use the LIKE, CONTAINS, or STARTING WITH predicates to search a column defined as a ANSI date-time data type, the internal representation that Oracle Rdb uses is identical to the following literal syntax:

```
'199501'
```

  Because these functions implicitly convert to text, comparisons may not work the same as with DATE VMS data types.

9.5 Handling DATE VMS Data Types in Applications

When you use the DATE VMS data type in applications, you can assign character strings to it as you can in interactive SQL. However, you should note the following:

- If you assign a character string literal to a DATE VMS column, use one of the following formats:
  - dd-mmm-yy hh:mm:ss:cc

    The following example inserts a character string value into a DATE VMS column:

    ```
    EXEC SQL INSERT INTO TIME_SHEET (START_TIME)
                        VALUES ('12-NOV-1979 12:34:56.78');
    ```

  - yyyymmddhhmmsscc

    If you use this format, use the CAST function to convert it to DATE VMS, as shown in the following example:

    ```
    EXEC SQL INSERT INTO TIME_SHEET (START_TIME)
                        VALUES (CAST ('1979111212345678' AS CHAR(16)));
    ```

- If you use a character string variable or parameter, use the format: yyyymmddhhmmsscc

  The following example assigns a character string to a host language variable and uses the variable to insert the data into a DATE VMS column:

  ```
  char st[17];
  strcpy(st, "1979111212345678");

  EXEC SQL INSERT INTO TIME_SHEET (START_TIME)
                      VALUES (:st);
  ```

The following excerpt from a precompiled C program shows how Oracle Rdb lets you insert a character string (in the 16-character format) directly into the colum TBL_DATE_VMS defined as DATE VMS. Oracle Rdb automatically performs an internal conversion of the character string to the DATE VMS format. The excerpt also shows the reverse operation—inserting the TBL_ DATE_VMS column of data type DATE VMS directly into the 16-character string format.

```
main()
{
    char    string_date[17];

    strcpy(string_date,"1989111212345678");

    EXEC SQL INSERT INTO ALL_DATE_TABLE (TBL_DATE_VMS)
                        VALUES  (:string_date);
```

```
    strcpy(string_date,"               ");
    EXEC SQL SELECT TBL_DATE_VMS INTO :string_date
                    FROM ALL_DATE_TABLE;
    printf("Date VMS (16-character string format): %s\n",string_date);
}
```

9.5.1 Using DATE VMS with Applications Specific to OpenVMS

Sometimes, date-time data is derived from other applications and sources, or is directed to other layered products and may not support the ANSI/ISO date-time data types. In these situations, use the CAST function to convert the data to DATE VMS format. Applications specific to OpenVMS can then use the run-time library functions to format international dates and times.

Oracle Rdb guarantees the output from the following CAST function to be a date format that is compatible with OpenVMS, even for future versions of Oracle Rdb. DATE VMS is an Oracle Rdb extension to the ANSI/ISO SQL standard.

```
SELECT CAST(birthday AS DATE VMS), LAST_NAME, FIRST_NAME
       FROM    ...;
```

9.5.2 Porting Applications That Contain DATE VMS Data Types

Applications that run on OpenVMS with data stored in columns of data type DATE VMS often call OpenVMS Run-Time Library routines to format and convert data between their character string and binary representations.

Because these services are not provided with other operating systems, you must change applications that run on other operating systems to handle date columns properly. The following describes two possible methods:

- Convert your database to one of the ANSI date-time types. These types are slightly different from DATE VMS. Although this is the best method, you may not choose to do so because of the impact on other applications.
- Use the character representation as the means of sending and receiving dates at run time rather than using the binary representation.

Host programs often define 8-byte character variables to return the binary date. SQL treats assignments between 8-byte character variables and DATE VMS columns as an assignment without conversion. The binary data is placed in the variable. Because character variables that are 16 bytes (char x[17] in C programs) are long enough to hold the character representation, SQL assigns the converted value to the variable.

Using the 16 character format (YYYYNNDDHHMMSSCC), code the date January 1, 2001 at 7:25 PM as follows:

```
DATE VMS '2001010119250000'
```

Examples in this section demonstrate the following:

- Simple storage and retrieval of DATE VMS data
- Operating on the character representation

The following program shows how to use user-written functions to replace OpenVMS system service calls that translate the DATE VMS format into the DATE ANSI format. The variables that hold the result are 16 bytes instead of eight. Note that if you remove calls to the system service, you no longer need the OpenVMS descriptor.

```
simple_date () {

  /* Note that because these variables are treated as character strings,
   * their size must include the null terminator for C.
   */
  char in_date[17];
  char ret_date[17];
  char string_date[MAX_DATE_SIZE];
  long SQLCODE;

  /* Ask the user for the date. */
  ReadInputDate(string_date);

  /* Convert the date to SQL character format. */
  ToSQLDate(string_date,in_date);

  EXEC SQL INSERT INTO SALARY_HISTORY (EMPLOYEE_ID,SALARY_START)
              VALUES ('09999', :in_date);
  check_sqlcode( &SQLCODE );

  EXEC SQL SELECT SALARY_START INTO :ret_date
              FROM SALARY_HISTORY WHERE EMPLOYEE_ID = '09999';
  check_sqlcode( &SQLCODE );

  /* Convert the date back to something that the user can read.*/
  FromSQLDate(string_date,ret_date);

  printf("Date: %s\n",string_date);

  }
```

The application can write the routines ToSQLDate and FromSQLDate for the specific operating system. Depending on how you want the application to present dates to the user, you can make the date format either operating system specific or consistent across all operating systems.

The following examples show how to change the size of the variables to force SQL treat the variables as different data types. Because they are SQL module procedures, you must change the data type of the parameter from DATE to CHAR.

As shown in the following example, the original code uses 8-byte variables to hold the results:

```
PROCEDURE INSERT_SAL_HIST
   (SQLCODE,
    :IN_DATE DATE VMS);

    INSERT INTO SALARY_HISTORY (EMPLOYEE_ID,SALARY_START)
                VALUES ('09999', :IN_DATE);
```

Change the size of the variables to be 16 bytes, as shown in the following example:

```
PROCEDURE INSERT_SAL_HIST
   (SQLCODE,
    :IN_DATE CHAR(16) );

    INSERT INTO SALARY_HISTORY (EMPLOYEE_ID,SALARY_START)
                VALUES ('09999', :IN_DATE);
```

After you convert the date format to a character string, you can operate on this string just as you would any other character string. You can use the CAST function to convert the data while it is in the database.

For example, to retrieve only the date portion of the SALARY_START column use the following statement:

```
SELECT SUBSTRING( CAST( SALARY_START AS CHAR(16) ) FROM 1 FOR 8 )
       FROM SALARY_HISTORY;
```

In this example, the CAST function converts the DATE VMS data type to the character representation. Then, the SUBSTRING function works on the resulting character string.

Note that you can also construct a date using the CAST function and concatenation. The following example builds the character representation with multiple strings concatenated together:

```
INSERT INTO SALARY_HISTORY (SALARY_START)
        VALUES (CAST( :yr || :mm || :yy || '00000000' AS DATE VMS));
```

The string of zeros representing the hours, minutes, seconds, and hundredths is required for proper conversion.

For more examples of using the CAST function to operate on DATE VMS data types, see the programs whose names begin with "sql_load" in the samples directory.

9.6 Using Date-Time Data Types with Dynamic SQL

As you program with dynamic SQL, keep in mind the following points about using date-time data types with dynamic SQL:

- Dynamic SQL inherits the DATE ANSI data type setting from the module in which it is prepared and executed.
- You must use the SQLDA2 structure with the date-time data types.
- SQL processes expressions of the form CAST(? AS INTEGER) differently than other expressions. For more information, see Section 9.6.1.

9.6.1 Using CAST with Parameter Markers

SQL processes expressions using CAST and parameter markers differently than other expressions. SQL cannot derive the data type from the context for expressions of this type. Thus, SQL interprets the parameter marker (?) as having the same data type as the one specified by the CAST function, in this case INTEGER. This means you may need to use two CAST functions, as shown in the following example:

```
CAST(CAST(? AS CHAR(10)) AS INTEGER)
```

The sql_dynamic.c program in the samples directory shows how to define date-time data types and the SQLDA2 structure in dynamic SQL. It demonstrates how to convert date-time data types to character strings before processing with the PREPARE and EXECUTE dynamic statements, thereby eliminating the need to use the two CAST functions with date-time data types.

Example 9–8 shows a portion of an interactive dialogue using the sql_dynamic program located in the samples directory. The example demonstrates that you must cast the parameter marker (?) in the SELECT statement to the proper date-time data type, in this case interval HOUR, before SQL can successfully perform the Boolean greater than (>) operation.

Example 9–8 Casting Parameter Markers in Dynamic SQL Programs

```
DynamicSQL>  SELECT * FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS
cont>          WHERE HOURS_WORKED > ?;

The SQL statement to be executed dynamically is:
  SELECT * FROM ADMINISTRATION.ACCOUNTING.DAILY_HOURS WHERE HOURS_WORKED > ?;

Enter value for parameter 'HOURS_WORKED' Interval Hour (2) To Second (2): 10

Error -304 returned from open_cursor
```

(continued on next page)

Example 9–8 (Cont.) Casting Parameter Markers in Dynamic SQL Programs

```
Error message:
%RDB-E-ARITH_EXCEPT, truncation of a numeric value at runtime
-SQL-F-DATCONERR, Data conversion error for string '10'
-COSI-F-IVTIME, invalid date or time

Error returned from sql_dynamic()

Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> literal>(<)<CR> or <CTL-Z> to exit>:

DynamicSQL> SELECT * FROM ACCOUNTING.DAILY_HOURS
cont>         WHERE HOURS_WORKED > CAST(? AS INTERVAL HOUR);

The SQL statement to be executed dynamically is:
       SELECT * FROM  ADMINISTRATION.ACCOUNTING.DAILY_HOURS  WHERE
HOURS_WORKED > CAST(? AS INTERVAL HOUR);

Enter value for parameter ''Interval Hour (2): 10
---------------------------------------
Field EMPLOYEE_ID:00415
Field START_TIME:[Timestamp (2)] 1995-07-17 07:15:06.62
Field END_TIME:[Timestamp (2)] 1995-07-17 17:45:24.19
Field HOURS_WORKED:[Interval Hour (2) To Second (2)]  10:30:17.57

No more records found.
   .
   .
   .
```

9.6.2 Passing Dates as Text Strings to Dynamic SQL Statements

Digital UNIX

Dynamic SQL contains a restriction when date strings of the format YYYYNNDDHHMMSSCC are stored into a DATE VMS column on Digital UNIX. When a user passes a date as a text string to a dynamic SQL EXECUTE statement (SQL/Services always passes dates as text), the string is improperly converted to a binary date. Do not use a statement such as the following:

```
INSERT INTO  TBL (DATE_COL) VALUES ( ? )
```

Instead, specifically query the database system for the date by using the following statement:

```
INSERT INTO TBL (DATE_COL) VALUES  (CAST( ? AS CHAR(16) ) )
```

This statement is also more efficient than the previous. This restriction may be lifted in a future release. ♦

Part III

Run-Time Processing

This part discusses how to handle:

- Completion conditions, expected errors, and unexpected errors
- Run-time processing of SQL statements through dynamic SQL

10
Handling Run-Time Errors

This chapter describes how to detect run-time errors, retrieve accompanying error messages, and either recover from the error or roll back the transaction. The chapter describes the following:

- The options SQL provides for error handling
- Monitoring the execution of SQL statements for errors
- Displaying error messages
- Handling errors associated with database integrity and setting constraint evaluation time
- Handling lock conflict and deadlock errors
- Handling errors caused by failure to attach to a database or start a transaction
- Improving program portability when handling errors

Reference Reading

This chapter describes only errors returned from SQL and the underlying database system. Refer to your programming language documentation for information on handling host language or system run-time errors.

10.1 Overview of SQL Error Handling

All programs are subject to run-time errors, both expected and unexpected. Table 10–1 describes these two types of errors.

Table 10–1 Types of Run-Time Errors

Error Type	Description
Expected	Expected errors fall into two categories: • Completion conditions, such as the "no next row" ("no data") condition, that your program checks to determine when to continue or stop execution of a loop. These conditions are not errors in the true sense, but conditions that control a normal execution cycle. • Exception conditions, such as deadlock, lock conflict, database integrity, and duplicate value errors, that may occur from time to time, but are unpredictable. Often, you want your program to recover and continue execution when one of these errors occurs.
Unexpected	Errors from which your program typically cannot recover, for example: arithmetic exceptions, declaring the wrong database, using obsolete metadata (data definitions), or a corrupt database. When your program encounters an unexpected error, it needs to handle it by displaying information about the error, rolling back any changes, and stopping execution.

SQL does not call system routines to stop your program when an exception condition or error returns. Therefore, your program may continue in ways you do not expect, often encountering additional errors that are caused by the first error. For this reason, you should always check for completion and exception conditions on SQL statements.

To handle a completion or exception condition or error, you must:

- Declare a parameter to store a value that represents the execution status of an SQL statement
- Check the value contained by that parameter after execution of each SQL statement
- Specify actions to take based on that value

  For example, roll back the transaction or exit the program.

Table 10–2 summarizes the techniques that SQL provides for handling errors.

Table 10–2 SQL Techniques for Handling Errors

Error-Handling Technique	Standards Compliant?	Description
SQLSTATE parameter	Yes	Stores a 5-character string value representing the execution status of an SQL statement. The ANSI/ISO SQL standard requires an SQLSTATE or an SQLCODE (or both) declaration in your program. The SQL interface to Oracle Rdb enforces this rule. SQLSTATE is the recommended status parameter. Use SQLSTATE in all new application development.
SQLCODE parameter	Yes[1]	Stores an integer value representing the execution status of SQL statements. The ANSI/ISO SQL standard requires an SQLCODE or an SQLSTATE (or both) declaration in your program. The SQL interface to Oracle Rdb enforces this rule.
GET DIAGNOSTICS statement	Yes	Extracts diagnostic information about the execution of the previous statement in a compound statement.
SIGNAL statement	Yes	Provides information about exception conditions in compound statements.
WHENEVER statement	Yes[1]	Provides error handling for all SQL statements that follow it. Its use is restricted to embedded SQL.
RDB$MESSAGE_ VECTOR address array	No	Stores information about the execution status of SQL statements. The ANSI/ISO SQL standard does not include the RDB$MESSAGE_VECTOR array. In addition, because this array is an Oracle Rdb internal data structure, the values returned may change from version to version of Oracle Rdb. The RDB$MESSAGE_VECTOR array provides more detail about the type of error than the SQLCODE or SQLSTATE parameters. Consider using it *only* if you cannot retrieve needed information by the using the recommended methods.
sql_get_message_ vector routine	No	Retrieves information from the RDB$MESSAGE_VECTOR array about the status of the last SQL statement.
sql_signal routine	No	Signals a condition using error information returned through the RDB$MESSAGE_VECTOR array. The sql_ signal routine, depending upon your program condition handler, can display messages for primary and follow-on errors and continue or stop the program.

[1] SQLCODE complies with the SQL89 ANSI/ISO SQL standard but has been deprecated in the SQL92 ANSI/ISO SQL standard. The WHENEVER statement uses SQLCODE.

(continued on next page)

Table 10–2 (Cont.) SQL Techniques for Handling Errors

Error-Handling Technique	Standards Compliant?	Description
sql_get_error_text routine	No	Returns error text to your program for processing. It uses error information returned through the RDB$MESSAGE_VECTOR array.
SQL error-handling routines	Yes	Three routines let you use user-written error handlers in SQL programs: • sql_register_error_handler (establish error handler) • sql_get_error_handler (invoke error handler) • sql_deregister_error_handler (remove error handler) These routines let you set up any number of error-handling routines that can override or enhance the error-handling options provided by SQL.

You can inspect the sample programs in the samples directory to find different techniques to handle errors. The sql_terminate program contains varied error-handling techniques and more explanatory comments than other programs.

10.2 Monitoring Execution of SQL Statements for Errors

This section describes how to use the SQLSTATE status parameter, the SQLCODE status parameter, the WHENEVER statement for precompiled SQL, the RDB$MESSAGE_VECTOR array, and SQL error-handling routines to detect errors, including completion and exception conditions.

The following list summarizes some guidelines that apply no matter which status parameter you use to monitor for errors:

- The ANSI/ISO SQL standard requires your program to declare SQLSTATE, SQLCODE, or both. The SQL interface to Oracle Rdb enforces this rule. If you are writing a program that adheres to the ANSI/ISO SQL standard, your program should check the value stored in the SQLSTATE or SQLCODE parameter after execution of each simple and compound statement.

  Because statements that are not executable do not return values to status parameters, do not check the value of the status parameter after non-executable statements. For example, do not check the value of the status parameter after a DECLARE CURSOR statement.

- Often, error handling routines roll back transactions. However, the roll back operation succeeds only if a transaction has been started, which may not always be true. In addition, either success or failure of the ROLLBACK statement affects the value of SQLSTATE or SQLCODE. Therefore, your program should display the messages associated with the original error before it executes any further SQL statements. See Section 10.6 for additional information on handling errors associated with rolling back.

Reference Reading

The *Oracle Rdb7 SQL Reference Manual* contains an appendix that describes the SQL Communications Area (SQLCA), SQLSTATE and SQLCODE status parameters, and the RDB$MESSAGE_VECTOR array.

In addition, the reference manual chapter on SQL statements contains a section on the WHENEVER statement.

10.2.1 Using SQLSTATE

The SQLSTATE status parameter reflects the execution status of each SQL statement. The SQLSTATE status parameter is compliant with the ANSI/ISO SQL standard. It attempts to make status codes more portable among SQL implementations. The SQLSTATE status parameter provides many more standard-defined status values than SQLCODE, while still leaving room for implementor-defined status values.

New applications should use SQLSTATE rather than SQLCODE. You must set the dialect to SQL92 to use SQLSTATE to monitor completion codes, but errors are visible in SQLSTATE no matter what the dialect. Your program should check the value stored in the SQLSTATE parameter after execution of each simple and compound statement.

Declare the SQLSTATE status parameter as a 5-character string. The first two characters hold a class code; the last three characters contain a subclass code. For example, a status value of five zeros (meaning a 00 class code and a 000 subclass code) identifies the successful completion of an SQL statement. The status parameter can consist of uppercase letters A through Z and digits 0 through 9. Status parameter values fall into two categories:

- Standards-defined status parameter values

  The letters A through H and the digits 0 through 4 start status values defined by the ANSI/ISO SQL standard.

- Implementor-defined status parameter values

The letters I through Z and the digits 5 through 9 begin status values that are implementor defined. Oracle Rdb and SQL use codes beginning with R and S.

An appendix in the *Oracle Rdb7 SQL Reference Manual* lists SQLSTATE status parameter values and their corresponding exception conditions. At a minimum, your program should check for the following conditions:

- If the SQLSTATE value equals '00000', the SQL statement executed successfully.
- If the SQLSTATE value equals '01000', the SQL statement generated a warning.
- If the SQLSTATE value equals '02000', the SQL statement cannot return any more data.
- If the SQLSTATE value begins with a value other than '00', '01', or '02', the SQL statement encountered an error. An error indicates a nonfatal exception condition that must be handled or corrected.

In most programs, you should check for and handle specific conditions, such as deadlock, lock conflict, attach failure, and, on rollback, no active transaction to roll back.

For compound statements, you can use the GET DIAGNOSTICS statement to return the value of SQLSTATE for the previous SQL statement. Refer to Section 12.9 for more information.

10.2.2 Using SQLCODE

The SQLCODE status parameter reflects the execution status of each SQL statement. The ANSI/ISO SQL standard defines only three values for SQLCODE: 0 (success), 100 (no data), and negative values (error). The SQL interface to Oracle Rdb provides SQLCODE values in addition to these. See the *Oracle Rdb7 SQL Reference Manual* for a complete list of SQLCODE values.

Your program should check the value of SQLCODE after execution of each SQL statement. For compound statements, you can use the GET DIAGNOSTICS statement to return the value of SQLCODE for the previous SQL statement. Refer to Section 12.9 for more information.

You can declare the SQLCODE parameter either independently or as part of a structure called the SQLCA. When you declare SQLCODE independently, you declare it as a signed longword. (See Section 8.12.2 for information about declaring SQLCODE in C host language programs that call SQL modules and in precompiled C programs.)

The **SQLCA (SQL Communications Area)** is a data structure that provides information about the execution of SQL statements to application programs. The SQLCA shows whether a statement was successful. For some errors, it also shows the particular error when a statement is not successful. The SQLCA is not part of the ANSI/ISO SQL standard.

The following fields of the SQLCA are of interest to SQL users in handling errors:

- The SQLCODE field
- The third element of an array named SQLERRD

  This array element stores a count of the rows processed by an SQL statement or, for a FETCH statement, a count of the current row position within a cursor. Note that you cannot declare the SQLERRD array independently of the entire SQLCA structure.

Your options for declaring SQLCODE as part of the SQLCA depend on whether you are using the SQL precompiler or the SQL module processor.

- In precompiled programs, specify the INCLUDE SQLCA statement. You are limited to one independent SQLCODE declaration or one INCLUDE SQLCA statement per source file for those languages where SQL does not support block structure.
- If you are using the SQL module language, declare a field for SQLCODE either independently or within a host language structure that represents the entire SQLCA. (The appendix on the SQLCA in the *Oracle Rdb7 SQL Reference Manual* has examples that guide you in explicitly defining a structure for the SQLCA.)

  To update all the fields in the SQLCA when an SQL statement is processed, specify the record name of your host language structure as a parameter in your call to a procedure in an SQL module. The record name you specify in your call corresponds to the keyword SQLCA, which you specify in the parameter list of the procedure being called in the SQL module. You may prefer to specify in some calls only the field that represents SQLCODE if you do not want to use the row count value in the SQLERRD array. In this case, specify SQLCODE rather than SQLCA as the corresponding keyword in the parameter list of the procedure you are calling.

  When a program calls procedures in an SQL module, you can choose a name other than SQLCODE for the host language declaration of the SQL status parameter. However, naming your status parameter SQLCODE makes its function clear to anyone else who might look at your source code. If you decide to work with different SQL status parameter declarations for different calls (not possible in precompiled programs for languages

where SQL does not support block structure), you must give each status parameter a different name. In this case, appending a digit or the name of an associated procedure to SQLCODE might be a reasonable convention to follow.

See Section 8.12.2 for information about declaring SQLCA in C host language programs that call SQL modules and in precompiled C programs.)

The appendix on the SQLCA in the *Oracle Rdb7 SQL Reference Manual* lists all possible numeric and literal values the SQLCODE field can contain and describes what each of these numeric and literal values means. In general, the values have the following meanings:

- A numeric value of 0 or a literal value of SQLCODE_SUCCESS indicates that a statement executed successfully.
- Positive integers represent completion codes—the statement executed to completion but something out of the ordinary occurred. The most important literal of these is the numeric value 100 or its literal value equivalent SQLCODE_EOS, which indicates that no row was found.
- Negative integer values or their literal value equivalents represent errors that prevent the statement from executing to completion.

Your program typically checks for the following kinds of conditions and operations. These may be expressed using a variety of host language statements.

- As long as a particular SQL statement executes successfully (SQLCODE = 0 or SQLCODE_SUCCESS), execute a set of statements to process data.
- Until no row is retrieved by a particular SQL statement (SQLCODE = 100 or SQLCODE_EOS), execute a set of statements to process data.
- If any error is encountered during execution of a particular SQL statement (SQLCODE is less than 0), execute a set of error-handling statements.

  A host language statement may include several of these conditions, each of which is associated with a different kind of expected error and program recovery procedure, and then end with an "else" condition associated with a set of error-handling statements for unexpected errors.

Example 10–1 to Example 10–3 show several ways of using SQLCODE to handle errors.

The syntactic host language construct implied by the examples is a nested IF statement block that is common to most languages. The examples do not attempt to adhere to format and syntax requirements for any host language IF statements. Your particular programming language may also offer statements,

such as EVALUATE (COBOL) or SWITCH (C), that implement conditional control of program flow more efficiently than an IF statement.

Example 10–1 shows how to to monitor the value of SQLCODE to determine when to end a FETCH loop. It uses a host language and calls to an SQL module.

Example 10–1 Monitoring SQLCODE in SQL Module Language

```
Execute the following statements until SQLCODE equals 100:
     Host language statements to initialize indicator parameters
     Call to name-of-fetch-procedure passing SQLCODE and
        a list of main parameters and indicator parameters
        as call parameters
     If SQLCODE is less than 0
        then branch to error-handling-section
     else
        Host language statements to evaluate indicator array
           items and manipulate fields in record

End of execution block
```

The error-handling-section to which Example 10–1 refers should evaluate SQLCODE to determine which error occurred and then execute the appropriate actions. If the error is one from which the program cannot recover, the program performs cleanup operations such as logging the error to a file, displaying messages for the primary and all follow-on errors, rolling back a transaction if one is in progress, and stopping. Monitoring for negative values in the SQLCODE field or a failure status in your host language status field lets you determine when a FETCH statement generates a fatal error. At that point, your program should exit the loop.

If you monitor the status of SQL statements using only host language conditional statements, you must include such a conditional statement in your host language program following each SQL statement or, if you are using the SQL module processor, following each call to a procedure in an SQL module.

Example 10–2 illustrates the simple solution of "clean up and stop the program" no matter which fatal error occurs.

Example 10–2 Monitoring SQLCODE and Stopping on Error

```
Precompiled SQL statement or SQL module procedure call
IF SQLCODE is less than 0
   branch to error-handler-1
End of IF block
   .
   .
   .
error-handler-1
   Display SQLCODE value and error messages
   Roll back transaction
   IF SQLCODE is less than 0
        Display messages and continue
   End of IF block
   Close any files
   Stop program execution
```

Many database conditions do not require your program to stop. Often you can simply restart the transaction or take other corrective action. For example, if an attach failed because the database does not exist, you might be able to ask the user to enter a new database name. Example 10–3 performs more than one evaluation of SQLCODE, and then either executes recovery actions or stops the program as a result of those evaluations.

Example 10–3 Using SQLCODE Values to Take Recovery Action

```
Precompiled SQL statement or SQL module procedure call
IF SQLCODE is less than 0
   branch to error-handler-1
End of IF block
   .
   .
   .
error-handler-1
   IF SQLCODE equals -1003 or -913 (lock conflict or deadlock error)
      Display or log messages about error
      Execute actions for locking problems
   else
      IF SQLCODE equals -803 (duplicate value in unique index error)
         Display or log messages about error
         Execute actions for duplicate value violations
   else
      IF SQLCODE equals -1002 or -1001 (violation of constraint)
         Display or log messages about error
         Execute actions for validation and integrity violations
```

(continued on next page)

Example 10–3 (Cont.) Using SQLCODE Values to Take Recovery Action

```
else
   Display or log message about unexpected error
   Execute orderly termination of transaction and program
End of IF block
```

See Section 10.4 and Section 10.5 for more information about errors from which you may want your program to recover.

10.2.3 Using the WHENEVER Statement

In precompiled programs, you can specify SQL WHENEVER statements to provide error handling for all SQL statements that follow each WHENEVER statement. You cannot use the WHENEVER statement with SQL module language. The WHENEVER statement handles categories of errors, not individual SQL statements, as shown in the following example:

```
EXEC SQL WHENEVER SQLERROR GOTO label-of-section-handling-errors

EXEC SQL WHENEVER NOTFOUND GOTO label-of-section-handling-condition

EXEC SQL WHENEVER SQLWARNING GOTO label-of-section-handling-warnings
```

When the SQL precompiler processes a WHENEVER statement, it inserts the same host language conditional statement after every SQL statement it processes until it comes to the end of your source file or encounters another WHENEVER statement of the same type. For example, for a WHENEVER SQLERROR statement, the precompiler inserts after each executable SQL statement the host language equivalent of the following IF statement:

```
If SQLCODE is less than 0, GOTO same-section-for-handling-errors
```

Therefore, you must specify your own host language conditional statements for error or condition handling that is statement-specific. You usually want conditional error control for exception conditions and you may want to use it for other kinds of errors as well. In particular, you should provide statement-specific error handling for SELECT and FETCH statements and for statements that might cause the database system to evaluate constraints. (See Section 10.4 for information about how to handle constraint violations.)

In the SQL WHENEVER statement, you specify the name of a section of a precompiled program. That section handles a particular set of errors.

The WHENEVER statement can detect three types of completion or exception conditions or errors:

- No next row (NOT FOUND)

This is equivalent to the condition SQLCODE = 100, SQLCODE_EOS, or SQLSTATE = '02000', also called "no data".

- Warnings (SQLWARNING)

  This is equivalent to SQLCODE containing a value that is greater than 0 or SQLSTATE containing a value that starts with '01'. However, it does not include the case where SQLCODE = 100 or SQLCODE_EOS, or where SQLSTATE = '02000'.

- Fatal errors (SQLERROR)

  This is equivalent to the condition SQLCODE containing a value that is less than 0, an SQLSTATE value beginning with a value other than '00', '01', or '02', and local host language methods for determining execution failure of statements.

In the following example, one WHENEVER SQLERROR statement handles errors for SQL statements A, B, and C. Another WHENEVER SQLERROR statement handles errors for SQL statements after statement-C. Each WHENEVER statement specifies the label of a section of the program that handles errors. Statements in those sections might display field values and messages on the terminal and then stop the program in an orderly manner, or they might execute recovery actions. When the program runs, if statement-D encounters an error, error-handler-2 takes action. If statement-B encounters the error, error-handler-1 takes action.

```
EXEC SQL WHENEVER SQLERROR GOTO label-of-error-handler-1

EXEC SQL statement-A
EXEC SQL statement-B
EXEC SQL statement-C

EXEC SQL WHENEVER SQLERROR GOTO label-of-error-handler-2

EXEC SQL statement-D
EXEC SQL statement-E
EXEC SQL statement-F
```

The WHENEVER statement is not an executable statement. The SQL statements whose executions are monitored by a WHENEVER statement are determined at precompile time, not at run time. Not-found condition, warning, or error detection by a WHENEVER statement applies to all SQL statements that follow that WHENEVER statement in source code until the next WHENEVER statement that monitors for the same kind of condition is encountered.

At run time, program flow might cause statement-E in the previous example to execute before statement-B. But errors for statement-B are always handled by error-handler-1 and errors for statement-E are always handled by error-handler-2.

If you want a host language conditional statement to handle an error for a particular SQL statement, but that SQL statement is already preceded by a WHENEVER statement handling the same kind of condition or error, you need to turn off the WHENEVER statement control for the SQL statement in question. Do this by specifying another WHENEVER statement with the action CONTINUE. For example, assume that you want an IF statement to monitor errors for SQL statement-C, but want to resume error monitoring with WHENEVER for SQL statements that occur after SQL statement-C:

```
EXEC SQL WHENEVER SQLERROR GOTO ERROR-HANDLER-1
EXEC SQL statement-A
EXEC SQL statement-B
EXEC SQL WHENEVER SQLERROR CONTINUE
EXEC SQL statement-C

IF SQLCODE < 0  . . .

EXEC SQL WHENEVER SQLERROR GOTO ERROR-HANDLER-1
```

If your program includes an SQL statement in the error handler named in the GOTO clause of a WHENEVER statement, that SQL statement is itself subject to error conditions. (Typically, this is a ROLLBACK statement.) In this case, you do not want errors for the SQL statement in the section handling errors to be monitored by the WHENEVER statement that specifies the section. An SQL statement error in that section would then cause infinite looping between the WHENEVER statement and the section that handles errors. Specify a WHENEVER statement with the action CONTINUE to prevent this sort of loop. For example:

```
EXEC SQL WHENEVER SQLERROR GOTO ERROR-HANDLER-1
   .
   .
   .
ERROR-HANDLER-1
   Display messages
   EXEC SQL WHENEVER SQLERROR CONTINUE
   EXEC SQL ROLLBACK
   If SQLCODE < 0...
```

You can use host language conditional statements to provide the same condition and error monitoring that WHENEVER statements provide. WHENEVER statements can monitor execution status of more than one SQL statement, but work independently of program flow at execution time. Host language conditional statements are executable and allow conditional branching for

error handling at run time. However, a host language conditional statement can monitor execution status of only one SQL statement, while WHENEVER statements can monitor many statements.

If you include WHENEVER statements in your source program, check the source lines before precompiling the program. This is particularly important if a program is long or you change an existing program to add or delete source lines. For each SQL statement (regardless of whether the program includes a host language conditional statement to monitor conditions or errors), ask yourself the following questions:

1. Does a WHENEVER SQLERROR statement occur before this SQL statement?
2. Does a WHENEVER SQLWARNING statement occur before this SQL statement?
3. Does a WHENEVER NOT FOUND statement occur before this SQL statement?

If the answer to any of these questions is yes, determine whether the actions in the section referred to in the GOTO clause are appropriate for the SQL statement under consideration. If the error-handling actions are not appropriate, specify another WHENEVER statement of the same type to prevent inappropriate error handling or to specify alternative error handling. Use the CONTINUE option for this WHENEVER statement to handle errors using host language statements.

To reestablish the error handling that was in effect before you started making the preceding changes, specify the original WHENEVER statement of that type again *after* the SQL statement whose error handling you have changed.

Before you precompile a program, also check the program section named in the WHENEVER statement to see whether the section includes any executable SQL statements. If it does, make sure that you have handled conditions or errors specifically for those SQL statements to prevent looping between the section and the WHENEVER statement that refers to it.

10.2.4 Using the sql_get_message_vector Routine and RDB$LU_STATUS

You can use the sql_get_message_vector routine to retrieve information from the RDB$MESSAGE_VECTOR array. Because the sql_get_message_vector routine is portable to other platforms, you can use it to simplify porting of applications from one Oracle Rdb platform to another. However, the routine is not compliant with the ANSI/ISO SQL standard.

The RDB$MESSAGE_VECTOR array provides information about the execution status of SQL statements. The second element of the RDB$MESSAGE_ VECTOR array, RDB$LU_STATUS, is a field whose function is comparable to the SQLCODE status parameter. You can use the sql_get_message_vector routine to monitor this field to determine success or failure of SQL statements.

Note

The RDB$MESSAGE_VECTOR array is an Oracle Rdb internal data structure. Because of this, the values returned may change from version to version. Consider using the RDB$LU_STATUS field *only* if you cannot retrieve needed information using the SQLCODE or SQLSTATE status parameters.

The ANSI/ISO SQL standard does not include the array RDB$MESSAGE_VECTOR or the field RDB$LU_STATUS.

The RDB$MESSAGE_VECTOR array lets you access messages for both the primary error and all follow-on errors that may be returned by either the database system or other facilities on your system. These messages provide supplementary information that your program can display or process to more precisely identify the problem that caused the execution of an SQL statement to fail.

Using the sql_get_message_vector Routine

The sql_get_message_vector routine retrieves information from the array RDB$MESSAGE_VECTOR, without requiring you to explicitly declare the array in your program.

The routine takes two arguments: the address of the variable that receives the vector element and the index value of the vector element that you want returned. The following table shows the index values and the information contained in each vector element:

Index Value	Information Returned
1	Number of arguments in the vector
2	Primary status code of the last SQL statement
3	Number of FAO arguments to primary message
4–20	Return status for follow-on messages, if any

Oracle Rdb maps the indexes returned by sql_get_message_vector to fields in the RDB$MESSAGE_VECTOR array. It maps Index 2 to the RDB$LU_STATUS field. For more information on the mapping, see the *Oracle Rdb7 SQL Reference Manual*.

Example 10–4 shows an excerpt of a C program that calls an SQL module and uses the sql_get_message_vector to return the status of the SQL statement.

Example 10–4 Using the sql_get_message_vector Routine

```
   .
   .
   .
/* Error handler, using sql_get_message_vector. */

get_msgvec( )

{
int index;
int status_code;
int arg_cnt;

/* Declare the literal for constraint violation status. */
int RDB$_INTEG_FAIL;

/* Get the message vector argument count. */

index = 1;
sql_get_message_vector(&arg_cnt, index);

/* Get the status code. */

index = 2;
sql_get_message_vector(&status_code, index);

  if (status_code == RDB$_INTEG_FAIL)
     printf("Constraint violation. ");
     printf("You are trying to insert a department code\n");
     printf("that already exists in the table.");
     exit(1);

/* You can also check for follow-on arguments, if the arg_cnt is greater
*  than 1.
*/
}

main( )
{
   .
   .
   .
```

(continued on next page)

Example 10–4 (Cont.) Using the sql_get_message_vector Routine

```
  insert_data (&SQLCODE, department_code, department_name, manager_id);
  if (SQLCODE != 0)
   get_msgvec();
}
```

As Example 10–4 shows, when you check the value of Index 2, you check for symbolic code values rather than numeric values.

To declare prototypes for the sql_get_message_vector routine, as well as other SQL routines, in C programs, you can use the include file sql_rdb_headers.h. For more information, see Section 10.3.4.

Using RDB$LU_STATUS Field

Although it is not recommended, you can use the RDB$LU_STATUS field directly. To do so, declare the RDB$MESSAGE_VECTOR array.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, if you use the SQL precompiler and specify the INCLUDE SQLCA statement, the precompiler automatically declares the array RDB$MESSAGE_VECTOR. ♦

Digital UNIX

On Digital UNIX, you can use the `-s 'msgvec'` switch to specify that precompiler automatically declare the RDB$MESSAGE_VECTOR array. You can use this switch with language compilers that support the dollar sign ($) special character. ♦

You can explicitly declare the RDB$MESSAGE_VECTOR array in your program. The appendix on the SQLCA and the message vector in the *Oracle Rdb7 SQL Reference Manual* shows how you can declare the array RDB$MESSAGE_VECTOR.

Even if your program does not use SQLCODE or SQLSTATE, you must still declare one of them and pass the field as a parameter when you call a procedure in an SQL module.

To use the RDB$LU_STATUS field, you usually declare the values as symbolic codes. When you check the value of RDB$LU_STATUS after execution of an SQL statement, you check for symbolic code values rather than numbers. For example, you might check for the value RDB$_NO_DUP to monitor for constraint violations.

Your program compares the value in RDB$LU_STATUS to a symbolic error code that is associated with an error from the Oracle Rdb or SQL facilities. You must explicitly declare a symbolic error code for each error or condition you want to check. You declare the symbolic error codes as unsigned longwords and external values. Table 10–3 shows how to declare the symbolic error codes in the languages supported by the SQL precompiler.

Table 10–3 Declaring Symbolic Error Codes in Embedded Host Languages

Language	Declaration
Ada	---Declare the system package. with system; -- Declare the symbolic error code. lock_conflict : system.unsigned_longword := system.import_value("RDB$_LOCK_CONFLICT");
C	globalvalue RDB$_LOCK_CONFLICT;
COBOL	05 RDB$_LOCK_CONFLICT PIC S9(9) COMP VALUE IS EXTERNAL RDB$_LOCK_CONFLICT.
FORTRAN	INTEGER*4 RDB$_LOCK_CONFLICT EXTERNAL RDB$_LOCK_CONFLICT
Pascal	RDB$_LOCK_CONFLICT : [value,external]INTEGER;
PL/I	DECLARE RDB$_LOCK_CONFLICT GLOBALREF FIXED BINARY (31);

Depending on how you want to use symbolic error codes in your program, you can declare them as variables or constants. You can test them directly using host program language constructs.

If you want to check for all fatal errors, you can monitor the execution of SQL statements using the same status field you declare in your program to monitor the execution of host language statements. In this case, the error-handling section to which your program transfers control can execute one set of statements.

Monitoring the value of RDB$LU_STATUS is useful when you anticipate constraint violations and want to return information that is not available through the SQLCODE or SQLSTATE status parameters. Example 10–5 evaluates the RDB$LU_STATUS field and then executes recovery actions when it encounters a constraint violation or a duplicate value error.

Example 10–5 Using RDB$LU_STATUS to Trap Constraint Violations

```
Declaration of ret-status
Declaration of RDB$MESSAGE_VECTOR  ! The INCLUDE SQLCA statement can
                                   ! replace an explicit declaration
                                   ! of RDB$MESSAGE_VECTOR in
                                   ! precompiled programs.
Declaration of RDB$_NO_DUP
Declaration of RDB$_INTEG_FAIL
   .
   .
   .
Precompiled SQL statement (or a call to a procedure in an SQL module)
Assignment of RDB$LU_STATUS to ret-status
IF ret-status does not equal success
   Branch to error-handler-1
End of IF block
   .
   .
   .
error-handler-1
   Evaluate RDB$LU_STATUS
      When RDB$_NO_DUP
         Display or log messages about error
         Execute actions for duplicate value violations
      When RDB$_INTEG_FAIL
         Display or log messages about error
         Execute actions for validation and integrity violations
      When other values
         Display or log message about error
         Execute appropriate actions
   End of evaluate block
```

See Section 10.4 and Section 10.5 for more information about errors from which you may want your program to recover.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, you can call the OpenVMS Run-Time Library routine LIB$MATCH_COND to evaluate the returned value. The RDB$LU_STATUS field and the declared symbolic error codes are passed by reference as call parameters. Following the call, a host language conditional statement evaluates which, if any, symbolic error code matches the error code returned by execution of the SQL statement and then directs program flow accordingly.

Because the LIB$MATCH_COND routine masks the severity levels of the error codes, you do not have to be concerned that the severity level of a symbolic error code may change in a future software release.

The following example, an excerpt from a Pascal program, shows how to use LIB$MATCH_COND to evaluate the returned status value.

```
[EXTERNAL] FUNCTION LIB$MATCH_COND
(ret_status:INTEGER; sym_name:[LIST]INTEGER):INTEGER;EXTERNAL;

begin

  (* Use LIB$MATCH_COND to determine which error has occurred. *)

  error := LIB$MATCH_COND (RDB$MESSAGE_VECTOR[2],
                           RDB$_DEADLOCK,          {1}
                           RDB$_LOCK_CONFLICT,     {2}
                           RDB$_NO_DUP,            {3}
                           RDB$_NOT_VALID,         {4}
                           RDB$_INTEG_FAIL,        {5}
                           RDB$_STREAM_EOF,        {6}
                           RDB$_NO_RECORD          {7}
                           );
```

The OpenVMS system routines documentation describes the LIB$MATCH_ COND run-time library routine. ♦

Language-specific versions of the online program sql_terminate in the samples directory contain comments about declaring and using symbolic error codes for the RDB$LU_STATUS field. For more information about using your host language to monitor execution status of statements, see your host language documentation.

Note

Ada does not allow the dollar sign ($) in names. Substitute an underscore (_) for the dollar sign in any name or symbolic code starting with RDB$. For example, declare and refer to the RDB$MESSAGE_ VECTOR array as RDB_MESSAGE_VECTOR.

10.2.5 Using the SQL Error-Handling Routines

SQL provides three routines to make it easier to pass control back to the application when SQL encounters an error. These SQL routines let you set up any number of error-handling routines that override or enhance the error handling provided by SQL. In addition, with these SQL routines, you can pass context-specific data to your error-handling routines.

SQL provides the following routines for SQL precompiled and module language programs:

- sql_register_error_handler

Registers the address of the application's error-handling routine and the address of the context-specific data.

- sql_get_error_handler

  Gets the address of the currently registered error-handling routine and the address of the context-specific data.

- sql_deregister_error_handler

  Deregisters the current error-handling routine.

You use these SQL routines to register your own error handler, which can handle errors differently than SQL does. For example, you can create a routine to display a different message than that returned by SQL when you violate a constraint, and you can pass information from the main program to that routine.

Because the SQL routines can pass context-specific data from your main program to your error-handling routine, you can use a single, common error-handling routine to handle a variety of errors. The common error-handling routine evaluates the data passed to it and, based on the data, may print out an error message, roll back the transaction, or take some other action.

To use the SQL error-handling routines, you must:

- Call the sql_register_error_handler routine, passing the name of your error handler and the address of the context-specific data as arguments. The following example shows how to call the sql_register_error_handler routine in a C language program:

  ```
  sql_register_error_handler (appl_error_handler,&user_data)
  ```

  See the Calling the SQL Error-Handling Routines subsection (within this section) for information on how to call the sql_register_error_handler routine in other languages.

- Declare error-handling routines so that they pass four parameters. The first three parameters, RDB$MESSAGE_VECTOR, SQLCODE, and SQLSTATE, are passed by reference. The fourth parameter, the address of the context-specific data, is passed by value. The following example shows how to declare an error-handling routine in a C language program:

  ```
  static
    void appl_error_handler(
         RDB$MESSAGE_VECTOR *msgvec,
         int                *sqlcode,
         char               *sqlstate,
         void               *user_info)
  ```

See the Declaring User-Written Error-Handling Routines section (within this section) for information on how to declare error-handling routines in other languages.

When your application executes an SQL statement that returns an error, SQL executes the currently registered error-handling routine. The routine can display a message, take some action (including changing the value of SQLCODE), or exit the application.

Note

Do not signal an exception from within your error-handling routine because doing so creates an infinite loop. When you use the SQL error-handling routines, SQL intercepts any exception signals and calls the user-written error handler. The user-written error handler, in turn, signals the exception, which is intercepted by SQL. This cycle repeats in an infinite loop.

If the error-handling routine changes the value of SQLCODE to 0, indicating success, the program returns control to SQL.

If the error-handling routine does not change the value of SQLCODE to 0, or if no error-handling routines are registered, SQL evaluates any SQL WHENEVER statements contained in the program. The program then returns control to SQL.

If you do not want the program to pass control back to SQL, take the following steps in the error-handling routine:

1. Deregister the error-handling routine so that the subsequent ROLLBACK statement does not call the error-handling routine. The following example shows how to deregister the routine:

   ```
   sql_deregister_error_handler ( );
   ```

2. Roll back the transaction. The following statement rolls back a transaction in a precompiled C program:

   ```
   EXEC SQL ROLLBACK;
   ```

3. Exit the image. When you do so, the SQL and Oracle Rdb exit handlers execute. In a C program, use the following statement:

   ```
   exit;
   ```

The routine sql_get_error_handler facilitates the use of more than one error-handling routine in an application. For example, if your program contains a subroutine that uses a different error-handling routine than that of the main program, the subroutine calls sql_get_error_handler to get the address of the currently registered routine and the context-specific data. Then, the subroutine stores these addresses in variables and registers its own error-handling routine. Before it returns control to the main program, the subroutine registers the main program's routine and data again by using the values in the variables.

To declare prototypes for the SQL error handling routines, as well as other SQL routines, in C programs, you can use the include file sql_rdb_headers.h. For more information, see Section 10.3.4.

Example 10–6 shows how to use the SQL error-handling routines in a precompiled C program.

Example 10–6 Using SQL Error-Handling Routines

```
/* This program demonstrates the use of the SQL error-handling routines,
*  sql_register_error_handler, sql_deregister_error_handler, and
*  sql_get_error_handler.   Although the use of the sql_get_error_handler
*  routine is not necessary in this simple program, it is included here
*  to demonstrate how to use the routine and how to store the address of the
*  currently registered routine and the address of user data in variables.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sql_literals.h>
#include <sql_rdb_headers.h>

/* Definition of rdb$message_vector. */
typedef struct {
   int  RDB$LU_NUM_ARGUMENTS;
   int  RDB$LU_STATUS;
   int  RDB$LU_ARGUMENTS[18];
   } RDB$MESSAGE_VECTOR;

/* Definition of structure to hold user data. */

typedef struct {
   char sql_proc_name[31];
   char sql_col_value[31];
          } err_struct;
```

(continued on next page)

Example 10–6 (Cont.) Using SQL Error-Handling Routines

```
/* Error-handling routine for constraint violations.  This routine traps
*  constraint violations and prints out an error message.   */

static
  void dupl_error_handler(
       RDB$MESSAGE_VECTOR *msgvec,
       int                *sqlcode,
       char               *sqlstate,
       void               *user_info)

    {

     err_struct  *my_info;
     my_info = (err_struct *)user_info;

     if ((*sqlcode == SQLCODE_INTEG_FAIL) &&
        ((strcmp(my_info->sql_proc_name, "INSERT_JOBS")) == 0))
      {
      printf(" The Job Code %s is already in use.\n", my_info->sql_col_value);
      }

/* You can add more conditional statements to this error-handling procedure
*  to handle errors from several SQL statements.   */

    }

/* Error-handling routine for errors that occur when you start a transaction.
*  This routine prints out an error message.       */

static
  void txn_error_handler(
       RDB$MESSAGE_VECTOR *msgvec,
       int                *sqlcode,
       char               *sqlstate,
       void               *user_info1)
    {

     if ((*sqlcode == SQLCODE_DEADLOCK) || (*sqlcode == SQLCODE_BAD_TXN_STATE)
           || (*sqlcode == SQLCODE_LOCK_CONFLICT))

       printf("Unable to start a transaction. \n");

    }

main( )

{

/* Variables used by the main program. */

    void    (*rtn_ptr)();
    err_struct  *err_struct_ptr = 0;
    char    j_code[5];
    char    w_class[2];
    char    j_title[21];
    char    release_screen;
```

(continued on next page)

Example 10–6 (Cont.) Using SQL Error-Handling Routines

```
/* Define the SQLCA. */
    EXEC SQL INCLUDE SQLCA;

/* Initialize user-defined information. */
    err_struct err_s = {" ", " "};

/* Declare the database. */
    EXEC SQL DECLARE ALIAS FILENAME 'PERSONNEL';

/* Register the first error-handling routine. */
    sql_register_error_handler(txn_error_handler,0);

/* Store the address of the currently registered pointer in a variable. */
    sql_get_error_handler(&rtn_ptr, &err_struct_ptr);


    printf("Please enter the Job Code (or EXIT):\n");
    scanf(" %s", j_code);
    release_screen = getchar();

    while (((strcmp(j_code,"exit")) != 0) &&
         ((strcmp(j_code,"EXIT")) != 0))
     {

      printf("Enter the Wage Class:  ", w_class);
      scanf(" %s", w_class);
      release_screen = getchar();
        while  (((strcmp(w_class,"1")) !=0) &&
               ((strcmp(w_class,"2"))  !=0) &&
               ((strcmp(w_class,"3"))  !=0) &&
               ((strcmp(w_class,"4"))  !=0))
         {
          printf("Please enter one of the following values for Wage Class:\n");
          printf(" 1  2  3  4\n");
          scanf(" %s", w_class);
          release_screen = getchar();
         }

      printf("Please enter the Job Title: \n");
      scanf(" %s", j_title);
      release_screen = getchar();

/* Start a transaction. */

      EXEC SQL SET TRANSACTION READ WRITE NOWAIT
                  EVALUATING JOB_CODE_PRIMARY AT VERB TIME
                  RESERVING JOBS FOR EXCLUSIVE WRITE;
```

(continued on next page)

Example 10–6 (Cont.) Using SQL Error-Handling Routines

```
/* Register the second error-handling routine. */
      sql_register_error_handler(dupl_error_handler, &err_s);

/* Store information in a structure for use by the error-handling routine. */
      strcpy(err_s.sql_proc_name, "INSERT_JOBS");
      strcpy(err_s.sql_col_value, j_code);

      EXEC SQL INSERT INTO JOBS
                   (JOB_CODE, WAGE_CLASS, JOB_TITLE)
                   VALUES
                   (:j_code, :w_class, :j_title );

      if (SQLCA.SQLCODE == SQLCODE_SUCCESS)
        EXEC SQL COMMIT;
      else
        EXEC SQL ROLLBACK;


/* Deregister the error-handling routine. */
      sql_deregister_error_handler();

      printf("Please enter the Job Code (or EXIT):\n");
      scanf(" %s", j_code);
      release_screen = getchar();

/* Register the txn_error_handler routine again.  Use the address stored in
*   rtn_ptr.   */
      sql_register_error_handler(rtn_ptr, 0);

     }

    return;

}
```

If you invoke the sql_register_error_handler routine from a shareable image, you must link the shareable image with a jacket routine that calls the sql_ register_error_handler routine. The following example shows a C language jacket routine:

```
extern void call_sql_regis_err_hand ( void (*user_error_handler) (),
                                       context_block_t *context_block)
    {
      sql_register_error_handler( user_error_handler, context_block);
      return;
    }
```

When you build multiple shareable images, you must register the error handler for each image, although all images can share the same error handler.

Calling the SQL Error-Handling Routines

This section shows how to call the error-handling routines in host languages supported by SQL.

- Ada

  In Ada, you must declare the SQL error-handling routines as procedures before you call them. The following example shows how to declare the procedures:

```
procedure sql_deregister_error_handler; -- no arguments
pragma interface( external, sql_deregister_error_handler );
pragma IMPORT_PROCEDURE (
    INTERNAL => sql_deregister_error_handler );

procedure sql_get_error_handler(
    handler : out system.address;
    context : out system.address );
pragma interface( external, sql_get_error_handler );
pragma IMPORT_PROCEDURE (
    INTERNAL => sql_get_error_handler,
    PARAMETER_TYPES => (system.address, system.address),
    MECHANISM => (REFERENCE,REFERENCE) );

procedure sql_register_error_handler(
    handler : in system.address;
    context : in system.address );
pragma interface( external, sql_register_error_handler );
pragma IMPORT_PROCEDURE (
    INTERNAL => sql_register_error_handler,
    PARAMETER_TYPES => (system.address, system.address),
    MECHANISM => (VALUE,VALUE) );
```

  The following example shows how to call the procedures:

```
pkg_handler.sql_deregister_error_handler;

pkg_handler.sql_get_error_handler(rtn_ptr, foo_ptr);

pkg_handler.sql_register_error_handler(
  pkg_other.my_error_handler'ADDRESS,
  foo'ADDRESS );
```

- BASIC

```
EXTERNAL LONG FUNCTION BASIC_ERRHND

CALL sql_deregister_error_handler
CALL sql_get_error_handler BY REF (ERR_HAND_ADDR, ERR_USER_CONTEXT))
CALL sql_register_error_handler &
   (LOC(BASIC_ERRHND) BY VALUE, REGISTERED_CONTEXT BY REF)
```

  BASIC is supported by SQL module language only.

- C

```
sql_deregister_error_handler();
sql_get_error_handler(&appl_error_handler_ptr,&user_data_ptr);
sql_register_error_handler(&appl_error_handler,&user_data);
```

- COBOL

```
01 appl_error_handler USAGE POINTER VALUE EXTERNAL MY_ERROR_HANDLER.
01 appl_error_handler_ptr USAGE POINTER VALUE 1.
01 user_data_ptr USAGE POINTER VALUE 1.

CALL "sql_deregister_error_handler".
CALL "sql_get_error_handler" USING BY REFERENCE appl_error_handler_ptr,
                                   BY REFERENCE user_data_ptr.
CALL "sql_register_error_handler"  USING BY VALUE appl_error_handler,
                                       BY REFERENCE user_data.
```

- FORTRAN

```
CALL sql_deregister_error_handler()
CALL sql_get_error_handler(%ref(rtn_ptr),%ref(ctx_ptr))
CALL sql_register_error_handler(%ref(my_error_handler),%ref(reg_ctx)))
```

- Pascal

  In Pascal, you must declare the error-handling routines as external procedures before you call them. The following example shows how to declare the procedures:

```
PROCEDURE sql_deregister_error_handler; EXTERNAL;

PROCEDURE sql_get_error_handler(
  var routine :         integer;
  var data :            integer
  ); EXTERNAL;

PROCEDURE sql_register_error_handler(
  procedure err_hand (
    var msgvec :        RDB$MESSAGE_VECTOR_REC;
    var sqlcode :       integer;
    var sqlstate :      sqlstate_type;
    var data_addr :     user_data_type);
  var data:             user_data_type
  ); EXTERNAL;
```

  The following example shows how to call the procedures:

```
sql_deregister_error_handler;

sql_get_error_handler(routine_addr, data_addr);

sql_register_error_handler(%immed my_error_handler,user_data);
```

- PL/I

In PL/I, you must declare the error-handling routines as external procedures before you call them.

Declaring User-Written Error-Handling Routines

To declare user-written error-handling routines using the SQL precompiler and host language programs, use the following formats:

- Ada

```
procedure my_error_handler (
  msgvec : in pkg_handler.rdb_message_vector;
  sqlcode : in out integer;
  sqlstate : in integer;
  user_info : in pkg_handler.foo_t );)
```

- BASIC

```
SUB BASIC_ERRHND BY REF (INTEGER MSG_VECTOR, &
                                 SQLCODE_VALUE, &
                                 SQLSTATE_VALUE, &
                         CTX_REC USER_CTX )

RECORD CTX_REC
  STRING     VALUE1 = 9
  STRING     VALUE2 = 10
  INTEGER    VALUE3
END RECORD CTX_REC

DECLARE INTEGER SQLCODE

SQLCODE = SQLCODE_VALUE)
```

BASIC is supported by SQL module language only.

- C

```
static
  void appl_error_handler (
       RDB$MESSAGE_VECTOR *msgvec,
       int                *sqlcode,
       char               *sqlstate,
       void               *user_info)
```

- COBOL

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SQLCODE PIC S9(9) COMP.

LINKAGE SECTION.
01 MSG_VECTOR USAGE POINTER.
01 SQLCODE_VALUE USAGE POINTER.
01 SQLSTATE_VALUE USAGE POINTER.
```

```
01 USER_CTX.
    02 VALUE1 PIC X(9).
    02 VALUE2 PIC X(10).
    02 VALUE3 PIC 9(9) USAGE COMP.

PROCEDURE DIVISION USING MSG_VECTOR, SQLCODE_VALUE, SQLSTATE_VALUE, USER_CTX.
HANDLE_ERROR.
    MOVE SQLCODE_VALUE TO SQLCODE.
```

- FORTRAN

```
subroutine my_error_handler (msgvec,sqlcode,sqlstate,usrctx)
integer*4 msgvec
integer*4 sqlcode
integer*4 sqlstate
structure /user_ctx/
   character*10 value1
   character*10 value2
   integer*4 value3
end structure
record /user_ctx/ usrctx)
```

- Pascal

```
procedure my_error_handler(
  var msgvec :      RDB$MESSAGE_VECTOR_REC;
  var sqlcode :     integer;
  var sqlstate :    sqlstate_type;
  var data :        user_data_type);)
```

10.3 Displaying Error Messages

Displaying messages is one of the most common actions in error handling. SQL provides the following methods:

- Call the sql_signal routine to display messages from all facilities and (optionally) terminate your program.
- Call the sql_get_error_text routine to pass error text of messages from all facilities to your program for processing

In addition, you can use host language routines or system service calls to display messages from all facilities and continue program execution or display messages from a message file created for your program.

Source programs in the samples directory include examples using different techniques to display error messages.

10.3.1 Calling sql_signal

The sql_signal routine displays error messages returned through the RDB$MESSAGE_VECTOR array. It signals to your program condition handler an error that occurs on the execution of an SQL statement. The fact that sql_signal signals an error to your program condition handler, rather than to an operating system condition handler, is an advantage to you if:

- The language in which your program is written automatically provides a condition handler for host language statements.

  For example, DEC COBOL for OpenVMS provides a program condition handler that should not be bypassed when an error is signaled.

- You have established your own program condition handler that you want to assume control whenever an error is signaled.

Digital UNIX

On Digital UNIX, the sql_signal routine prints an error message and exits the program when an error occurs. ♦

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the sql_signal routine signals to your program condition handler an error that occurs on the execution of an SQL statement. If your program does not contain a condition handler, the routine prints an error message and exits the program when an error occurs.

If your program condition handler does not recognize an error with the RDB, SQL, or repository facility codes as an error it can handle, it resignals the error to the OpenVMS condition handler. This is likely when your program condition handler is provided automatically for you by the programming language you are using. The OpenVMS condition handler displays messages and either stops your program if the error is severe or lets your program continue if the error is not severe.

For more information about error-signaling concepts, refer to the chapter on condition handling in the OpenVMS Run-Time Library documentation. ♦

To declare prototypes for the sql_signal routine, as well as other SQL routines, in C programs, you can use the include file sql_rdb_headers.h. For more information, see Section 10.3.4.

The sql_signal routine returns no value and requires no parameters. The following example shows how to use it in a C language program:

```
sql_signal();
```

The format of the call to sql_signal is language-specific. The chapter about routines in the *Oracle Rdb7 SQL Reference Manual* describes how to call the routine from different programming languages.

10.3.2 Calling sql_get_error_text

Use the sql_get_error_text routine when you want to pass error text with formatted ASCII output (FAO) substitutions to your program for processing.

To use the sql_get_error_text routine, you must include a buffer (field) in your program declarations to receive the text SQL will pass to it. Declare this field as a text string with a length sufficient to accommodate the number of characters you expect for the message associated with the RDB$LU_STATUS field and for all follow-on messages. As an option, you can declare the buffer length as a separate field (defined as a signed word).

Declare the sql_get_error_text routine using three arguments:

- The buffer to receive the text
- The length of the buffer to receive the text
- The number of characters allotted for the returned error message.

The following example shows how to declare and use the sql_get_error_text routine in a C program:

```
/* This function uses the sql_get_error_text routine to display the messages
   returned by SQL and Rdb for unexpected error conditions.  */
    void display_sqlget_message()

    {

    char    get_error_buffer[1024];
    long    error_msg_len;

    return_status = sql_get_error_text(get_error_buffer, 1024, &error_msg_len);
    get_error_buffer[error_msg_len] = '\0';

    printf("\n\nThis condition was not expected.\n\n");
    printf("%.*s\n", get_error_buffer, error_msg_len );
    release_screen = getchar();
    printf("\n");

    return;
    }
```

The *Oracle Rdb7 SQL Reference Manual* describes how to declare and use the sql_get_error_text routine from a variety of programming languages.

Note that the sql_get_error_text routine returns a carriage return and line-feed character to separate follow-on messages from the primary message, and to separate follow-on messages from each other.

To declare prototypes for the sql_get_error_text routine, as well as other SQL routines, in C programs, you can use the include file sql_rdb_headers.h. For more information, see Section 10.3.4.

The sql_get_error_text routine inserts the characters in the buffer declared to receive the text as delimiters between the messages. Typically, their presence eases display of the text to the terminal screen.

However, if a program uses a forms product to display the message, the carriage return and line-feed characters are interpreted as unprintable characters.

The following COBOL example shows one way to handle the presence of the carriage return and line-feed characters in the buffer:

```
* CRLF is a PIC XX field that contains <cr><lf>.
* MSG-TXT-RDBFEL is an array of lines to be displayed for the error message.
*
    STRING CARRIAGE-RET, LINE-FEED DELIMITED BY SIZE INTO CRLF.
    CALL "sql_get_error_text" USING BY REFERENCE BUFFER,
                                    BY VALUE BUF_LEN
                                    BY REFERENCE MSG_LEN.
    UNSTRING BUFFER DELIMITED BY CRLF INTO MSG-TXT-RDBFEL(1),
        MSG-TXT-RDBFEL(2), MSG-TXT-RDBFEL(3).
```

OpenVMS VAX OpenVMS Alpha

On OpenVMS, SQL also supports the SQL$GET_ERROR_TEXT routine, but because it is not portable to other platforms, Oracle Rdb recommends that you use the sql_get_error_text routine. ♦

10.3.3 Displaying User-Supplied Error Messages

Your program can display literals as error messages by using host language statements that write to the terminal.

Your program can also display messages from a message file customized for its users.

OpenVMS VAX OpenVMS Alpha

For information on editing and processing such a file, refer to the manual that describes the Message utility in the OpenVMS documentation set. ♦

For an example of a message file, refer to the sql$persmsg.msg file in the samples directory. The sql$persmsg.msg file is associated with the program sql_terminate.sco. The sql_terminate.sco program includes parameter declarations and calls to retrieve messages from the image version of sql$persmsg.

10.3.4 Declaring SQL Routines Using an Include File

When you call certain SQL routines in C programs, the programs can generate compilation informational messages. For example, if you do not explicitly declare sql_signal as a function, you receive the following informational message:

```
                sql_signal();
        ........^
%CC-I-IMPLICITFUNC, In this statement, the identifier "sql_signal" is
 implicitly declared as a function.
```

Oracle Rdb provides an include file that eliminates the messages by explicitly providing prototypes for explicitly called SQL functions.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the include file is:

```
SYS$LIBRARY:SQL_RDB_HEADERS.H
```

♦

Digital UNIX

On Digital UNIX, the include file is:

```
/usr/lib/dbs/sql/v<nn>/include/sql_rdb_headers.h
```

♦

10.4 Handling Duplicate Value Errors and Constraint Violations

The integrity of a relational database depends on valid values being stored in columns and also on preserving meaningful relationships between rows stored in different tables. An Oracle Rdb database enforces these requirements through the constraints you define in table definitions. A requirement that a column store a unique value may be enforced either by a constraint specified in a table definition or by the UNIQUE keyword in an index definition.

This section explains:

- The errors your program can encounter for constraint violations and for attempts to store a duplicate value in a unique index
- The options you have for controlling when constraints are evaluated during a transaction

Example 10–5 shows a pseudocode example that checks for constraint violations.

10.4.1 Status Values for Constraint Violations and Duplicate Value Errors

With SQL, you define all data validation requirements as constraints when you create a table. If your program violates a constraint when updating a table, the database system returns the value RDB$_INTEG_FAIL in RDB$LU_STATUS, –1001 in SQLCODE, and '23000' in SQLSTATE.

However, if there is a requirement that a column contain a unique value and if there is a unique index based on that column, the person who defines the database frequently decides to impose the uniqueness requirement (at least after a table is loaded and a unique index is in place) only through the index definition. In this case, your program encounters an error only when SQL attempts to update a unique index with a duplicate value. For this error, the value returned is RDB$_NO_DUP in RDB$LU_STATUS, –803 in SQLCODE, or 'R2000' in SQLSTATE.

If the uniqueness requirement is imposed both as a constraint and in the index definition, whether your program encounters the constraint violation or the duplicate value violation first can vary.

Note

Versions of SQL earlier than Version 2.0 implemented the NOT NULL requirement through the "valid if" option for columns rather than through the database constraint option. If interfaces other than SQL define domains and tables for a database, those interfaces may currently implement both NOT NULL and other validation requirements for columns by using the "valid if" option. Violation of a "valid if" requirement returns a different error than violation of a comparable constraint. Therefore, if you are accessing a database created by versions of Oracle Rdb earlier than Version 2.0, and you want to determine if an integrity violation occurred, monitor for the value –1002 in SQLCODE, for the SQLCODE_NOT_VALID literal value, or for RDB$_NOT_VALID in RDB$LU_STATUS *in addition* to error codes for constraint violations and duplicate value errors.

Uniqueness violations and violations of "valid if" criteria are always detected when SQL statements such as INSERT, UPDATE, or DELETE execute.

10.4.2 Controlling Constraint Evaluation

Constraint violations may be detected either when SQL statements such as INSERT, UPDATE, DELETE, or ALTER TABLE execute, or not until a COMMIT statement executes. Exactly when an error is detected varies depending on the options that were specified when the database was created or altered and when the transaction was started. For example, Oracle Rdb evaluates a constraint when you define it if there is data in the table. If you add a constraint with the ALTER TABLE statement, Oracle Rdb evaluates the constraint and returns a constraint violation if data existing in the table does not meet the criteria defined by the constraint.

Section 16.7 discusses options for controlling constraint evaluation. This section tells how to handle errors that arise from constraint evaluation.

As a general rule, the checking by the database system should not replace host language or forms product routines that your program uses to validate input from a user. Program or forms product routines let you check user input that is used to insert or update rows and, if necessary, prompt a user for corrections, before starting a transaction to write the changes to the database. In this way, the time it takes a transaction to complete is minimal, and locks that the transaction places on table rows and indexes are in place only briefly.

To efficiently handle errors associated with constraints or other requirements on data values, you first need to become thoroughly familiar with the table, domain, index, and constraint definitions associated with the tables your program will update. To the extent that it is practical, you should then write program or forms product routines that impose the same validation requirements that are defined for the database and, if needed, some additional requirements that may be appropriate for the data being entered. If you design your program in this way, you can treat most integrity, "valid if," and duplicate index value errors as unexpected errors rather than expected errors.

10.5 Handling Lock Conflicts and Deadlocks

Lock conflicts and deadlock errors occur with multiuser database access. These errors are most likely to occur in high-contention, multiuser environments, particularly those supporting many read/write transactions that are performing data update simultaneously with data retrieval. Lock conflicts and deadlocks are less common in single-user databases, or databases that are accessed solely in read/write, shared read, or read-only modes, but they can still occur and your program should be prepared to handle them.

10.5.1 Handling Lock-Conflict Errors

A lock-conflict error typically occurs when you have specified the no-wait option in your DECLARE TRANSACTION or SET TRANSACTION statement and the data your program needs is locked by another user's transaction. When the shared share mode applies to all transactions accessing the same table, lock-conflict errors (if any) are not usually encountered by any of those transactions until they start to process rows in the table. However, a lock conflict error can occur also at the following times:

- On transaction start, if you specify the no-wait option

  A lock conflict error occurs on transaction start when tables that you specified in a RESERVING clause in a DECLARE TRANSACTION or SET TRANSACTION statement are locked by transactions that other users have started in protected or exclusive share mode. It can occur also on transaction start if other users are processing tables that your transaction statement explicitly reserves in protected or exclusive mode.

- On database attachment, regardless of the transaction options you choose

  In interactive SQL, if all the system tables in a Oracle Rdb database are locked by other users, you may encounter the lock-conflict error when you enter an ATTACH statement. (Users who are executing data definition statements are frequently the cause of system table locking.) Oracle Rdb must start an internal transaction using the system table at the time a database is attached, and returns the lock-conflict error if system tables are locked. You may encounter a lock-conflict error for the same reason in programs with SET TRANSACTION, OPEN, SELECT, UPDATE, DELETE, INSERT, CREATE, ALTER, or DROP statements. Any of these statements may cause database attachment.

- During transactions declared read-only, even if you specify the WAIT option

  Oracle Rdb considers the exclusive write mode to be incompatible with a transaction declared read-only. A transaction that specifies the exclusive write mode does not write data to the snapshot file. However, all data committed to the database before a read-only transaction starts must be available to that read-only transaction. It is impossible to determine whether a transaction in exclusive write mode has written data to the database by using the snapshot file; therefore, the read-only transaction's requirements can never be satisfied. In this case, Oracle Rdb disregards the WAIT option.

For example, if a user starts a transaction using the exclusive write mode, and your program has declared a transaction read-only with the WAIT option, your program waits until the first transaction ends. Then, your program encounters an error on the first executable SQL statement and Oracle Rdb rejects the program's WAIT option and issues a lock-conflict error to the program.

Therefore, it is good practice to anticipate lock conflict at times other than during a transaction, even if you specify the WAIT option for all transactions.

You can detect lock conflict by monitoring SQLCODE for the numeric value –1003 or the literal value SQLCODE_LOCK_CONFLICT or by monitoring SQLSTATE for a value of R1002.

If your program detects a lock-conflict error after a transaction is started, it should roll back the transaction and do one of the following:

- Try to start the transaction again immediately
- Try to start the transaction again after a set period of time
- Start the transaction again using the WAIT option
- Display a message telling the user to run the program at another time and then stop execution

If you decide to start the transaction again, use the SET TRANSACTION statement and not the DECLARE TRANSACTION statement to start the transaction in wait mode. You can specify a wait interval in the SET TRANSACTION statement.

If you do not roll back the transaction after a lock-conflict error, subsequent statements may produce unexpected and unexplained errors.

The sql_terminate sample program in the samples directory shows how to detect and handle a lock-conflict error.

10.5.2 Handling Deadlock Errors

A deadlock error occurs when data you need is locked by another user and you have locked data the other user needs. In this case, neither you nor the other user can continue. The database system returns a deadlock error to one of the users. The deadlocked transaction must roll back to release locks so the other user can continue.

A deadlock error can occur over database resources other than tables. For example, simultaneous needs to read and update the same node of an index can cause a deadlock error. Deadlock errors are infrequent and unpredictable on a case-by-case basis; however, the likelihood of deadlock increases with the

number of users who are simultaneously accessing a table in shared share mode.

You can detect deadlock by monitoring SQLCODE for the numeric value –913 or the literal value SQLCODE_DEADLOCK in SQLCODE or by monitoring SQLSTATE for a value of R1001.

Note

If you use the SQLV40 dialect, and your program encounters a lock-conflict error on database attachment, or either a lock-conflict or deadlock error on the transaction start itself, a COMMIT or ROLLBACK statement fails because no transaction is started. See Section 10.6 for the appropriate action to take in this case.

Although Oracle Rdb allows you to continue to issue updates to the database after it detects a deadlock error, the statement that caused the deadlock is not applied to the database.

The sql_terminate sample program in the samples directory shows how to detect and handle a deadlock error.

10.6 Handling Errors Caused by Failure to Attach to a Database or Start a Transaction

You may want to write an error-handling section designed to handle fatal errors and may not be sure if a rollback operation is appropriate in all cases. For example, if the transaction context for your program is specified using a DECLARE TRANSACTION statement, then any database attachment and transaction start failures occur on the first execution of statements such as SELECT . . . INTO, OPEN, UPDATE, DELETE, or INSERT. If these statements are specified in a loop that also contains a COMMIT statement, it is possible that they may fail to start a new transaction during any iteration of the loop.

Note

A FETCH statement may be included in a loop but cannot start a transaction. If a cursor is closed, you cannot continue to fetch rows from it.

If you use the SQLV40 dialect and a statement fails because it does not start a transaction as it normally would, branching to an error-handling section of your program that attempts to execute a ROLLBACK statement causes the ROLLBACK statement to fail. The COMMIT and ROLLBACK statements fail when there is no active transaction. Unless you handle this situation, failure of the ROLLBACK statement may cause premature exit of the error-handling section or your program may continue unexpectedly.

One way to test for an active transaction is to use a multistatement procedure with the GET DIAGNOSTICS statement, as follows:

```
BEGIN
    DECLARE :txn INTEGER;
    GET DIAGNOSTICS :txn = TRANSACTION_ACTIVE;
    IF :txn = 1
         THEN ROLLBACK;
         END IF;
END;
```

If you use a dialect other than SQLV40, the ROLLBACK and COMMIT statements do not fail when a transaction is not active.

Another way to handle this problem is to explicitly start all transactions with a SET TRANSACTION statement. Then, you can write error-handling actions specifically for the SET TRANSACTION statement. The actions can omit the ROLLBACK statement because you can safely assume that either database attachment or transaction start or both failed to occur.

10.7 Improving Program Portability When Handling Errors and Constraints

This section applies only to applications that may need to work with more than one SQL product. The guidelines in this section supplement, rather than replace, any information about Oracle Rdb extensions to SQL or host languages that you may find in the *Oracle Rdb7 SQL Reference Manual* or host language documentation.

You should keep in mind that not all SQL implementations include all aspects of the standards, so even when something is standard, it might not be portable. Consider the following guidelines:

- If your objective is to write code that requires the least amount of change from one implementation of SQL to another, you should monitor statement execution and display error information using only SQLSTATE or (in precompiled programs) SQL WHENEVER statements.

- SQLSTATE values are defined to be consistent across vendors, but not all vendors implement them.
- Not all SQLCODE values are portable. For instance, the values 0, 100, less than 0, and greater than 0 should always have consistent meaning from one SQL implementation to another. However, the meaning of specific positive or negative SQLCODE values, such as –1003, often depend on database behavior that is vendor-defined.
- The SQLCA is not included in the ANSI/ISO SQL standard.
- Do not insert operating system-specific calls, such as the OpenVMS system service calls, directly in host language source files.
- Use portable SQL routines. For example, use the sql_get_error_text routine, rather than the non-portable SQL$GET_ERROR_TEXT routine.
- If you do not want to sacrifice the additional error-handling flexibility and information that can be provided only through system-specific routines, constructs, and SQLCODE values, consider developing system-specific modules that you can copy or include into source files before or as part of the process of compiling programs. Using this strategy in precompiled programs, you monitor the execution of SQL statements using either SQLCODE or WHENEVER statements, and you copy in an error-handling section that is system-specific. For example:

```
IF SQLCODE is less than 0
   branch to error-section
   .
   .
   .
error-section
   IF SQLCODE equals value-list       -
      system-specific actions          |
   else IF SQLCODE equals value-list   |  Statements to be
      system-specific actions          |  copied.
   else                                |
      system-specific actions          |
   end of IF block                    -
end-of-error-section
```

- The SQL interface to Oracle Rdb provides constraint checking at verb time and at commit time, but the ANSI/ISO SQL standard requires constraint checking at the end of each statement. If you want ANSI/ISO-standard SQL compatibility, use SQL module processor and SQL precompiler command line qualifiers to change the constraint evaluation mode for commit-time constraints.

To evaluate commit-time constraints at the end of each statement, use the CONSTRAINTS=immediate or `-cm immediate` qualifier for the SQL module language processor or the SQLOPTIONS=(CONSTRAINTS=IMMEDIATE) or `-s '-cm immediate'` qualifier for the SQL precompiler.

11

Using Dynamic SQL

Dynamic SQL lets programs accept or generate SQL statements at run time. Unlike precompiled SQL or SQL module language, in which SQL statements are known at compile time, dynamic SQL lets the user formulate the statements at run time.

This chapter describes how to use the dynamic SQL interface. The sections that follow explain how to:

- Use the terminology and concepts associated with dynamic execution of SQL statements
- Write programs that dynamically execute SQL non-SELECT statements without parameter markers
- Use parameter markers and select list items in your programs
- Write programs that dynamically execute non-SELECT statements that contain parameter markers
- Write programs that dynamically execute SQL SELECT statements with and without parameter markers
- Use a single set of dynamic SQL statements to concurrently process any number of dynamically generated statements
- Use the dynamic SQL programs in the sample directory

11.1 Introducing Dynamic SQL

Precompiled SQL and SQL module language are both forms of **static SQL**—when the program is compiled, it already has all necessary information about the database, data structures, and the SQL statements it will process. If you do not know what SQL statements need to be processed until the program is run, you need to use **dynamic SQL**. Dynamic SQL lets you write an application program even when you cannot predict the form or type of SQL statement your program needs to process.

This chapter often uses the terms dynamic SQL statements and dynamically generated and executed SQL statements. **Dynamic SQL statements** are SQL statements embedded in a program's source code that are used to process **dynamically generated and executed SQL statements**. Dynamically generated and executed statements are not necessarily part of a program's source code, but can be formulated and executed while the program is running.

Reference Reading

The *Oracle Rdb7 SQL Reference Manual* lists in detail the SQL statements that can be dynamically generated and executed, along with the dynamic SQL statements used to process them.

11.1.1 Categories of Statements That Can Be Dynamically Executed

SQL statements to be dynamically executed fall into several categories that require increasingly complex programs, depending on the kinds of SQL statements your program processes at run time and how much information your program has about those statements before it processes them.

SQL does not allow host language variables in SQL statements prepared for dynamic execution. However, in places where SQL allows host language variables embedded in an SQL statement, you can instead specify parameter markers. **Parameter markers** indicate parameters in SQL statements to be prepared for dynamic execution and are represented by question marks (?) in the SQL statements.

When you process a SELECT statement in dynamic SQL, you must find out information about the number of select list items in order to handle the output.

The presence of parameter markers and of select list items changes the kind of processing an SQL statement requires. The categories of dynamic SQL statements include:

- Statements other than SELECT statements that do not contain parameter markers

  You can use the dynamic SQL statement EXECUTE IMMEDIATE to execute these statements. To dynamically execute the statement more than once, use the PREPARE and EXECUTE statements, instead of the EXECUTE IMMEDIATE statement. See Section 11.2.

- Statements other than SELECT statements that contain parameter markers

If a statement includes parameter markers, you cannot use the EXECUTE IMMEDIATE statement. Instead, your program passes the SQL statement to the PREPARE statement and then to the DESCRIBE . . . MARKERS statement. SQL determines the number of parameter markers and writes that information to the SQLDA structure. Your program uses that information to allocate storage for the parameters, and then uses the EXECUTE statement to execute the statement.

Your program must declare host language variables or allocate dynamic memory corresponding to all the parameter markers specified in the prepared non-SELECT statement.

You can use parameter markers in INSERT, UPDATE, and DELETE statements, as well as in SELECT statements.

Section 11.3 provides general information on how to process parameter markers, as well as select list items. Section 11.4 provides information about processing non-SELECT statements that contain parameter markers. Section 11.3.1 describes how to use the SQLDA structure.

- SELECT statements that do not contain parameter markers

  Your program must declare a dynamic or extended dynamic cursor for the SELECT statement to receive the values of rows in the result table created by the SELECT statement and must include an SQLDA structure to receive information about any select list items in the statement. Your program uses the DESCRIBE . . . SELECT LIST statement to get information about the select list items, and then uses that information to allocate storage for each select list item in the SELECT statement.

  After you open the cursor for the SELECT statement, issue FETCH statements to return the values from the result table to your host language variables values. Note that the FETCH statement itself cannot be dynamically executed.

  Section 11.6 details how to process SELECT statements. Section 11.6.1 deals specifically with SELECT statements that do not contain parameter markers.

- SELECT statements that contain parameter markers

  If the SELECT statement contains parameter markers as well as select list items, your program must set up host language variables for the parameter markers and assign values to them. This means you must allocate two SQLDA structures, one to receive information about select list items and one to receive information about parameter markers.

Section 11.6 details how to process SELECT statements. Section 11.6.2 deals specifically with SELECT statements that contain parameter markers.

- Any statement with unknown number and data type of parameter markers, select list items, or both

  Often, a program does not have any information about the number and data type of parameters markers or select list items. These are the most complicated statements for programs to handle in dynamic SQL.

  Section 11.7 explains how to write programs that handle such situations.

SQL needs to determine the number and data type of parameter markers and select list items and communicate that information to the program. Once the program has the number and data types of parameters, it must allocate memory for them and communicate the location of that memory to SQL. Dynamic SQL uses a structure called the SQLDA (SQL Descriptor Area) to communicate information about parameter markers or select list items (or both) to your program.

Section 11.3.1 explains the SQLDA in more detail. Section 11.8 describes a sample program, sql_dynamic, that illustrates many of the processing techniques discussed in this chapter. Many of the examples in this chapter use excerpts of the sample program in C. Versions in other languages are also included in sample directory.

11.1.2 Using Dynamic SQL Statements to Process Other SQL Statements

To process dynamically generated and executed statements, dynamic SQL provides the following statements:

- EXECUTE IMMEDIATE: Prepares and executes in one step any statement (other than the SELECT statement) that has no parameter markers.
- PREPARE: Checks the SQL statement to be dynamically executed for errors and assigns an identifier to it. That identifier is referred to in the DESCRIBE, EXECUTE, dynamic DECLARE CURSOR, and extended dynamic DECLARE CURSOR statements. The PREPARE statement generates the code that will be executed.
- DESCRIBE: Checks a prepared statement for the existence of parameter markers or select list items. If there are any, it stores information about their number and data type in the SQL Descriptor Area (SQLDA).
- EXECUTE: Executes a previously prepared statement other than the SELECT statement.

- Dynamic DECLARE CURSOR: Declares a cursor for a prepared SELECT statement. A dynamic DECLARE CURSOR statement specifies the cursor name at compile time; however, it does not explicitly specify the SELECT statement. Instead, you supply the name of a prepared statement where you would ordinarily place the SELECT statement.
- Extended dynamic DECLARE CURSOR: Declares a cursor for which neither the cursor name nor the SELECT statement are known until run time. Instead, you supply parameters for the cursor name and for the SELECT statement.

  SQL treats the extended dynamic DECLARE CURSOR statement as an executable statement, unlike how SQL treats the nonextended dynamic DECLARE CURSOR statement.
- OPEN: Opens a cursor declared for a prepared SELECT statement.
- FETCH: Retrieves values from a cursor declared for a prepared SELECT statement.
- RELEASE: Releases all resources used by a prepared SQL statement and prevents the prepared SQL statement from executing again.

Note that none of these statements used to process dynamically executed statements can themselves be dynamically executed.

Reference Reading

The *Oracle Rdb7 SQL Reference Manual* chapter on SQL statements contains a section on each of the preceding statements.

A table in the *Oracle Rdb7 SQL Reference Manual* lists the statements that SQL can dynamically execute, whether or not they allow parameter markers and select list items, and the associated embedded statements used to process them.

11.1.3 Processing SQL Statements in Dynamic SQL

Programs that accept SELECT statements or statements with parameter markers follow these general steps to process an SQL statement at run time:

1. Accept the statement into a program variable.
2. Pass the program variable to the PREPARE statement.
3. Use the DESCRIBE . . . MARKERS statement to find out if the statement has parameter markers and, if it does, allocate storage for those variables.

4. Test whether the statement is a SELECT statement. If it is not a SELECT statement, process it with the EXECUTE statement.
5. If it is a SELECT statement:
   a. Use the DESCRIBE . . . SELECT ITEMS statement to find the select list items in the parameter string. Allocate appropriate storage for them.
   b. Open a cursor.
   c. Fetch each row in the result table using the storage allocated.
6. Release the prepared SQL statement when the program finishes.

If your program processes only non-SELECT statements that do not use parameter markers, you can use the EXECUTE IMMEDIATE statement, explained in Section 11.2.

11.2 Executing Non-SELECT Statements Without Parameter Markers

When the SQL statement you want to process is not a SELECT statement and it does not contain any parameter markers, you do not need to prepare it and then execute it. You can combine the steps in a single EXECUTE IMMEDIATE statement.

The EXECUTE IMMEDIATE statement is useful for any write-only or data definition application that generates a statement string without parameter markers.

Example 11–1 illustrates such a case. The COBOL program accepts a statement from the terminal and processes it dynamically with an EXECUTE IMMEDIATE statement. The program allows only non-SELECT statements for dynamic execution. Because the statements are generated interactively, they do not contain parameter markers.

Example 11–1 Executing Non-SELECT Statements Using the EXECUTE IMMEDIATE Statement

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   EXECUTE_IMMEDIATE_EXAMPLE.
*
* Illustrate EXECUTE IMMEDIATE with a dynamic INSERT statement.
*
```

(continued on next page)

Example 11–1 (Cont.) Executing Non-SELECT Statements Using the EXECUTE IMMEDIATE Statement

```
DATA DIVISION.
WORKING-STORAGE SECTION.

* Variable for DECLARE ALIAS:
01 FILESPEC     PIC X(20). ❶

* Variable to hold the command to be dynamically executed:
01 COMMAND_STRING       PIC X(240). ❷

* Buffer for error handling:
01 GETERRVARS.
           02  error-buffer-len        PIC S9(9) COMP VALUE 132.
           02  error-msg-len           PIC S9(9) COMP.
           02  error-buffer            PIC X(132).

* Load definition for SQL Communications Area (SQLCA) for error handling:
EXEC SQL        INCLUDE SQLCA END-EXEC.

*************************************************************
*
*       P R O C E D U R E    D I V I S I O N
*
*************************************************************
PROCEDURE DIVISION.
START-UP.

* Assign value to FILESPEC:
     DISPLAY "Enter the file spec for the database you want to declare:"
     ACCEPT FILESPEC. ❸

* Declare the database:
* (You can name any file for COMPILETIME FILENAME because no
* embedded statement in this example requires that SQL attach
* to the database.)

     EXEC SQL DECLARE ALIAS ❹
              COMPILETIME FILENAME personnel
              RUNTIME FILENAME :FILESPEC
     END-EXEC

     DISPLAY "Enter an SQL statement (not SELECT):"
     ACCEPT COMMAND_STRING. ❺

* Use EXECUTE IMMEDIATE to execute the statement in COMMAND_STRING:
     EXEC SQL EXECUTE IMMEDIATE :COMMAND_STRING ❻
     END-EXEC
     PERFORM CHECK

     EXEC SQL EXECUTE IMMEDIATE 'ROLLBACK' END-EXEC.
     PERFORM CHECK.

     DISPLAY "Rolled back changes.  All done.".
```

(continued on next page)

Example 11–1 (Cont.) Executing Non-SELECT Statements Using the EXECUTE IMMEDIATE Statement

```
CLEAR-IT-EXIT.
     STOP RUN.

CHECK.
     IF SQLCODE NOT = 100 AND SQLCODE NOT = 0 ❼
             DISPLAY "Error: SQLCODE = ", SQLCODE WITH CONVERSION
        CALL "sql_get_error_text" USING BY REFERENCE error-buffer,
                                        BY VALUE error-buffer-len,
                                        BY REFERENCE error-msg-len
        DISPLAY error-buffer
     END-IF.
```

The following callouts are keyed to Example 11–1:

❶ Declares the variable FILESPEC for the file name of the database to be declared.

❷ Declares the variable COMMAND_STRING for the SQL command that is to be executed dynamically.

❸ Accepts input from the terminal for the FILESPEC variable.

❹ Declares the alias. Because the DECLARE ALIAS statement is embedded in the source program, this program does not consider it to be a dynamically executable statement. You pass a variable to the DECLARE ALIAS statement in the same way you pass variables to statements in static SQL programs.

❺ Accepts input from the terminal for the COMMAND_STRING variable.

❻ Uses the EXECUTE IMMEDIATE statement to execute the statement entered at the terminal.

❼ Checks for errors. If SQL returns an error, the program displays the value of SQLCODE and the text for the error.

If your program will dynamically execute a particular statement more than once, it is more efficient to issue a PREPARE statement and then execute the statement each time with the EXECUTE statement. For example, you would modify Example 11–1 by substituting the following lines for the EXECUTE IMMEDIATE :COMMAND_STRING statement:

```
* Use these commands to execute one statement several times.
     EXEC SQL PREPARE STMT1 FROM :COMMAND_STRING
     END-EXEC
          PERFORM CHECK
     DISPLAY "Statement prepared."

     PERFORM UNTIL RESPONSE  = "NO"

        EXEC SQL EXECUTE STMT1
        END-EXEC
        PERFORM CHECK

        DISPLAY "Do you want to execute the statement again:"
        ACCEPT RESPONSE

     END-PERFORM.
```

You can also use dynamic SQL and the EXECUTE IMMEDIATE statement to work around the common problem of trying to drop and then create again the same schema object in one program.

For example, if a precompiled SQL program contains both a DROP INDEX statement and a CREATE INDEX statement, the precompiler returns an error similar to the following:

```
%SQL-F-IND_EXISTS, Index DEG_EMP_ID already exists in this database or schema
```

Because the index is not dropped during the *compilation*, it still exists in the compile-time database, thus generating the error. To work around this restriction, use the EXECUTE IMMEDIATE statement to drop and then create the index again.

11.3 Handling Parameter Markers and Select List Items

This section explains how to handle parameter markers and select list items. For information about processing statements that contain only parameter markers, see Section 11.4. For more information about processing SELECT statements, see Section 11.6.

You can use parameter markers in INSERT, UPDATE, and DELETE statements, as well as in SELECT statements. SQL replaces parameter markers with values in host language variables or dynamic memory when the prepared statement is dynamically executed by a subsequent EXECUTE statement.

If a statement includes parameter markers, the program must declare host language variables or allocate dynamic memory corresponding to all the parameter markers specified in the prepared non-SELECT statement.

A statement with parameter markers or select list items cannot use the EXECUTE IMMEDIATE statement. Instead, the program must prepare the statement for dynamic execution with the PREPARE statement. In addition, the program must declare host language variables or allocate dynamic memory for all parameter markers and for all select list items. To execute the statement, the program issues an embedded EXECUTE statement or calls an SQL module language procedure that contains an EXECUTE statement.

Your program declares a data structure called the SQLDA and then uses the DESCRIBE statement to get information about the parameter markers or select list items.

When SQL processes a DESCRIBE statement, it writes the number and data type of any select list items (for a DESCRIBE . . . SELECT LIST statement) or parameter markers (for a DESCRIBE . . . MARKERS statement) of a prepared statement into the SQLDA. Your host language program reads this information from the SQLDA, allocates storage (host language variables or dynamic memory) for each of the parameters, and writes the addresses for that storage to the SQLDA. The program also supplies values that will be substituted for parameter markers and writes those values to the storage it allocated for parameter markers.

When you use parameter markers in SQL statements, you should not make any assumptions about the data types of the parameters. SQL may convert the parameter to a data type that is more appropriate to a particular operation. For example, when you use a parameter marker as one value expression in a LIKE predicate, SQL returns a data type of VARCHAR for that parameter, even though the other value expression has a data type of CHAR.

Reference Reading

The *Oracle Rdb7 SQL Reference Manual* section on the DESCRIBE statement discusses the MARKERS and SELECT LIST clause in more detail.

11.3.1 Using the SQLDA and SQLDA2 Structures

SQL needs to communicate to the program information about the number of parameter markers and select list items, if any, and their data types. Once the program has the number and data types of parameters, it must allocate memory for them and communicate to SQL the location of that memory. Programs must write the number and data type of any parameter markers and select list items into the SQL Descriptor Area. The **SQL Descriptor Area** (SQLDA) is a data structure that SQL provides to hold this information.

The SQLDA holds information about any parameter markers and select list items in a prepared statement. SQL writes information about the number and data type of parameter markers and select list items to the SQLDA when it processes a DESCRIBE . . . SELECT LIST or DESCRIBE . . . MARKERS statement. See Section 11.6.3 for information about using SQLDAs to test for the presence of parameter markers and select list items and for information about using the SQLERRD field to determine whether a statement is a SELECT statement.

Of the languages supported by the precompiler, Ada, C, and PL/I can use the SQLDA. Any other language that supports pointer variables can use the SQLDA, but must call SQL module procedures that contain SQL statements instead of embedding the SQL statements directly in the source code.

Host languages that do not support pointer variables cannot take advantage of the full flexibility of dynamic SQL. They are limited to dynamic SQL applications that use some other means to determine if dynamically generated SQL statements have parameter markers and select list items, and if they do, their number and data type.

SQL also provides the **SQL Descriptor Area 2** (SQLDA2), an extended version of the SQLDA, which supports additional fields and field sizes. You can use either the SQLDA or SQLDA2 in any dynamic SQL programs that call for a descriptor area; however, if you want to use any of the date-time data types, you must use the SQLDA2. Section 11.3.2 shows how to declare this structure in a C program. The C version of the sql_dynamic sample program shows how to use the SQLDA2 structure in a dynamic SQL program.

Note

Remember that SQL may use a data type for the parameter marker or select list item that is not supported by the language you are using. The DESCRIBE statement returns information about the data type of parameter markers and select list items to the SQLTYPE and SQLLEN fields of the SQLDA. If necessary, your program can change the value of the SQLTYPE and SQLLEN fields to a data type and length that both SQL and the host language support.

11.3.2 Declaring SQLDA and SQLDA2 Structures

You can declare the SQLDA and SQLDA2 by using the INCLUDE statement or by explicitly declaring the SQLDA or SQLDA2 in programs written in host languages that support pointer variables. In addition, for Ada precompiled programs, you can specify the SQLDA_ACCESS type; for C precompiled programs and C host language programs, you can use the sql_sqlda.h header file.

The following example shows how to include the file in a C program:

```
#include <sql_sqlda.h>
```

The sql_sqlda.h header file includes typedef statements for the SQLDA and SQLDA2 structures defining the SQL_T_SQLDA and SQL_T_SQLDA2 data types. In addition, it defines the SQL_T_SQLDA_FULL and SQL_T_SQLDA2_FULL data types as superset definitions of the SQLDA and SQLDA2 structures. The SQL_T_SQLDA_FULL data type is identical in layout to the SQL_T_SQLDA data type except that it contains additional unions with additional fields that SQL uses when describing CALL statements. Likewise, the SQL_T_SQLDA2_FULL data type is identical in layout to the SQL_T_SQLDA2 data type except that it contains additional unions with additional fields that SQL uses when describing CALL statements. These fields use the same names in both the SQL_T_SQLDA_FULL and SQL_T_SQLDA2_FULL data types, as follows:

- SQLPRCSN—length of numeric fields
- SQLSCALE—scale of numeric fields
- SQLARGPOS—ordinal position of argument
- SQLFLAGS—16 flag bits, used in the following fields:
  - SQLFLAGS_VALID—flags and ordinal position are valid
  - SQLPARAM_IN—parameter is input
  - SQLPARAM_OUT—parameter is output

Field reference expressions for the SQL_T_SQLDA and SQL_T_SQLDA2 data types are the same as those generated when you use the INCLUDE SQLDA or SQLDA2 statement. Refer to the sql_sqlda.h header file to determine the correct field reference expression when using either the SQL_T_SQLDA_FULL or SQL_T_SQLDA2_FULL data types.

If you use the INCLUDE SQLDA statement, you cannot use the INCLUDE SQLDA2 statement in the same program because both statements define a globally visible data object using the same name. However, because the sql_sqlda.h header file does not define any globally visible data definitions or references, you can use the sql_sqlda.h header file in the same program as either INCLUDE statement. In programs that use the header file, you can create definitions or references using typedefs for the SQLDA, the SQLDA2, or both.

Digital UNIX

On Digital UNIX, you can use the INCLUDE SQLDA or INCLUDE SQLDA2 statement only once in your application.

On Digital UNIX, the fields in the SQLDA and SQLDA2 structures are naturally aligned on quadword boundaries. If you define an SQLDA or SQLDA2 structure in a host language, you must align it to make it compatible with the SQL internal format. For example, you can use the COBOL `--align` switch. ♦

OpenVMS VAX OpenVMS Alpha

On OpenVMS, you can use the INCLUDE SQLDA or INCLUDE SQLDA2 statement only once in your application if your application shares the globally visible data object between multiple images. ♦

OpenVMS Alpha

On OpenVMS Alpha, when you use the SQL module processor and specify C in the module header language clause, or when you use the SQL precompiler for C, SQL aligns fields in structures by default. However, you should not allow member alignment of the SQLDA and SQLDA2. ♦

If you explicitly define the SQLDA and SQLDA2 structures, you must surround the structure definitions with the C preprocessor directive #pragma nomember_alignment to prevent alignment of the structures. If you use the SQL INCLUDE statement, you do not need to use the preprocessor directive.

The sql_dynamic program described in Section 11.8 declares two separate SQLDA2 structures, SQLDA_IN and SQLDA_OUT. Example 11–2 shows these declarations. The SQLDA_IN structure is the target of DESCRIBE . . . MARKERS statements and receives information about any parameter markers in the statement string entered by the user. The SQLDA_OUT structure is the target of PREPARE and DESCRIBE . . . SELECT LIST statements and receives information about any select list items in the statement string.

The program explicitly declares both structures as pointer structures, so that it can allocate memory for the structures dynamically at run time. Example 11–2 shows the SQLDA2 declarations from sql_dynamic.c. The *Oracle Rdb7 SQL Reference Manual* provides examples of SQLDA2 declarations for other languages.

Example 11–2 Declaring SQLDA2 Structures

```
/* Declare the SQLDA2.  */

typedef
    struct
        {
            char  sqldaid[8];
            int   sqlabc;
            short sqln;
            short sqld;
            struct sqlvar_struct
                {
                    short sqltype;
                    long  sqllen;
                    int  sqloctet_len;
                    char  *sqldata;
                    int  *sqlind;
                    int  sqlchrono_scale;
                    int  sqlchrono_precision;
                    short sqlname_len;
                    char  sqlname[IDLEN];
                    char  sqlchar_set_name[IDLEN];
                    char  sqlchar_set_schema[IDLEN];
                    char  sqlchar_set_catalog[IDLEN];
                } sqlvar[MAXPARAMS];
        } sqlda_rec, *sqlda;
   .
   .
   .
  int   sql_dynamic (psql_stmt, input_sqlda, output_sqlda, stmt_id, is_select)
  char  *psql_stmt;
  sqlda *input_sqlda;
  sqlda *output_sqlda;
  long  *stmt_id;
  int   *is_select;

  {
      sqlda sqlda_in, sqlda_out;    /* declare the SQLDA structures */
   .
   .
   .
```

Reference Reading

The *Oracle Rdb7 SQL Reference Manual* contains an appendix describing the SQLDA.

11.4 Executing Non-SELECT Statements with Parameter Markers

If a statement includes parameter markers, the program must declare host language variables or allocate dynamic memory corresponding to all the parameter markers specified in the prepared SQL statement. The program uses the information in the SQLDA to allocate storage for parameter markers.

Example 11–3 shows a simple precompiled C program that dynamically executes non-SELECT statements with parameter markers.

Example 11–3 Executing Non-SELECT Statements with Parameter Markers

```
/* This program dynamically executes non-SELECT statements that contain
* parameter markers.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql_rdb_headers.h>

/* Maximum number of parameters or select list items is 50.*/
#define MAXPARAMS 50

#if defined (__osf__) && defined (__DECC)
#pragma member_alignment save
#pragma member_alignment
#elif defined (__DECC)
#pragma member_alignment save
#pragma nomember_alignment
#endif
#pragma nostandard

struct SQLDA_STRUCT {                                    ❶
          char SQLDAID[8];
          int SQLDABC;
          short SQLN;
          short SQLD;
          struct {
            short SQLTYPE;
            short SQLLEN;
            char *SQLDATA;
            int  *SQLIND;
            short SQLNAME_LEN;
            char SQLNAME[30];
              } SQLVAR[MAXPARAMS];
       } *SQLDA;
```

(continued on next page)

Example 11–3 (Cont.) Executing Non-SELECT Statements with Parameter Markers

```
#if defined (__osf__) && defined (__DECC)
#pragma member_alignment restore
#elif defined (__DECC)
#pragma member_alignment restore
#endif
#pragma standard

main()

{

    char         command_string[256];     /* SQL statement text  */
    long         SQLCODE;
    short        param;
    int          loop_cntr;
    char         column_name[31];

/* Declare pointer variables for each possible data type.  The program
uses the pointers to dynamically allocate memory to hold values for
parameter markers.
*/

    char         *charbuf;
    long         *longbuf;
    short        *smallintbuf;
    SQL_DATE_VMS my_date_vms;

/* Allocate SQLDA structures.  */                    ❷

SQLDA = malloc(534);
SQLDA->SQLN = MAXPARAMS;


/* Declare the alias. */
     EXEC SQL DECLARE ALIAS FILENAME personnel;

/* Start an SQL transaction. */
    EXEC SQL SET TRANSACTION READ WRITE;
     if (SQLCODE != 0)
       goto err;

    printf("Enter any dynamically-executable SQL statement, \n");     ❸
    printf("except SELECT. \n");

    gets(command_string);
     if (SQLCODE != 0)
       goto err;

/* Prepare the statement from input entered at run time. */
    EXEC SQL PREPARE STMT_NAME FROM :command_string;                  ❹
     if (SQLCODE != 0)
       goto err;
```

(continued on next page)

Example 11–3 (Cont.) Executing Non-SELECT Statements with Parameter Markers

```
/* Write information about the parameter markers into the SQLDA. */
    EXEC SQL DESCRIBE STMT_NAME MARKERS INTO SQLDA;                    ❺

/* For each parameter marker, check the value of SQLDA.SQLVAR.SQLTYPE
to determine the data type of the parameter marker, and branch to
the appropriate code to prompt and store the value for that data type.
*/

    param = 0;
    for (param = 0; param < SQLDA->SQLD; param++)                      ❻
      {
      /* Handle null-terminated column name and prompt with the column name. */
      strncpy (&column_name[0], &SQLDA->SQLVAR[param].SQLNAME[0],
                                SQLDA->SQLVAR[param].SQLNAME_LEN);
      column_name[SQLDA->SQLVAR[param].SQLNAME_LEN] = '\0';

      printf ("\nEnter value for parameter '%s'", column_name);

      switch (SQLDA->SQLVAR[param].SQLTYPE)                            ❼
        {
        case 453 :                          /* character variable */
        case 449 :                          /* variable char variable */

          printf("\n(Maximum length is %d)",SQLDA->SQLVAR[param].SQLLEN);
          printf("\ndsql> ");

          /* Allocate storage and get the address for the character
             string. */
          charbuf = (char *) malloc (SQLDA->SQLVAR[param].SQLLEN);

          /* Get the string. */
          gets (charbuf);

          /* Assign the address to SQLDA. */
          SQLDA->SQLVAR[param].SQLDATA = charbuf;

          /* Change the character data type to null-terminated. */
          SQLDA->SQLVAR[param].SQLTYPE = 506;                          ❽

          break;  /* Exit the switch block. */

        case 497 :                          /* integer variable */

          printf ("\ndsql> ");

          /* Allocate storage for the character string version of
             the integer to be entered.
          */
          charbuf = (char *) malloc (14);

          /* Get the string. */
          gets(charbuf);
```

(continued on next page)

Example 11–3 (Cont.) Executing Non-SELECT Statements with Parameter Markers

```
        /* Allocate storage for the integer and convert the string. */
        longbuf  = (long *) malloc (SQLDA->SQLVAR[param].SQLLEN);
         *longbuf = atol(charbuf);

        /* Assign the address to SQLDA. */
        SQLDA->SQLVAR[param].SQLDATA = (char *) longbuf;


        break;  /* Exit the switch block. */

    case 501 :                          /* short int variable */

        printf ("\ndsql> ");

        /* Allocate storage for the character string version of
           the short integer to be entered. */

        charbuf = (char *) malloc (5);

        /* Get the string. */
        gets (charbuf);

        /* Allocate storage for the short int and convert the string. */
        smallintbuf  = (short *) malloc(SQLDA->SQLVAR[param].SQLLEN);
        *smallintbuf = atoi(charbuf);

        /* Assign the address to SQLDA. */
        SQLDA->SQLVAR[param].SQLDATA = (char *) smallintbuf;

        break;  /* Exit the switch block. */

     case 503 :                          /* Date VMS variable */

        printf ("\nDate/time format is DD-MMM-YYYY HH:MM:SS.HH ");
        printf ("\ndsql> ");

        /* Allocate storage for the character string version of
           the date and time to be entered.     */

        charbuf = (char *) malloc (23);

        /* Get the string. */
        gets (charbuf);

        /* Use the CAST statement to convert the input character string
           to the DATE VMS data type. */
        EXEC SQL
           BEGIN SELECT CAST(substring(:charbuf FROM 8 FOR 4) ||
                -- Convert the month to a number.
                   (CASE SUBSTRING(:charbuf FROM 4 FOR 3)
                           WHEN 'JAN' THEN '01'
                           WHEN 'FEB' THEN '02'
                           WHEN 'MAR' THEN '03'
                           WHEN 'APR' THEN '04'
                           WHEN 'MAY' THEN '05'
                           WHEN 'JUN' THEN '06'
                           WHEN 'JUL' THEN '07'
```

(continued on next page)

Example 11–3 (Cont.) Executing Non-SELECT Statements with Parameter Markers

```
                        WHEN 'AUG' THEN '08'
                        WHEN 'SEP' THEN '09'
                        WHEN 'OCT' THEN '10'
                        WHEN 'NOV' THEN '11'
                        WHEN 'DEC' THEN '12'
                    END) ||
                -- Parse the day, hour, minutes, seconds.
                        SUBSTRING(:charbuf FROM 1 FOR 2)  ||
                        SUBSTRING(:charbuf FROM 13 FOR 2) ||
                        SUBSTRING(:charbuf FROM 16 FOR 2) ||
                        SUBSTRING(:charbuf FROM 19 FOR 2) ||
                        SUBSTRING(:charbuf FROM 22 for 2)
                AS DATE VMS) into :my_date_vms
                FROM rdb$database LIMIT TO 1 ROW;
            END;

           if (SQLCODE != 0)
             goto err;

         /* Assign the address to SQLDA. */
         SQLDA->SQLVAR[param].SQLDATA = (char *) &my_date_vms;

         break;  /* Exit the switch block. */

       default :

         printf("\n\nError: no data type match on %d",
                        SQLDA->SQLVAR[param].SQLTYPE);
         break;

       }  /* End switch. */

   }  /* End the for loop. */

/* Execute the statement and replace the parameter markers with the
   values in the SQLDA. */
    EXEC SQL EXECUTE STMT_NAME USING DESCRIPTOR SQLDA;
     if (SQLCODE != 0)
       goto err;

    EXEC SQL COMMIT;

    return;

err:    printf("\n unexpected error %d", SQLCODE);
        sql_signal();

}
```

The following callouts are keyed to Example 11–3:

❶ Declares the SQLDA structure.

❷ Allocates the SQLDA structure.

❸ Prompts the user for an SQL command to be executed.

❹ Prepares the statement that the user entered. If an error is returned, branch to an error-handling routine.

❺ Uses the DESCRIBE . . . MARKERS statement to write information about parameter markers into the SQLDA structure.

❻ Sets up a loop that executes the same number of times as the value stored in the SQLD field, that is, once for each parameter marker in the statement string.

❼ In a case statement, tests the value of the SQLTYPE field in the SQLDA to determine the data type of the parameter marker. Based on the information in the SQLTYPE field, the program uses the malloc( ) function to allocate storage of the right length. It prompts the user for a column value and assigns that address to the SQLDA.

❽ Changes the data type of string data from character or varying character strings to null-terminated strings.

11.5 Testing Whether or Not a Statement Is a SELECT Statement

Example 11–3 assumes that the user will not enter a SELECT statement. Often, your program cannot make that assumption.

The second element of the SQLERRD array in the SQLCA structure (described in Chapter 10) indicates whether the prepared statement is a SELECT statement. If the value of the second element of the SQLERRD array is 0, the statement string is not a SELECT statement and you can execute it directly. If the value is 1, the statement string is a SELECT statement. Section 11.6 explains how to process SELECT statements.

Example 11–4, an excerpt from the sql_dynamic program, shows how a C program might test the second element of the SQLERRD array. If the statement is not a SELECT statement, the program calls an EXECUTE statement in an SQL module language procedure.

Example 11–4 Testing SQLERRD to Identify Non-SELECT Statements

```
   .
   .
   .
        /* Save the value of the SQLCA.SQLERRD[1] field so that the program
           can determine if the statement is a SELECT statement or not.
           If the value is 1, the statement is a SELECT statement.*/

           *is_select = SQLCA.SQLERRD[1];
   .
   .
   .
            if (*is_select)
   .
   .
   .
        }   /* end if SQLCA.SQLERRD[1] == 1) */
    else
        {
   .
   .
   .
/*
** If the SQLCA.SQLERRD[1] field is not 1, then the prepared statement is not a
** SELECT statement and only needs to be executed.  Call an SQL module language
** procedure to execute the statement, using information about parameter
** markers stored in sqlda_in by the local routine get_in_params:
*/
/*       if (SQLCA.SQLERRD[1] != 1)*/

        {
        execute_stmt (&SQLCA, stmt_id, sqlda_in);
        if (SQLCA.SQLCODE != sql_success)
            {
            printf("\n\nError %d returned from execute_stmt",SQLCA.SQLCODE);
            display_error_message();
            return (-1);
            }
        } /* end if SQLCA.SQLERRD[1] != 1  */
        }  /* end if SQLCA.SQLERRD[1] == 1 */
```

11.6 Processing SELECT Statements

To execute SELECT statements, whether or not they contain parameter markers, programs must declare dynamic cursors or extended dynamic cursors. The cursor receives the values of rows in the result table created by the SELECT statement.

A **dynamic cursor** specifies the cursor name at compile time; however, it does not explicitly specify the SELECT statement. Instead, you supply the name of a prepared statement for the SELECT statement. Because the dynamic DECLARE CURSOR statement lets your program supply the

SELECT statement at run time, you can use the dynamic DECLARE CURSOR statement to process an arbitrary number of dynamically generated SQL statements.

An **extended dynamic cursor** does not specify either the cursor name or the statement name at compile time. Instead, you supply parameters for the cursor name and for the SELECT statement. Because the dynamic DECLARE CURSOR statement lets your program supply this information at run time, you can use this dynamic statement to process an arbitrary number of dynamically generated SQL statements concurrently.

In addition to setting up and using a dynamic or extended dynamic cursor, your program must prepare the SELECT statement (using the PREPARE statement), then use the DESCRIBE . . . SELECT LIST statement to write the information about the number and data type of select list items to the SQLDA.

The program uses the information in the SQLDA to allocate storage for the select list items. See Section 11.3.1 and Section 11.6.3 for more information about declaring and using SQLDAs.

If any parameters might return NULL values, your program also needs to declare and use indicator parameters (explained in Chapter 8).

After your program opens the cursor for the SELECT statement, use FETCH statements to return to host language variables the values in each row in the SQLDA. (Note that the FETCH statement itself cannot be dynamically executed.)

11.6.1 Executing SELECT Statements Without Parameter Markers: Declaring Dynamic and Extended Dynamic Cursors

Example 11–5 shows an SQL module, c_mod_dyn_curs, that dynamically executes a SELECT statement that does not contain parameter markers. The module declares a dynamic cursor to specify the result table that holds the values of the rows.

A host language program that calls this SQL module is shown in Example 11–6.

Example 11–5 Executing SELECT Statements Without Parameter Markers in an SQL Module

```
--  This program uses dynamic cursors to fetch a row from a table.
--
MODULE          C_MOD_DYN_CURS
LANGUAGE        GENERAL
AUTHORIZATION   RDB$DBHANDLE
PARAMETER COLONS
DECLARE ALIAS FILENAME personnel

-- Declare the dynamic cursor. Use a statement name to identify a
-- prepared SELECT statement.
DECLARE CURSOR1 CURSOR FOR STMT_NAME

-- Prepare the statement from a statement entered at run time
-- and specify that SQL write information about the number and
-- data type of select list items to the SQLDA.

PROCEDURE PREP_STMT
    (SQLCODE,
     :COMMAND_STRING  CHAR (256));

      PREPARE STMT_NAME FROM :COMMAND_STRING;

PROCEDURE DESCRIBE
    (SQLCODE,
     SQLDA);

      DESCRIBE STMT_NAME SELECT LIST INTO SQLDA;

PROCEDURE OPEN_CURSOR
    (SQLCODE);

    OPEN CURSOR1;

PROCEDURE FETCH_CURSOR
    (SQLCODE,
     SQLDA);

    FETCH CURSOR1 USING DESCRIPTOR SQLDA;

PROCEDURE CLOSE_CURSOR
    (SQLCODE);

    CLOSE CURSOR1;

PROCEDURE ROLLBACK
    (SQLCODE);

    ROLLBACK;
```

Example 11–6 shows the program that calls the routines in the SQL C_MOD_DYN_CURS module. When you specify GENERAL in the language clause as in Example 11–5, you must pad the string passed to the prepare routine (command_string[256]) with spaces because SQL, with the GENERAL keyword specified, does not look for the NULL terminator in character strings that are passed as parameters. Otherwise, SQL generates an error.

In contrast, if you specify C in the LANGUAGE clause of an SQL module, no field padding is necessary because the SQL module looks for the NULL terminator in character strings that are passed as parameters.

Example 11–6 Executing SELECT Statements Without Parameter Markers in a Host Language Program

```
/*  This program uses dynamic cursors to fetch a row from a table. */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql_rdb_headers.h>

/* Maximum number of parameters or select list items is 50.*/
#define MAXPARAMS 50

/* Declare the SQLDA structure. */
struct SQLDA_STRUCT {
          char SQLDAID[8];
          int SQLDABC;
          short SQLN;
          short SQLD;
          struct {
            short SQLTYPE;
            short SQLLEN;
            char *SQLDATA;
            int  *SQLIND;
            short SQLNAME_LEN;
            char SQLNAME[30];
              } SQLVAR[MAXPARAMS];
        } *SQLDA;

main()
{

/* General purpose local variables.  */
int     i;
long    sqlcode;
char    command_string[256];

/* Function prototypes. */
void PREP_STMT ();
void DESCRIBE ();
void OPEN_CURSOR ();
void FETCH_CURSOR ();
void CLOSE_CURSOR ();
void ROLLBACK ();

/* Allocate SQLDA structures.  */
SQLDA = malloc(534);
SQLDA->SQLN = 20;
```

(continued on next page)

Example 11–6 (Cont.) Executing SELECT Statements Without Parameter Markers in a Host Language Program

```
/* Get the SELECT statement at run time. */

printf("\n Enter a select statement that retrieves only character ");
printf("\n and integer data. \n");
printf("\n Do not end the statement with a semicolon.\n");
gets(command_string);

/* Pad the string to end with spaces when using LANGUAGE keyword GENERAL
   in module. */

for (i=strlen(command_string);i<=256;i++)
    command_string[i] = ' ';

/* Prepare the SELECT statement. */
PREP_STMT( &sqlcode, &command_string);
if (sqlcode != 0)
    goto err;

/* Write the information about the number and data type of the select list
   items to the SQLDA.  */
DESCRIBE( &sqlcode, SQLDA);
if (sqlcode != 0)
    goto err;

/* Open the cursor. */
OPEN_CURSOR( &sqlcode );
if (sqlcode != 0)
    goto err;

/* Allocate memory.  */
for (i=0; i < SQLDA->SQLD; i++) {
    SQLDA->SQLVAR[i].SQLDATA = malloc( SQLDA->SQLVAR[i].SQLLEN);
    SQLDA->SQLVAR[i].SQLIND  = malloc( 2 );
}

/* Fetch a row. */
FETCH_CURSOR( &sqlcode, SQLDA );
if (sqlcode != 0)
    goto err;

/* Use the SQLDA to determine the data type of each column in the row
   and print the column.  For simplicity, test for only two data types. */

for (i=0; i < SQLDA->SQLD; i++) {

    switch (SQLDA->SQLVAR[i].SQLTYPE) {

    case 453:   /* Character */
        printf( "%.*s ",SQLDA->SQLVAR[i].SQLLEN, SQLDA->SQLVAR[i].SQLDATA );
        break;
```

(continued on next page)

Example 11–6 (Cont.) Executing SELECT Statements Without Parameter Markers in a Host Language Program

```
    case 497:   /* Integer */
        printf( "%d", *SQLDA->SQLVAR[i].SQLDATA );
        break;

    }
}

/* Close the cursor. */
CLOSE_CURSOR( &sqlcode );

ROLLBACK( &sqlcode );
return;

err:    printf("\n unexpected error %d", sqlcode);
        sql_signal();
        ROLLBACK( &sqlcode );
        exit (0);
}
```

11.6.2 Executing SELECT Statements That Contain Parameter Markers

In addition to using dynamic or extended dynamic cursors, SELECT statements that contain parameter markers require you to declare and use two SQLDA structures, one for the parameter markers and one for the select list items.

You must use both the DESCRIBE . . . MARKERS statement and the DESCRIBE . . . SELECT LIST statement. The DESCRIBE . . . MARKERS statement writes information about parameter markers into the SQLDA; the DESCRIBE . . . SELECT LIST statement writes information about select list items into the SQLDA.

Your program uses the information in the SQLDA to allocate storage for the select list items and parameter markers. The program must supply values for parameter markers in that allocated storage. SQL substitutes these values for the parameter markers when it dynamically executes the statement.

Example 11–7 shows a program, c_dyn_extcurs.sc, that uses precompiled SQL to dynamically execute SELECT statements that contain parameter markers. This program uses extended dynamic cursors to process any number of dynamically generated SELECT statements.

Example 11–7 Executing SELECT Statements with Parameter Markers in an SQL Precompiled Program

```
/* This program dynamically executes SELECT statements that contain
   parameter markers. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql_rdb_headers.h>

/* Maximum number of parameters or select list items is 50.*/
#define MAXPARAMS 50

/* Declare two SQLDA structures -- one for the parameter markers and one
   for the select list items.  */
struct SQLDA_STRUCT {
          char SQLDAID[8];
          int SQLDABC;
          short SQLN;
          short SQLD;
          struct {
            short SQLTYPE;
            short SQLLEN;
            char *SQLDATA;
            int  *SQLIND;
            short SQLNAME_LEN;
            char SQLNAME[30];
              } SQLVAR[MAXPARAMS];
        } *SQLDA, *SQLDA_SL;
main()
{

    long           SQLCODE;
    short          param, *indicator_param;
    int            loop_cntr;
    int            X;
    char           column_name[31];
    char           curs_name[10];          /* Name of statement */

/* An array to hold two SQL statements. */
   char           command_string[2][256];
   char           *command_string1;


/* Declare pointer variables for each data type.  The program uses
   the pointers to dynamically allocate memory to hold values for
   parameter markers and select list items.  */
    char          *charbuf;
    long          *longbuf;
    short         *smallintbuf;
    char          *datebuf;
    SQL_DATE_VMS  my_date_vms;
```

(continued on next page)

Example 11–7 (Cont.) Executing SELECT Statements with Parameter Markers in an SQL Precompiled Program

```
/* Allocate SQLDA structures.  */

SQLDA_SL = malloc(534);
SQLDA_SL->SQLN = 12;
SQLDA_SL = malloc(534);
SQLDA_SL->SQLN = 12;

/* Declare the alias. */
    EXEC SQL DECLARE ALIAS FILENAME personnel;

/* Execute a loop to get two SELECT statements from the user. */

X=0;
while ( X < 2 )
{
    printf("Enter any SELECT statement. \n");
    printf("Do not end the statement with a semicolon. \n");

    gets(&command_string[X][0]);

/* Assign a statement name to each SELECT statement. */
     if (X == 0)
       strcpy(curs_name, "ONE");
     else
      if (X == 1)
       strcpy(curs_name, "TWO");
     else printf("\n No such statement. \n");

       X++;
}

/* Prompt the user for the name of a statement. */
     printf("\n Which statement do you want to execute, \n");
     printf("ONE or TWO?  To quit, type EXIT.\n");

     gets(curs_name);

/* Assign the correct SQL statement to the cursor name. */
    if (strcpy(curs_name, "ONE" ))
       command_string1 = &command_string[0][0];
     if (curs_name == "TWO" )
       command_string1 = &command_string[1][0];
    if (curs_name == "EXIT")
       goto end_program;

/* Start an SQL transaction. */
    EXEC SQL SET TRANSACTION READ WRITE;
     if (SQLCODE != 0)
       goto err;

/* Prepare the statement from input entered at run time. */
    EXEC SQL PREPARE stmt_id FROM :command_string1;
     if (SQLCODE != 0)
       goto err;
```

(continued on next page)

Example 11–7 (Cont.) Executing SELECT Statements with Parameter Markers in an SQL Precompiled Program

```
/* Write information about the parameter markers into the SQLDA. */
    EXEC SQL DESCRIBE stmt_id MARKERS INTO SQLDA;
     if (SQLCODE != 0)
       goto err;

/* Write information about the select list items into the SQLDA. */
    EXEC SQL DESCRIBE stmt_id SELECT LIST INTO SQLDA_SL;
     if (SQLCODE != 0)
       goto err;

/* For each parameter marker, check the value of SQLDA.SQLVAR.SQLTYPE
   to determine the data type of the parameter marker, and branch to
   the appropriate code to prompt and store the value for that data type. */

    param = 0;
    for (param = 0; param < SQLDA->SQLD; param++)
      {

      /* Handle a null-terminated column name and prompt with the column name. */
      strncpy (&column_name[0], &SQLDA->SQLVAR[param].SQLNAME[0],
                                SQLDA->SQLVAR[param].SQLNAME_LEN);
      column_name[SQLDA->SQLVAR[param].SQLNAME_LEN] = '\0';

      printf ("\nEnter value for parameter '%s'", column_name);

      switch (SQLDA->SQLVAR[param].SQLTYPE)
        {
        case 453 :                          /* character variable */
        case 449 :                          /* variable char variable */

          printf("\n(Maximum length is %d)",SQLDA->SQLVAR[param].SQLLEN);
          printf("\ndsql> ");
          /* Allocate storage and get the address for the character
             string. */
          charbuf = (char *) malloc (SQLDA->SQLVAR[param].SQLLEN);

          /* Get the string. */
          gets (charbuf);

          /* Assign the address to SQLDA. */
          SQLDA->SQLVAR[param].SQLDATA = charbuf;

          break;  /* Exit the switch block */

        case 497 :                          /* integer variable */

          printf ("\ndsql> ");

          /* Allocate storage for the character string version of
             the integer to be entered.  */
          charbuf = (char *) malloc (14);

          /* Get the string. */
          gets (charbuf);
```

(continued on next page)

Example 11–7 (Cont.) Executing SELECT Statements with Parameter Markers in an SQL Precompiled Program

```
        /* Allocate storage for the integer and convert the string. */
        longbuf  = (long *) malloc (SQLDA->SQLVAR[param].SQLLEN);
        *longbuf = atol (charbuf);

        /* Assign the address to SQLDA */
        SQLDA->SQLVAR[param].SQLDATA = (char *) longbuf;

        break;  /* Exit the switch block */

      case 501 :                          /* short int variable */

        printf ("\ndsql> ");

        /* Allocate storage for the character string version of
           the short integer to be entered.*/
        charbuf = (char *) malloc (5);
        /* Get the string. */
        gets (charbuf);

        /* Allocate storage for the short int and convert the string. */
        smallintbuf  = (short *) malloc(SQLDA->SQLVAR[param].SQLLEN);
        *smallintbuf = atoi (charbuf);
        /* Assign the address to SQLDA. */
        SQLDA->SQLVAR[param].SQLDATA = (char *) smallintbuf;

        break;  /* Exit the switch block */

      case 503 :                          /* date type variable */

        printf ("\nDate-time format is DD-MMM-YYYY HH:SS:MM.CC ");
        printf ("\ndsql> ");

        /* Allocate storage for the character string version of
           the date-time.   */

        charbuf = (char *) malloc (23);
        /* Get the string. */
        gets (charbuf);

  EXEC SQL SELECT
       CAST(substring(:charbuf FROM 8 FOR 4) ||
        -- Convert the month to a number.
          (CASE SUBSTRING(:charbuf FROM 4 FOR 3)
                  WHEN 'JAN' THEN '01'
                  WHEN 'FEB' THEN '02'
                  WHEN 'MAR' THEN '03'
                  WHEN 'APR' THEN '04'
                  WHEN 'MAY' THEN '05'
                  WHEN 'JUN' THEN '06'
                  WHEN 'JUL' THEN '07'
                  WHEN 'AUG' THEN '08'
```

(continued on next page)

Example 11–7 (Cont.) Executing SELECT Statements with Parameter Markers in an SQL Precompiled Program

```
                WHEN 'SEP' THEN '09'
                WHEN 'OCT' THEN '10'
                WHEN 'NOV' THEN '11'
                WHEN 'DEC' THEN '12'
             END) ||
     -- Parse the day, hour, minutes, seconds.
       SUBSTRING(:charbuf FROM 1 FOR 2)  ||
       SUBSTRING(:charbuf FROM 13 FOR 2) ||
       SUBSTRING(:charbuf FROM 16 FOR 2) ||
       SUBSTRING(:charbuf FROM 19 FOR 2) ||
       SUBSTRING(:charbuf FROM 22 for 2)
       AS DATE VMS) into :my_date_vms
       FROM rdb$database LIMIT TO 1 ROW;

         /* Assign the address to SQLDA. */
         SQLDA->SQLVAR[param].SQLDATA = (char *) &my_date_vms;

         break;  /* Exit the switch block. */

       default :

         printf("\n\nError: no data type match on %d",
                         SQLDA->SQLVAR[param].SQLTYPE);
         break;

       }  /* End switch. */

   }  /* End the for loop. */

/* Declare the extended dynamic cursor, using the name the user enters. */
     EXEC SQL DECLARE curs_name CURSOR FOR stmt_id;
     if (SQLCODE != 0)
       goto err;

/* Open the cursor. */
     EXEC SQL OPEN curs_name USING DESCRIPTOR SQLDA;
     if (SQLCODE != 0)
       goto err;

/* Allocate memory. */

for (param=0; param < SQLDA_SL->SQLD; param++) {
    SQLDA_SL->SQLVAR[param].SQLDATA = malloc( SQLDA_SL->SQLVAR[param].SQLLEN );
    SQLDA_SL->SQLVAR[param].SQLIND  = malloc( 2 );
}
/* Fetch the cursor and get information about the select list items
   from the SQLDA. */
     EXEC SQL FETCH curs_name USING DESCRIPTOR SQLDA_SL;
     if (SQLCODE != 0)
       goto err;
```

(continued on next page)

Example 11–7 (Cont.) Executing SELECT Statements with Parameter Markers in an SQL Precompiled Program

```
/* Allocate storage for select list items in the statement string supplied
   by the user.  Also, allocate storage for indicator parameters associated
   with the select list items, which indicate the occurrence of a NULL
   value being returned for an item in the database.   */

    /* For each select list item, execute a loop. */

    for (param = 0; param < SQLDA_SL->SQLD; param++)
        {

        /* Allocate storage for an indicator array. */
        indicator_param = (short *) malloc (sizeof (short));

        /* Write the address of indicator_param to SQLDA_SL.SQLVAR.SQLIND */
        SQLDA_SL->SQLVAR[param].SQLIND = indicator_param;

        /* Check the value of SQLDA_SL.SQLVAR.SQLTYPE to determine the data
           type of the select list item and branch to the allocation block
           for that data type.  */

      switch (SQLDA_SL->SQLVAR[param].SQLTYPE)
        {
        case 453 :                          /* character variable */
        case 449 :                          /* variable char variable */

          /* Allocate storage and get address for the character string. */
          charbuf = (char *) malloc (SQLDA_SL->SQLVAR[param].SQLLEN);

          charbuf = (char *) malloc(SQLDA_SL->SQLVAR[param].SQLLEN+1);

          /* Copy the value to a string with a null terminator. */
          strncpy (charbuf, SQLDA_SL->SQLVAR[param].SQLDATA,
                          SQLDA_SL->SQLVAR[param].SQLLEN);

          *(charbuf + SQLDA_SL->SQLVAR[param].SQLLEN) = '\0';

          /* Display the value in the buffer. */
          printf ("%s  ", charbuf);

          break;  /* Exit the switch block. */

        case 497 :                          /* integer variable */

          /* Allocate storage for the integer. */
          longbuf  = (long *) malloc (SQLDA_SL->SQLVAR[param].SQLLEN);

          printf ("%ld  ", (long)(*SQLDA_SL->SQLVAR[param].SQLDATA));

          break;  /* Exit the switch block. */

        case 501 :                          /* short int variable */
```

(continued on next page)

Example 11–7 (Cont.) Executing SELECT Statements with Parameter Markers in an SQL Precompiled Program

```
          /* Allocate storage for the short int. */
          smallintbuf  = (short *) malloc (SQLDA_SL->SQLVAR[param].SQLLEN);

          printf ("%d  ", (short)(*SQLDA_SL->SQLVAR[param].SQLDATA));

          break;  /* Exit the switch block. */

        case 503 :                          /* date type variable */
         memcpy( &my_date_vms,SQLDA_SL->SQLVAR[param].SQLDATA,
                               sizeof( SQL_DATE_VMS ) );

         datebuf = (char *) malloc (24);
         charbuf = (char *) malloc (24);

         EXEC SQL
             BEGIN
               SET :datebuf  = CAST(:my_date_vms AS CHAR(24));
             END;

         if (SQLCODE != 0)
         goto err;

         printf ("%s  ", datebuf);

          break;  /* Exit the switch block. */

         }   /* End switch. */

        }    /* End the for loop. */

/* Close the cursor. */
     EXEC SQL CLOSE curs_name;
     if (SQLCODE != 0)
       goto err;

    EXEC SQL COMMIT;

    return;

err:    printf("\n unexpected error %d", SQLCODE);
        sql_signal();

end_program:
        return;
}
```

11.6.3 Using SQLDA2 and SQLERRD Structures to Test for Parameter Markers and SELECT Statements

The sql_dynamic sample program uses the SQLDA2 and SQLERRD structures to determine what type of processing the statement string entered by the user requires. Example 11–8 shows the logic from the main program.

Example 11–8 Testing Whether a Statement Is a SELECT Statement

```
/* Declare arrays for storage of original data types and allocate memory. */

    mem_ptr output_save;
    mem_ptr input_save;


    /* If NULL sqlda was passed, then a statement is being prepared. */

    if ((*input_sqlda == NULL) && (*output_sqlda == NULL))
         {
         new_statement = TRUE;

         /* Allocate separate SQLDAs for parameter markers (SQLDA_IN) and select
            list items (SQLDA_OUT).  Assign value of the constant MAXPARAMS to
            the SQLN field of both SQLDA structures.  SQLN specifies to SQL the
            maximum size of the SQLDA.       */                                  ❶

        if ((sqlda_in = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
            {
            printf ("\n\n*** Error allocating memory for sqlda_in: Abort");
            return (-1);
            }
        else    /* Set number of possible parameters. */
            sqlda_in->sqln = MAXPARAMS;

        if ((sqlda_out = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
            {
            printf ("\n\n*** Error allocating memory for sqlda_out: Abort");
            return (-1);
            }
        else
            /* Set number of possible select list items. */
            sqlda_out->sqln = MAXPARAMS;

        /* Copy name SQLDA2 to identify the SQLDA. */

        strncpy(&sqlda_in->sqldaid[0],"SQLDA2  ",8);
        strncpy(&sqlda_out->sqldaid[0],"SQLDA2  ",8);

        /* Call the SQL module language procedures prepare_stmt and
           describe_select which contain PREPARE and DESCRIBE...SELECT LIST
           statements to prepare the dynamic statement and write
           information about any select list items in it to sqlda_out.  */

        *stmt_id = 0;  /* if <> 0 the BADPREPARE error results in the PREPARE*/

        prepare_stmt (&SQLCA, stmt_id, psql_stmt);                             ❷
        if (SQLCA.SQLCODE  != sql_success)
            {
            printf ("\n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
                        SQLCA.SQLCODE);
            display_error_message();
            return (-1);
            }
```

(continued on next page)

Example 11–8 (Cont.) Testing Whether a Statement Is a SELECT Statement

```
    describe_select (&SQLCA, stmt_id, sqlda_out);                    ❸
    if (SQLCA.SQLCODE  != sql_success)
        {
        printf ("\n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
                    SQLCA.SQLCODE);
        display_error_message();
        return (-1);
        }

  /* Call an SQL module language procedure, describe_parm, which contains a
     DESCRIBE...MARKERS statement to write information about any parameter
     markers in the dynamic statement to sqlda_in.   */

    describe_parm (&SQLCA, stmt_id, sqlda_in);                       ❹
    if (SQLCA.SQLCODE  != sql_success)
        {
        printf ("\n\n*** Error %d returned from describe_parm: Abort",
                 SQLCA.SQLCODE);
        display_error_message();
        return (-1);
        }

    /* Save the value of the SQLCA.SQLERRD[1] field so that the program
       can determine if the statement is a SELECT statement or not.
       If the value is 1, the statement is a SELECT statement.*/

       *is_select = SQLCA.SQLERRD[1];
.
.
.
 /* Check to see if the prepared dynamic statement contains any parameter
    markers by looking at the SQLD field of sqlda_in.  SQLD contains the
    number of parameter markers in the prepared statement. If SQLD is
    positive, the prepared statement contains parameter markers.  The program
    executes a local routine, get_in_params, that prompts the user for
    values, allocates storage for those values, and updates the SQLDATA field
    of sqlda_in.      */

 if (sqlda_in->sqld > 0)                                             ❺
     if ((status = get_in_params(sqlda_in,input_save)) != 0)
         {
         printf ("\nError returned from get_in_params. Abort");
         return (-1);
         }

 /* Check to see if the prepared dynamic statement is a SELECT statement
    by looking at the value in is_select, which stores the value of
    the SQLCA.SQLERRD[1] field.  If that value is equal to 1, the
    the prepared statement is a SELECT statement.  The program allocates
    storage for rows for SQL module language procedures to open and
    fetch from a cursor and displays the rows on the terminal.   */
```

(continued on next page)

Example 11–8 (Cont.) Testing Whether a Statement Is a SELECT Statement

```
    if (*is_select)                                                    ❻
        {
        if (new_statement == TRUE)      /* allocate buffers for output */
            {
            /* assign a unique name for the cursor */
            sprintf(cursor_name,"%2d",++cursor_counter);

            if ((status = allocate_buffers(sqlda_out)) != 0)
   .
   .
   .

/* If the SQLCA.SQLERRD[1] field is not 1, the prepared statement is not a
   SELECT statement and only needs to be executed.  Call an SQL module language
   procedure to execute the statement, using information about parameter
   markers stored in sqlda_in by the local routine get_in_params.  */

       if (SQLCA.SQLERRD[1] != 1)
        {
        execute_stmt (&SQLCA, stmt_id, sqlda_in);
   .
   .
   .
```

The following callouts are keyed to the Example 11–8:

❶ Allocates storage for the declared SQLDA_IN and SQLDA_OUT structures.

❷ Calls a PREPARE statement in the SQL module procedure to check the SQL statement and assign an identifier to it.

❸ Calls a DESCRIBE . . . SELECT LIST statement in the SQL module procedure to write information about any select list items in the statement string to SQLDA_OUT.

❹ Calls a DESCRIBE . . . MARKERS statement in an SQL module procedure to write information about any parameter markers in the statement string to SQLDA_IN.

❺ Checks the value of SQLDA_IN.SQLD. If the value is positive, the statement string contains parameter markers, and the program branches to the GET_IN_PARAMS subroutine. If the value is not positive, the statement string contains no parameter markers, and program control does not branch.

❻ Checks the value of the second element of the SQLERRD array. If the value is 1, the statement string is a SELECT statement, and the program branches to the ALLOCATE BUFFERS subroutine. If the value is 0, the

statement string is not a SELECT statement (or CALL statement), and the program calls an EXECUTE statement to process it.

11.7 Processing Sets of Dynamically Generated Statements

You can use a single set of dynamic SQL statements, such as PREPARE, DESCRIBE, extended dynamic DECLARE CURSOR, OPEN, and FETCH, to concurrently process any number of dynamically generated statements by specifying parameters for the statement and cursor names in the associated dynamic SQL statements. By using parameters instead of explicit statement names, a program can supply statement and cursor names at run time, instead of at compile time, and process an arbitrary number of dynamically generated SQL statements.

For non-SELECT statements, you can use an integer parameter as a statement identifier instead of an explicit statement name in the PREPARE statement. The PREPARE statement returns a value for the statement identifier that you can pass to the EXECUTE statement.

For SELECT statements as for non-SELECT statements, you can use a parameter as a statement identifier instead of an explicit statement name in the PREPARE statement. You pass the prepared statement to an extended dynamic DECLARE CURSOR statement by using the statement identifier. The extended dynamic DECLARE CURSOR statement lets you specify a parameter for the cursor name instead of explicitly declaring the name of the cursor in your source code. You supply both the cursor name and SELECT statement at run time.

To process an arbitrary number of statements at run time, you must save the values of the parameters for the statement identifier and the cursor name so that you can refer to a previously prepared statement or previously declared cursor. You can set up arrays to store the values of these parameters in a data structure. Set up one array to hold the value of the statement identifier parameter and assign the value of the parameter to an array element after each PREPARE or extended dynamic DECLARE CURSOR statement. If you are processing SELECT statements, you must also store the value of the cursor name parameter in an array. Assign the value of that parameter to an array element after each extended dynamic DECLARE CURSOR statement.

The following sections use excerpts from the sql_multi_stmt_dyn.sqlada sample program, which demonstrates the use of parameters for statement and cursor names. The program constructs SQL statements from user input at run time. Using a single set of dynamic SQL statements, the program can process many dynamically generated statements.

11.7.1 Storing Statement Identifiers and Cursor Names

Example 11–9 shows sections of code from the sql_multi_stmt_dyn.sqlada sample program. It shows how you can declare arrays for statement identifiers and cursor names and assign the value of the statement identifiers and cursor name parameters to elements of the arrays. In addition, it shows arrays that the sample program uses to keep track of each statement and its associated parameters.

Example 11–9 Storing Statement Identifiers and Cursor Names in Arrays

```
-- Declare tables of procedure names, identifiers, and cursor names,
-- a counter, and an index.

-- The array PROCEDURE_NAMES holds the name that the user gives each
-- statement. You may decide to keep track of the statements in a
-- different way. The array PROCEDURE_IDS holds the statement
-- identifiers generated by the PREPARE statement. The array
-- CURSOR_NAMES holds the cursor names.

    PROCEDURE_NAMES : array(1..maxprocs) of string(1..name_strlng);
    PROCEDURE_IDS : array(1..maxprocs) of integer;
    CURSOR_NAMES : array(1..maxprocs) of string(1..name_strlng);

-- The NUMBER_OF_PROCS procedure increments the array elements when
-- the program stores information.  The STMT_INDEX procedure increments
-- the array elements when the program looks up information.

    NUMBER_OF_PROCS : short_integer := 0;
    STMT_INDEX : short_integer := 0;
   .
   .
   .
-- This section of the program stores the values of the parameters for
-- statement identifiers and cursor names in arrays.

number_of_procs := number_of_procs + 1;

-- Assign information about parameter markers and select list items to
-- array elements.

sqlda_in_array(number_of_procs) := sqlda_in;
sqlda_out_array(number_of_procs) := sqlda_out;

-- Assign the user-specified statement name to an array element. Assign
-- the statement identifier returned by the PREPARE statement to an
-- array element.  In addition, if the statement is a SELECT statement,
-- assign the cursor name to an array element.
```

(continued on next page)

Example 11–9 (Cont.) Storing Statement Identifiers and Cursor Names in Arrays

```
procedure_names(number_of_procs) := cur_name;
procedure_ids(number_of_procs) := cur_procid;
if cur_op(1) = 'R' then
    cursor_names(number_of_procs) := cur_cursor;
end if;
```

11.7.2 Executing Multiple Non-SELECT Statements

To process more than one non-SELECT statement with a single set of dynamic SQL statements, use a parameter in place of the statement name in the PREPARE statement. The parameter has an integer data type. The PREPARE statement returns a statement identifier in this parameter, which you pass to the DESCRIBE . . . MARKERS and EXECUTE statements.

Example 11–10, an excerpt from the sample sql_multi_stmt_dyn.sqlada program, shows how you can use one set of dynamic SQL statements to process any number of non-SELECT statements.

The PREPARE_SQL procedure contains a PREPARE statement, a DESCRIBE . . . SELECT LIST statement, and a DESCRIBE . . . MARKERS statement. The program uses the PREPARE_SQL procedure to prepare both SELECT and non-SELECT statements, in contrast to the EXECUTE_SQL procedure, which executes only non-SELECT statements. If you expect the PREPARE statement to process only non-SELECT statements, you do not need to include the DESCRIBE . . . SELECT LIST statement.

Example 11–10 Executing More Than One Non-SELECT Statement

```
-- This procedure prepares a statement for dynamic execution from the
-- string passed to it.  It can prepare any number of statements because
-- the statement is passed to it as the parameter cur_stmt.

procedure PREPARE_SQL is
    CUR_CURSOR : string(1..31) := (others => ' ');
    CUR_PROCID : integer := 0;
    CUR_STMT : string(1..1024) := (others => ' ');
```

(continued on next page)

Example 11–10 (Cont.) Executing More Than One Non-SELECT Statement

```
begin
-- Allocate separate SQLDAs for parameter markers (sqlda_in) and select list
-- items (sqlda_out).  Assign the value of the constant MAXPARMS (set in the
-- declarations section) to the SQLN field of both SQLDA structures. SQLN
-- specifies to SQL the maximum size of the SQLDA.

sqlda_in := new sqlda_record;
sqlda_in.sqln := maxparms;
sqlda_out := new sqlda_record;
sqlda_out.sqln := maxparms;

-- Assign the SQL statement that was constructed in the procedure
-- CONSTRUCT_SQL to the variable cur_stmt.

cur_stmt := sql_stmt;

-- Use the PREPARE statement to prepare the dynamic statement
-- for dynamic execution from the string passed to it.

EXEC SQL PREPARE :cur_procid  FROM :cur_stmt;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;

-- Use the DESCRIBE ... SELECT LIST statement to write information
-- about the number and data type of any select list items in the
-- statement to an SQLDA (specifically, the sqlda_out SQLDA specified).

EXEC SQL DESCRIBE :cur_procid SELECT LIST INTO :sqlda_out;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;


-- Use the DESCRIBE...MARKERS statement to write information about any
-- parameter markers in the dynamic statement to sqlda_in.  The statement
-- writes information to an SQLDA (specifically, the sqlda_in SQLDA
-- specified) about the number and data type of any parameter markers in
-- the prepared dynamic statement.  SELECT statements may also have
-- parameter markers.

EXEC SQL DESCRIBE :cur_procid MARKERS INTO sqlda_in;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;
   .
   .
   .
procedure EXECUTE_SQL is
```

(continued on next page)

Example 11–10 (Cont.) Executing More Than One Non-SELECT Statement

```
begin
--
-- This procedure dynamically executes a non-SELECT statement.
-- SELECT statements are processed by DECLARE CURSOR, OPEN CURSOR,
-- and FETCH statements.
--
-- The EXECUTE statement specifies an SQLDA (specifically, the sqlda_in
-- SQLDA specified) as the source of addresses for any parameter
-- markers in the dynamic statement.
--
-- The EXECUTE statement with the USING DESCRIPTOR clause also
-- handles statement strings that contain no parameter markers.
-- If a statement string contains no parameter markers, SQL sets
-- the SQLD field of the SQLDA to zero.
--
sqlda_in := new sqlda_record;
sqlda_in := sqlda_in_array(stmt_index);
cur_procid := procedure_ids(stmt_index);

EXEC SQL EXECUTE :cur_procid USING DESCRIPTOR :sqlda_in;

end EXECUTE_SQL;
```

11.7.3 Executing Multiple SELECT Statements

To process more than one SELECT statement with a single set of dynamic SQL statements, use a parameter instead of an explicit statement name in the PREPARE statement. The PREPARE statement returns a statement identifier in this parameter, which you pass to the DESCRIBE . . . SELECT LIST, DESCRIBE . . . MARKERS, and extended dynamic DECLARE CURSOR statements.

In addition, use a parameter for the name of the cursor in the extended dynamic DECLARE CURSOR statement. You then pass the parameter for the cursor name to the OPEN, FETCH, and CLOSE statements. The parameter has an integer data type. The statement identifier in the PREPARE statement also has an integer data type, which must be initialized before a PREPARE statement. The cursor name parameter is a character string with a 31-character maximum length.

If you use an extended dynamic DECLARE CURSOR statement, you specify a parameter rather than an explicit statement name for both the cursor name and the statement identifier. You must also use the parameters instead of explicit cursor and statement names in associated dynamic SQL statements such as PREPARE, DESCRIBE, FETCH, OPEN, or CLOSE.

Example 11–11, which is an excerpt from the sql_multi_stmt_dyn.sqlada sample program, shows how you can use one set of dynamic statements to process any number of SELECT statements. The PREPARE_SQL procedure contains PREPARE, DESCRIBE . . . SELECT LIST, DESCRIBE . . . MARKERS, and extended dynamic DECLARE CURSOR statements. The main program uses the PREPARE_SQL statement to prepare both SELECT and non-SELECT statements.

The DISPLAY_DATA procedure contains OPEN, FETCH, and CLOSE statements that use parameters to refer to the cursor name.

Example 11–11 Executing More Than One SELECT Statement

```
-- This procedure prepares a statement for dynamic execution from the
-- string passed to it.  It can prepare any number of statements because
-- the statement is passed to it as the parameter cur_stmt.

procedure PREPARE_SQL is
    CUR_CURSOR : string(1..31) := (others => ' ');
    CUR_PROCID : integer := 0;
    CUR_STMT : string(1..1024) := (others => ' ');

begin

-- Allocate separate SQLDAs for parameter markers (sqlda_in) and select list
-- items (sqlda_out).  Assign the value of the constant MAXPARMS (set in the
-- declarations section) to the SQLN field of both SQLDA structures. SQLN
-- specifies to SQL the maximum size of the SQLDA.

sqlda_in := new sqlda_record;
sqlda_in.sqln := maxparms;
sqlda_out := new sqlda_record;
sqlda_out.sqln := maxparms;

-- Assign the SQL statement that was constructed in the procedure
-- CONSTRUCT_SQL to the variable cur_stmt.

cur_stmt := sql_stmt;

-- Use the PREPARE statement just to prepare the dynamic statement
-- for dynamic execution from the string passed to it and not
-- to write information about select list items to an SQLDA.

EXEC SQL PREPARE :cur_procid  FROM :cur_stmt;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;

-- Use the DESCRIBE ... SELECT LIST statement to write information
-- about the number and data type of any select list items in the
-- statement to an SQLDA (specifically, the sqlda_out SQLDA specified).

EXEC SQL DESCRIBE :cur_procid SELECT LIST INTO :sqlda_out;
```

(continued on next page)

Example 11–11 (Cont.) Executing More Than One SELECT Statement

```
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;

-- Use the DESCRIBE...MARKERS statement to write information about any
-- parameter markers in the dynamic statement to sqlda_in.  The statement
-- writes information to an SQLDA (specifically, the sqlda_in SQLDA
-- specified) about the number and data type of any parameter markers in
-- the prepared dynamic statement.  SELECT statements may also have
-- parameter markers.

EXEC SQL DESCRIBE :cur_procid MARKERS INTO sqlda_in;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;

-- If the operation is 'R' (read), create a unique name for the cursor
-- name so that the program can pass the cursor name to the extended
-- dynamic DECLARE CURSOR statement.

if cur_op(1) = 'R' then
    cur_cursor(1) := 'C';
    cur_cursor(2..name_strlng) := cur_name(1..name_strlng - 1);

-- Declare the extended dynamic cursor.

    EXEC SQL DECLARE :cur_cursor CURSOR FOR :cur_procid;
    case sqlca.sqlcode is
        when sql_success => null;
        when others => raise syntax_error;
    end case;
end if;

-- This section of the program stores the values of the parameters for
-- statement identifiers and cursor names in arrays.

number_of_procs := number_of_procs + 1;

-- Assign information about parameter markers and select list items to
-- array elements.

sqlda_in_array(number_of_procs) := sqlda_in;
sqlda_out_array(number_of_procs) := sqlda_out;

-- Assign the user-specified statement name to an array element. Assign
-- the statement identifier returned by the PREPARE statement to an
-- array element.  In addition, if the statement is a SELECT statement,
-- assign the cursor name to an array element.

procedure_names(number_of_procs) := cur_name;
procedure_ids(number_of_procs) := cur_procid;
```

(continued on next page)

Example 11–11 (Cont.) Executing More Than One SELECT Statement

```
if cur_op(1) = 'R' then
    cursor_names(number_of_procs) := cur_cursor;
end if;
   .
   .
   .
end PREPARE_SQL;
   .
   .
   .
procedure DISPLAY_DATA is
   .
   .
   .
begin  -- procedure body DISPLAY_DATA

-- Before displaying any data, allocate buffers to hold the data
-- returned by SQL.
--
    allocate_buffers;

-- Allocate and assign SQLDAs for the requested SQL procedure.
--
sqlda_in := new sqlda_record;
sqlda_in := sqlda_in_array(stmt_index);
sqlda_out := new sqlda_record;
sqlda_out := sqlda_out_array(stmt_index);
cur_cursor := cursor_names(stmt_index);

-- Open the previously declared cursor.  The statement specifies
-- an SQLDA (specifically, sqlda_in) as the source of addresses for any
-- parameter markers in the cursor's SELECT statement.
--
EXEC SQL OPEN :cur_cursor USING DESCRIPTOR sqlda_in;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise unexpected_error;
end case;

-- Fetch the first row from the result table.  This statement fetches a
-- row from the opened cursor and writes it to the addresses specified
-- in an SQLDA (specifically, sqlda_out).
--

EXEC SQL FETCH :cur_cursor USING DESCRIPTOR sqlda_out;
case sqlca.sqlcode is

--  Check to see if the result table has any rows.

    when sql_success => null;
    when stream_eof =>
        put_line("No records found.");
        new_line;
    when others => raise unexpected_error;
end case;
```

(continued on next page)

Example 11–11 (Cont.) Executing More Than One SELECT Statement

```
-- Set up a loop to display first row, fetch and display second and
-- subsequent rows.

    rowcount := 0;
    while sqlca.sqlcode = 0 loop
        rowcount := rowcount + 1;

--      Execute the DISPLAY_ROW procedure.
        display_row;

--      To display only 5 rows, exit the loop if the loop counter
--      equals MAXROW (hardcoded as 5 in this program).

        if rowcount = maxrows then exit; end if;

--      Fetch another row, exit the loop if no more rows.

        EXEC SQL FETCH :cur_cursor USING DESCRIPTOR sqlda_out;
        case sqlca.sqlcode is
            when sql_success => null;
            when stream_eof => exit;
            when others => raise unexpected_error;
        end case;
    end loop;

-- Close the cursor.

EXEC SQL CLOSE :cur_cursor;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise unexpected_error;
end case;
   .
   .
   .
end DISPLAY_DATA;
```

11.8 Finding the Sample Programs Used in This Chapter

The entire sql_multi_stmt_dyn.sqlada program used in Section 11.7 is available on line in the samples directory.

The complete sql_dynamic program is also available on line. It uses the SQL module language to illustrate the most general type of dynamic SQL application. As does the program in Example 11–1, sql_dynamic accepts a statement from the terminal, but handles SELECT and non-SELECT statements with or without parameter markers. This program also uses the SQLDA2 data structure, which allows for the storage of information necessary for the date-time data types.

The main C program, sql_dynamic, is used to illustrate two different types of user implementation. Separate driver modules are used to illustrate each of these. The entire application consists of the following source modules:

- The sql_dynamic_driver_i.c module accepts an SQL statement from the user and passes it to the sql_dynamic.c module. This driver module is used with sql_dynamic.c and sql_dynamic_c.sqlmod to simulate interactive SQL. It also allows the user to execute SQL queries contained in a script file.
- The sql_dynamic_driver_m.c module repeatedly executes a series of preestablished SQL queries. The sql_dynamic_driver_m.c module includes one INSERT, one SELECT, and one UPDATE statement that execute repeatedly until the user exits. You can modify this program to accept queries from a file.

  This example also requires that the SQL$DATABASE logical name or SQL_DATABASE configuration parameter be defined.
- The sql_dynamic.c module processes the statement passed to it from the driver module, displaying results on the terminal screen.
- The sql_dynamic_c.sqlmod file is an SQL module language file that contains all the subroutines called by the host language modules.

The sql_dynamic_i program can be used both interactively and to execute queries from a script file. To execute queries from a file, create the file, and use the @filename.sql statement at the DynamicSQL> prompt as follows:

```
This program processes any valid SQL statement using dynamic SQL.

Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> <<CR> or <CTRL-Z> to exit>:

DynamicSQL> @dynamic.sql;
```

The dynamic.sql file might look like this:

```
ATTACH 'FILENAME personnel';
SELECT LAST_NAME,FIRST_NAME,EMPLOYEE_ID FROM EMPLOYEES
ORDER BY EMPLOYEE_ID;
SELECT LAST_NAME,BIRTHDAY FROM EMPLOYEES WHERE EMPLOYEE_ID > ?;
   .
   .
   .
```

The following example shows the execution of sql_dynamic_i in an interactive mode:

```
This program processes any valid SQL statement using dynamic SQL.

Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> <<CR> or <CTRL-Z> to exit>:

DynamicSQL> ATTACH 'FILENAME mf_personnel';

The SQL statement to be executed dynamically is:
 ATTACH 'FILENAME mf_personnel';

Enter the SQL statement to process on the following line(s), terminating your
statement with a semicolon <;> <<CR> or <CTRL-Z> to exit>:

DynamicSQL> SELECT * FROM EMPLOYEES WHERE LAST_NAME = ?;

The SQL statement to be executed dynamically is:
 SELECT * FROM EMPLOYEES WHERE LAST_NAME = ?;

Enter value for parameter 'LAST_NAME'
(Maximum length is 14)Nash
---------------------------------------
Field EMPLOYEE_ID:00168
Field LAST_NAME:Nash
Field FIRST_NAME:Norman
Field MIDDLE_INITIAL: NULL
   .
   .
   .
DynamicSQL> Ctrl/Z
$
```

The sql_dynamic program accepts a statement the user types at the terminal and passes it to a processing module. (To see the complete source code for the module, look in the samples directory.)

To process the SQL statement string passed by the driver module, sql_dynamic issues calls to SQL module language procedures that execute PREPARE and DESCRIBE statements for the statement string. The program then uses information written by the PREPARE and DESCRIBE statements to the SQLERRD field of the SQLCA structure to test if the statement string is a SELECT statement and to an SQLDA2 structure to test if the statement contains parameter markers.

Depending on the results of the tests, sql_dynamic executes one or both of the following subroutines:

- GET_IN_PARAMS if the statement string contains parameter markers

  The GET_IN_PARAMS subroutine allocates storage buffers for all the parameter markers in the statement string, writes the addresses of the buffers to an SQLDA2 structure, prompts the user for values corresponding to each parameter marker, and stores those values in the variables.

- ALLOCATE_BUFFERS if the statement string is a SELECT statement

  The ALLOCATE_BUFFERS subroutine allocates storage buffers for all the select list items in the statement string and writes the addresses of the buffers to an SQLDA2 structure.

For SELECT statements, sql_dynamic calls an SQL module language procedure to open a cursor and executes a loop. For each row of the result table created by the OPEN statement, the loop:

- Calls an SQL module language procedure to fetch the row.
- Calls another subroutine, DISPLAY_ROW, which uses the storage allocated by the ALLOCATE_BUFFERS subroutine to hold values for the row and displays them at the terminal.

For non-SELECT statements, sql_dynamic processes the prepared statement with an EXECUTE statement.

If you want to see the sql_dynamic program in its entirety, copy it from the samples directory.

Part IV

Programmatic Structures

This part discusses the components SQL provides to give you more control over your programs:

- Compound statements
- Stored routines
- External routines

12

Using Compound Statements in SQL

You can use standard programming constructs, such as flow-control statements, in SQL. To do so, you use those constructs within compound statements. A **compound statement** groups one or more SQL statements, including flow-control statements, into the context of a single statement.

In the sections that follow, you will become familiar with:

- The concept of compound statements
- Using compound statements to increase the performance of your database applications
- Writing compound statements
- Controlling the atomicity of compound statements
- Controlling transactions in compound statements
- Processing compound statements dynamically
- Debugging compound statements
- Retrieving information, such as row count, connection information and transactions in a compound statement
- Handling exception and completion conditions in compound statements

12.1 Introducing Compound Statements

Compound statements allow you to use a set of standard programming constructs, such as flow-control statements, in SQL and allow you to group SQL statements into the context of a single statement. Procedures that contain a compound statement are called **multistatement procedures**.

Compound statements allow you to control the sequence of statement execution and to perform both simple and complex decision-making tasks within the compound statement. Compound statements can include the following:

- SQL data manipulation statements

- Local variables
- Flow-control statements, such as IF statements and FOR loops
- Clauses to control the atomicity of the statement
- The CALL statement, to invoke external or stored procedures
- Statements to begin or end transactions

You can use compound statements in SQL module language, embedded SQL, and interactive SQL, and you can process compound statements with dynamic SQL. In addition, you must use compound statements when you define stored procedures or functions.

12.2 Using Compound Statements to Increase Performance

When you use compound statements, you can perform comprehensive business transactions and complex program control within the database environment.

Packaging multiple SQL statements in a compound statement improves client and server performance, particularly remote access, because it reduces the amount of interaction between the Oracle Rdb system and the application program accessing the database system. In addition, it improves performance even if all interaction occurs on one node.

Compound statements, and thus multistatement procedures, provide the following benefits. They:

- Reduce procedure calls and simplify application program logic by letting a single request perform many operations on the database system without requiring a return to the application program for each statement, thus increasing the performance of your application.
- Let you achieve procedural abstraction in many cases. **Procedural abstraction** means that one physical module performs one logical database function.
- Provide a cleaner separation between database requests and access.
- Reduce network traffic by reducing the number of procedure calls in a client/server configuration, resulting in better performance.
- Provide faster execution, especially in high-performance database applications.
- Allow multiple operations to be processed as one (using the ATOMIC keyword).

- Reduce mismatching of data types between the application and the database.

12.3 Writing a Compound Statement

When you write a compound statement, you use the BEGIN keyword to specify the beginning of the compound statement and the END keyword to specify the end. You can include one or more SQL statements within a compound statement, as shown in the following interactive SQL example:

```
SQL>  BEGIN
cont>     UPDATE DEPARTMENTS
cont>            SET MANAGER_ID = '00167'
cont>            WHERE DEPARTMENT_CODE = 'SALE';
cont>     UPDATE JOB_HISTORY
cont>            SET JOB_END = CURRENT_TIMESTAMP
cont>            WHERE EMPLOYEE_ID = '00167' AND JOB_END IS NULL;
cont>     INSERT INTO JOB_HISTORY
cont>            (EMPLOYEE_ID, DEPARTMENT_CODE, JOB_CODE, JOB_START)
cont>            VALUES
cont>            ('00167', 'SALE', 'DMGR', CURRENT_TIMESTAMP);
cont> END;
SQL>
```

SQL sequentially executes the SQL statements within a compound statement.

You can include the following in a compound statement:

- Variable declarations and assignments
- SQL data manipulation statements

  You can use a singleton SELECT, UPDATE, INSERT, or DELETE statement in a compound statement. (Note that you cannot use a SELECT statement other than a singleton SELECT.)
- CASE statement

  The CASE statement executes one of a sequence of alternate statement blocks in a compound statement.
- FOR statement

  The FOR statement executes SQL statements for each row of a query expression. The FOR statement provides the functionality of cursors in compound statements. (You cannot use the DECLARE CURSOR, OPEN, FETCH, and CLOSE statements in compound statements.)
- IF statement

  The IF statement conditionally executes one or more SQL statements.

- LOOP statement

  The LOOP statement allows the repetitive execution of one or more SQL statements in a compound statement until an error occurs or a LEAVE statement is executed, or as long as a WHILE predicate clause evaluates to TRUE.

- LEAVE statement

  The LEAVE statement lets the program exit from the compound or flow-control statement that contains it.

- TRACE statement

  The TRACE statement writes values to a log file for each value expression that the statement evaluates. Use the TRACE statement to debug compound statements.

- CALL statement

  The CALL statement lets the program call stored or external procedures. To call external procedures, the CALL statement *must* be in a compound statement.

- GET DIAGNOSTICS statement

  The GET DIAGNOSTICS statement provides information about the current environment and completion conditions for compound statements.

- SIGNAL statement

  The SIGNAL statement provides information about exception conditions for compound statements.

- Clauses to control the atomicity of the statement
- Statements to begin or end transactions

You can nest compound statements to any desired depth. Many examples in the following sections show nested compound statements.

12.3.1 Declaring and Assigning Variables

In compound statements, you can declare variables, assign values to those variables, and use the variables in SQL statements. In compound statements, variable declarations *must* appear before any executable SQL statement.

To declare a variable, use the DECLARE variable clause of the compound statement. To assign a value to a variable, you can use the SET statement or the DEFAULT clause of the DECLARE variable clause.

The following example shows how to declare the variable :mgrid, assign a value to it, and use the variable in an UPDATE statement:

```
SQL> BEGIN
cont>    DECLARE :mgrid CHAR(5);
cont>    SET :mgrid = '00167';
cont>    UPDATE DEPARTMENTS
cont>           SET MANAGER_ID = :mgrid
cont>           WHERE DEPARTMENT_CODE = 'SALE';
cont> END;
SQL>
```

You can specify the data type explicitly or you can specify it implicitly by using a domain name. However, you cannot specify or use the LIST data type in a compound statement.

When you declare a variable, you can assign a default value to the variable by using the DEFAULT clause, as the following example shows:

```
DECLARE :mgrid CHAR(5) DEFAULT '00167';
```

In place of the DEFAULT keyword, you can use an equals sign (=).

The default value can be almost any value expression, including subqueries, conditional, character, date/time and numeric expressions. However, the default value cannot be an aggregate function. If you do not specify a default value, but you specify a domain name as the data type and a default value is defined for that domain, SQL uses the domain's default value as the variable's default value.

You can also specify whether or not a variable can be updated by specifying the UPDATABLE or CONSTANT keywords. In the following example, because :mgrid is declared as a constant, you cannot change the value of the variable:

```
DECLARE :mgrid CONSTANT CHAR(5) DEFAULT '00167' ;
```

If you use the CONSTANT keyword, you must specify a default value.

The UPDATABLE keyword, which is the default, lets you change the value of a variable. If you use the variable in a SET statement, an INTO clause, or as an OUT or INOUT procedure parameter, you must declare the variable as updatable.

Variables in compound statements can hold null values, unlike formal parameters in SQL module procedures or in embedded SQL statements, which require the use of indicator parameters to handle null values. You do not use indicator parameters with variables in compound statements.

You can assign a null value to a variable directly by using the SET statement or the DEFAULT clause of the DECLARE variable statement or indirectly by using the variable in an SQL data manipulation statement. For example, the following compound statement returns a null value into the :job_end_date variable:

```
BEGIN
   DECLARE :job_end_date DATE;
   SELECT JOB_END INTO :job_end_date FROM JOB_HISTORY
          WHERE EMPLOYEE_ID = '00180';
END;
```

When you declare a variable in a compound statement, the scope of the variable is limited to that compound statement. That is, when the compound statement completes executing, you can no longer use the variable without declaring it again.

When you nest compound statements, you can declare and assign values to the same variable name in an inner compound statement, even if you used the variable in the outer compound statement. When you do, the variable declaration and assignment in the inner compound statement occludes (or hides) the variable declaration and assignment in the outer compound statement.

The following excerpt from an SQL module shows a nested compound statement that declares and assigns a value to the variable :inc in an outer compound statement. Then, in the inner compound statement, it declares the variable :inc again, but with different precision, and assigns a different value to it.

```
BEGIN
-- Because :inc is declared as SMALLINT(3), SQL multiplies the salaries
-- of employees in NH by 1.055.
    DECLARE :inc SMALLINT(3);
    SET :inc = 1.055;

    UPDATE SALARY_HISTORY
           SET SALARY_AMOUNT = (SALARY_AMOUNT * :inc)
           WHERE SALARY_END IS NULL
              AND EMPLOYEE_ID IN (SELECT EMPLOYEE_ID FROM EMPLOYEES
                                  WHERE STATE = 'NH');

    BEGIN
    -- Because :inc is declared as INTEGER(2), SQL rounds the number to
    -- two decimal places (1.07) before it multiplies the salaries
    -- of employees in MA.
        DECLARE :inc INTEGER(2);
```

```
    -- The SET statement assigns a new value to the variable :inc.
        SET :inc = 1.066;
        UPDATE SALARY_HISTORY
               SET SALARY_AMOUNT = (SALARY_AMOUNT * :inc)
               WHERE SALARY_END IS NULL
                   AND EMPLOYEE_ID IN (SELECT EMPLOYEE_ID FROM EMPLOYEES
                               WHERE STATE = 'MA');
    END;
-- Because the following statement is in the outer compound statement,
-- :inc is declared as SMALLINT(3) with the value 1.055.  As a result,
-- SQL multiplies the salaries of employees in CT by 1.055.
    UPDATE SALARY_HISTORY
           SET SALARY_AMOUNT = (salary_amount * :inc)
           WHERE SALARY_END IS NULL
              AND EMPLOYEE_ID IN (SELECT EMPLOYEE_ID FROM EMPLOYEES
                               WHERE STATE = 'CT');
END;
```

For more information about the SET statement and the DECLARE variable clause, see the *Oracle Rdb7 SQL Reference Manual*.

12.3.2 Using the IF Statement

Use the IF statement to conditionally execute one or more SQL statements.

The following excerpt from an embedded SQL program shows how you use an IF statement, along with host language variables and variables in the compound statement, to control the conditional execution of an UPDATE statement.

In the example, the host language program declares the variable :mgrid and passes the value in the variable to the compound statement. The compound statement retrieves the MANAGER_ID of the current manager of a department by using a singleton SELECT statement and assigning the value to the variable in the INTO clause. Then, it uses the IF statement to check whether the current MANAGER_ID is different from the ID of the manager to be assigned. If it is different, SQL executes the UPDATE statement in the THEN clause. If the MANAGER_ID is not different, SQL does not execute the UPDATE statement in the THEN clause.

```
-- Use the EXEC SQL keywords with compound statements, just as with simple
-- SQL statements.
EXEC SQL BEGIN
            DECLARE  :cur_mgrid CHAR(5);

            SELECT MANAGER_ID INTO :cur_mgrid FROM DEPARTMENTS
                  WHERE DEPARTMENT_CODE = 'SALE';
```

```
-- You must precede host language variables with a colon when you use them
-- in compound statements.
            IF :cur_mgrid <> :mgrid
            THEN
               UPDATE DEPARTMENTS
                      SET MANAGER_ID = :mgrid
                      WHERE DEPARTMENT_CODE = 'SALE';
            END IF;
        END;
```

As with all flow-control statements, you can nest IF statements to any level. In many cases, however, using the ELSEIF clause of the IF statement can make your code easier to read. The following excerpt from an SQL module shows examples of both nesting IF statements and using the ELSEIF clause:

```
BEGIN
    DECLARE :state_code CHAR(2);
    DECLARE :mgrid CHAR(5);
    DECLARE :cur_mgrid CHAR(5);
    SET :mgrid = '00167';

    SELECT D.MANAGER_ID, E.STATE INTO :cur_mgrid, :state_code
           FROM DEPARTMENTS D, EMPLOYEES E
           WHERE DEPARTMENT_CODE = 'SALE'
                 AND D.MANAGER_ID = E.EMPLOYEE_ID;

-- Outer IF statement.
-- If the manager is a new manager, update the table.
    IF :cur_mgrid <> :mgrid
    THEN
       UPDATE DEPARTMENTS
              SET MANAGER_ID = :mgrid
              WHERE DEPARTMENT_CODE = 'SALE';

    -- Nested IF statement.
    -- Give the new manager a raise. The amount depends on the manager's
    -- state.
       IF :state_code = 'NH'
       THEN
          UPDATE SALARY_HISTORY S
                 SET SALARY_AMOUNT = (salary_amount * 1.05)
                 WHERE SALARY_END IS NULL
                       AND EMPLOYEE_ID = :mgrid;

    -- ELSEIF clause of nested IF statement.
       ELSEIF :state_code = 'MA'
       THEN
          UPDATE SALARY_HISTORY S
                 SET SALARY_AMOUNT = (salary_amount * 1.07)
                 WHERE SALARY_END IS NULL
                       AND EMPLOYEE_ID = :mgrid;
```

```
    -- ELSE clause executes if the :state_code is neither NH nor MA.
       ELSE
          UPDATE SALARY_HISTORY S
                 SET SALARY_AMOUNT = (salary_amount * 1.06)
                 WHERE SALARY_END IS NULL
                       AND EMPLOYEE_ID = :mgrid;
       END IF;
    END IF;
END;
```

The preceding example increases employee salaries based on the state in which they reside, to take tax differences into account.

12.3.3 Using the CASE Statement

The CASE statement lets you list several courses of action and choose one to be executed at run time, depending on the value of an expression.

The following example performs the same tasks as the IF . . . ELSEIF example in Section 12.3.2, but it uses the CASE statement instead of the nested IF statement:

```
BEGIN
    DECLARE :state_code CHAR(2);
    DECLARE :mgrid CHAR(5);
    DECLARE :cur_mgrid CHAR(5);
    SET :mgrid = '00167';

    SELECT D.MANAGER_ID, E.STATE INTO :cur_mgrid, :state_code
           FROM DEPARTMENTS D, EMPLOYEES E
           WHERE DEPARTMENT_CODE = 'SALE'
                 AND D.MANAGER_ID = E.EMPLOYEE_ID;

-- Outer IF statement.
    IF :cur_mgrid <> :mgrid
    THEN
       UPDATE DEPARTMENTS
              SET MANAGER_ID = :mgrid
              WHERE DEPARTMENT_CODE = 'SALE';

-- CASE statement.  Give the new manager a raise. Because of differences in
-- state taxes, the amount varies with the state in which the manager resides.

       CASE :state_code
       WHEN 'NH'
       THEN UPDATE SALARY_HISTORY S
                   SET SALARY_AMOUNT = (salary_amount * 1.05)
                   WHERE SALARY_END IS NULL
                         AND EMPLOYEE_ID = :mgrid;
```

```
      WHEN 'MA'
      THEN UPDATE SALARY_HISTORY S
                  SET SALARY_AMOUNT = (salary_amount * 1.07)
                  WHERE SALARY_END IS NULL
                        AND EMPLOYEE_ID = :mgrid;

   -- The ELSE clause executes if the :state_code is neither NH or MA.
      ELSE UPDATE SALARY_HISTORY S
                 SET SALARY_AMOUNT = (salary_amount * 1.06)
                 WHERE SALARY_END IS NULL
                       AND EMPLOYEE_ID = :mgrid;
      END CASE;
   END IF;
END;
```

If a CASE statement does not contain an ELSE clause and the value of the expression does not match the values in any WHEN clause, SQL generates an exception.

12.3.4 Using the LOOP Statement

The LOOP statement allows the repetitive execution of one or more SQL statements in a compound statement as long as a WHILE predicate clause evaluates to TRUE, or until an error occurs or a LEAVE statement is executed.

The following example shows how to use the LOOP statement to trace the management hierarchy in a company from the bottom up. The example begins with one employee and finds that employee's supervisor. Then it finds the supervisor's supervisor and continues looping until it finds a row where the EMPLOYEE_ID is the same as the SUPERVISOR_ID.

```
-- This procedure begins with the employee with EMPLOYEE_ID 00180 and
-- finds his supervisor, tracing the hierarchy of the company.
BEGIN
   DECLARE :emp_id  CHAR(5);
   DECLARE :sup_id CHAR(5);
   DECLARE :id CHAR(5);
   SET :emp_id = '00180';

-- Initialize :id and :sup_id to be different values, so that the LOOP
-- will execute at least once.
   SET :id = '00000';
   SET :sup_id  = '00001';
```

```
   WHILE :id <> :sup_id
   LOOP
      SELECT EMPLOYEE_ID, SUPERVISOR_ID INTO :id, :sup_id
             FROM JOB_HISTORY
             WHERE EMPLOYEE_ID = :emp_id
                   AND JOB_END is NULL;
      SET :emp_id = :sup_id;
  END LOOP;
END;
```

12.3.5 Using the FOR Statement

The FOR statement lets you execute an SQL statement, including a compound statement, for each row returned from a select expression. A FOR statement provides features for compound statements that are similar to those provided by cursors. The FOR statement implicitly declares and opens a cursor, repeatedly fetches rows of values, and closes the cursor when the cursor reaches end-of-stream.

The FOR statement evaluates a select expression that you specify in the AS EACH ROW OF clause. For each iteration of the FOR loop, SQL stores the value of each column that you specify in the AS EACH ROW OF clause in a field in a record. You specify the name of the record immediately following the FOR keyword. For example, in the following excerpt, SQL stores the columns MINIMUM_SALARY and MAXIMUM_SALARY in the record :jobrec:

```
-- The :jobrec variable represents a record that holds columns from the
-- selected row.
   FOR :jobrec
       AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR

-- The select expression specifies that only two columns be stored in the
-- record :jobrec.
          SELECT MINIMUM_SALARY, MAXIMUM_SALARY FROM JOBS
                 WHERE MINIMUM_SALARY < 20000
```

When you refer to columns stored in the record, you must qualify the column name with the record name, as shown in the following excerpt:

```
IF :jobrec.MINIMUM_SALARY  >  :jobrec.MAXIMUM_SALARY
```

You can nest FOR statements to any depth.

The following example shows a simple FOR statement that selects all rows that have a minimum salary of less than $20000.00 and, for each selected row, increases the minimum salary. In addition, the example uses a nested IF statement to increase the maximum salary if it is less than the minimum salary.

```
BEGIN
-- The :jobrec variable represents a record that holds columns from the
-- selected row.
   FOR :jobrec
       AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR

-- The select expression.
       SELECT MINIMUM_SALARY, MAXIMUM_SALARY FROM JOBS
                 WHERE MINIMUM_SALARY < 20000
       DO
-- Update the current row in the JOB_CURSOR.
          UPDATE JOBS
-- No need to qualify the column names.
                 SET MINIMUM_SALARY = MINIMUM_SALARY * 1.10
                 WHERE CURRENT OF JOB_CURSOR;

-- If the minimum salary is now greater than the maximum salary, increase the
-- maximum salary.
          IF :jobrec.MINIMUM_SALARY  >  :jobrec.MAXIMUM_SALARY
          THEN
             UPDATE JOBS
                    SET MAXIMUM_SALARY = MAXIMUM_SALARY * 1.10
                    WHERE CURRENT OF JOB_CURSOR;
          END IF;

   END FOR;
END;
```

As with UPDATE statements that you use with cursors, you use the WHERE CURRENT OF clause to specify that SQL update only the row on which the named cursor is positioned.

The following example, which nests IF statements within the FOR statement, shows an SQL module procedure that contains a compound statement. The procedure calculates the highest raise an employee has received. The host language program passes the employee ID to the procedure through the parameter :ID. The procedure returns the maximum percent raise to the host language program in the parameter :RESULT.

```
PROCEDURE max_raise (SQLCODE,
                     :ID CHAR(5),
                     :RESULT REAL);

BEGIN
   DECLARE :last_salary INTEGER(2) DEFAULT 0;
   DECLARE :pct_raise INTEGER(2) DEFAULT 0;
   SET :RESULT =0;

   FOR :salhist_rec
       AS EACH ROW OF
       SELECT SALARY_AMOUNT FROM SALARY_HISTORY
              WHERE EMPLOYEE_ID = :ID
              ORDER BY SALARY_START ASC
```

```
   DO
       IF :last_salary <> 0
       THEN
          SET :pct_raise = (:salhist_rec.salary_amount - :last_salary)
                           / :last_salary;
       END IF;

       IF :pct_raise > :RESULT
       THEN
          SET :RESULT = :pct_raise;
       END IF;

       SET :last_salary = :salhist_rec.salary_amount;

   END FOR;
END;
```

See Section 4.4 for more information about using compound statements in SQL modules.

12.3.6 Using Labels in Compound Statements

You can name compound statements and FOR and LOOP statements using a label. Labeling compound statements is particularly useful when you use nested compound statements and FOR and LOOP statements, and you want to specify that the program exit from a specific statement. (See Section 12.3.7 for information on using the LEAVE statement to exit from a compound statement.)

To improve the readability of a program, specify the same label at the end of the compound statement. The following excerpt from an SQL module specifies the label UPD_MGRID for the compound statement:

```
UPD_MGRID:     -- Beginning label for compound statement
BEGIN
    DECLARE :mgrid CHAR(5);
    SET :MGRID = '00167';
    UPDATE DEPARTMENTS
           SET MANAGER_ID = :mgrid
           WHERE DEPARTMENT_CODE = 'SALE';
END
UPD_MGRID      -- Ending label for compound statement
;              -- Ending semicolon for compound statement
```

You must append a colon (:) to the label name when you specify the beginning label. You can use labels in the following cases:

- At the beginning and end of compound statements

  Specify the beginning label before the BEGIN keyword and the ending label after the END keyword.

However, note the following:

- In interactive SQL and precompiled SQL, you *cannot* use a label on the outermost compound statement, although you can use labels on compound statements nested in another compound statement.
- In SQL module language, you *can* use labels on the outermost, as well as the inner, compound statements. If you do not provide a label for the outermost compound statement, SQL assigns the name of the SQL module procedure to the compound statement.

- At the beginning and end of a LOOP statement

  Specify the beginning label before the WHILE keyword and the ending label after the END LOOP keywords.

- At the beginning and end of a FOR statement

  Specify the beginning label before the FOR keyword and the ending label after the END FOR keywords.

12.3.7 Using the LEAVE Statement

The LEAVE statement lets your program exit from the compound or flow-control statement that contains it. You can use the LEAVE statement to exit from the following program structures:

- Compound statement
- FOR statement
- LOOP statement
- SQL module procedure

To use the LEAVE statement, specify the label of the statement from which you want your program to exit. For example, to exit from a LOOP statement that has the label loop1, use the following statement:

```
LEAVE loop1;
```

To exit from an SQL module procedure, specify the name of the procedure as an argument to the LEAVE statement.

The following example counts the number of employees who have been supervisors and the number of employees who have been both supervisors and managers. It uses the LEAVE statement to exit from the inner FOR statement when the JOB_CODE for a given employee is 'DMGR', ensuring that an employee is not counted more than once.

```
BEGIN
    DECLARE :supnum INTEGER;
    DECLARE :bothnum INTEGER;

    SET :supnum = 0;
    SET :bothnum = 0;

-- Count the number of employees who have been supervisors.
    jobloop:
    FOR :job_rec
        AS EACH ROW OF
        SELECT DISTINCT(EMPLOYEE_ID) FROM JOB_HISTORY
               WHERE JOB_CODE = 'DSUP'
    DO
        SET :supnum = :supnum + 1;

-- Count the number of employees who have been managers, as well as
-- supervisors.
        innerjobloop:
        FOR :innerjob_rec
            AS EACH ROW OF
            SELECT EMPLOYEE_ID, JOB_CODE FROM JOB_HISTORY
                   WHERE EMPLOYEE_ID = :job_rec.employee_id
        DO
            IF :innerjob_rec.JOB_CODE = 'DMGR'
            THEN
               SET :bothnum = :bothnum + 1;

-- Exit from the inner FOR loop.
               LEAVE innerjobloop;
            END IF;
        END FOR innerjobloop;

    END FOR jobloop;
END;
```

You can exit from a compound, FOR, or LOOP statement no matter how deeply nested it is. In the preceding example, if you specified LEAVE jobloop instead of LEAVE innerjobloop, SQL would exit not only from the inner FOR loop, but also from the outer FOR loop.

12.3.8 Invoking Stored or External Procedures

You can invoke stored or external procedures by using the CALL statement. When you use a CALL statement in a compound statement, you can use almost any value expression as an IN parameter, unlike a CALL statement in a single statement, which is limited to host variables and numeric and string literals. (You cannot use dbkeys or aggregate functions as parameters.)

To invoke an external procedure, the CALL statement must be within a compound statement. To invoke a stored procedure, the CALL statement can be a simple statement or within a compound statement.

The following example shows a stored procedure and the compound statement that calls it:

```
-- Create the module that contains the stored procedure.
CREATE MODULE DEPT_BUDG_MOD
    LANGUAGE SQL
    PROCEDURE DEPT_BUDG_PROC ( IN :incr INTEGER (2), INOUT :new_budg INTEGER,
                                  INOUT :cur_budg INTEGER );
    BEGIN
        SELECT BUDGET_PROJECTED INTO :cur_budg
            FROM DEPARTMENTS
            WHERE DEPARTMENT_NAME = 'Engineering';
        SET :new_budg = :cur_budg + (:cur_budg * :incr);
        UPDATE DEPARTMENTS
            SET BUDGET_PROJECTED = :new_budg
            WHERE DEPARTMENT_NAME = 'Engineering';
    END;
END MODULE;

-- Call the DEPT_BUDG_PROC stored procedure.
BEGIN
    DECLARE :cur_budg_var INTEGER;
    DECLARE :new_budg_var INTEGER;
    DECLARE :incr_var INTEGER (2) DEFAULT .08;

    CALL DEPT_BUDG_PROC ( :incr_var, :new_budg_var, :cur_budg_var);
END;
```

You can use the CALL statement within a stored procedure or function to call another stored procedure. In this way, you can nest stored procedures to any depth, limited only by your system's resources. However, recursion is not allowed. That is, you cannot call a stored procedure that is in use by the current CALL statement.

12.4 Controlling the Atomicity of Compound Statements

Like an individual SQL statement, a compound statement possesses a transaction characteristic called atomicity. **Atomicity** defines what happens to a single SQL statement or a compound statement when an exception occurs. The execution of all single SQL statements is always treated as an atomic event. This means that the statement completes successfully, or if SQL returns an exception, the statement is undone. However, you can control the atomicity of compound statements; they are not atomic by default.

You specify the atomicity of a compound statement by using the ATOMIC and NOT ATOMIC keywords following the BEGIN keyword, as the following example shows:

```
SQL>  BEGIN  ATOMIC
cont>     UPDATE DEPARTMENTS
cont>            SET MANAGER_ID = '00167'
cont>            WHERE DEPARTMENT_CODE = 'SALE';
cont>     UPDATE JOB_HISTORY
cont>            SET JOB_END = CURRENT_TIMESTAMP
cont>            WHERE EMPLOYEE_ID = '00167' AND JOB_END IS NULL;
cont>     INSERT INTO JOB_HISTORY
cont>            (EMPLOYEE_ID, DEPARTMENT_CODE, JOB_CODE, JOB_START)
cont>            VALUES
cont>            ('00167', 'SALE', 'DMGR', CURRENT_TIMESTAMP);
cont> END;
SQL>
```

If you specify that the outer compound statement is ATOMIC, you must specify the ATOMIC keyword for any nested compound statements.

You can nest an ATOMIC compound statement within a NOT ATOMIC statement.

What happens when SQL encounters an exception in a compound statement depends on whether the compound statement is defined as ATOMIC or NOT ATOMIC. The actions are as follows:

- ATOMIC

  No SQL statements within a compound statement succeed or fail as a unit. If all SQL statements within a compound statement succeed, the compound statement block succeeds as a whole.

  When an SQL statement raises an exception, however, SQL rolls back any changes, does not execute any statements located after the failed statement, and terminates the compound statement at the point of failure.

  SQL does not undo variable assignments as a result of a statement failure.

- NOT ATOMIC (default)

  No SQL statements that complete successfully, up to the point that a statement fails, are rolled back. Success of some statements in a NOT ATOMIC compound statement can occur. However, SQL immediately terminates the processing in NOT ATOMIC compound statements when an SQL statement returns an exception. The partial work of the statement causing a compound statement to terminate is always rolled back.

In the preceding example, because the statement is ATOMIC, all the statements complete or none complete.

The following example further illustrates the principles of atomicity. It presumes that the INSERT INTO JOB_HISTORY statement generates an exception and shows which SQL statements within a nested compound statement complete or are undone in this circumstance.

```
BEGIN NOT ATOMIC

    BEGIN ATOMIC
          UPDATE DEPARTMENTS  ...                    ❹
          UPDATE JOB_HISTORY  ...                    ❹
    END;

    BEGIN ATOMIC
        INSERT INTO DEPARTMENTS   ...                ❷
        INSERT INTO JOB_HISTORY   ...                ❶ ❷
        INSERT INTO SALARY_HISTORY  ...              ❸
    END;
END;
```

In executing the compound statement shown in the preceding example, SQL executes the SQL statements, until an exception occurs. The following list is keyed to the numbered callouts in this example:

❶ The INSERT INTO JOB_HISTORY statement generates an exception.

❷ Because the inner compound statement is ATOMIC, SQL rolls back the following statements:

- INSERT INTO DEPARTMENTS
- INSERT INTO JOB_HISTORY

❸ SQL does not execute the INSERT INTO SALARY_HISTORY statement because the exception generated by the INSERT INTO JOB_HISTORY statement effectively terminates the inner compound statement.

❹ Because the outer compound statement is NOT ATOMIC, SQL does not undo the UPDATE statements.

In both ATOMIC and NOT ATOMIC compound statements, SQL continues execution of the statements when a completion condition, such as no data, is returned.

You cannot include a SET TRANSACTION, COMMIT, or ROLLBACK statement in an ATOMIC compound statement.

12.5 Controlling Transactions in Compound Statements

You can include SET TRANSACTION, COMMIT, and ROLLBACK statements in compound statements. However, you can include them only in NOT ATOMIC compound statements, not in ATOMIC compound statements.

Beginning and ending transactions within a compound statement improves performance because it reduces the amount of interaction between Oracle Rdb and the application program.

Some of the ways that you can use the SET TRANSACTION, COMMIT, and ROLLBACK statements in a compound statement include the following:

- You can begin and end a transaction within one compound statement.
- You can begin a transaction within one compound statement and end it in another SQL procedure.
- You can begin a transaction in one SQL procedure and end it in a compound statement, or you can begin a transaction in a compound statement and end it outside a compound statement.
- You can use flow-control statements in compound statements to begin or end transactions, based on certain conditions.

  The following example begins a read/write transaction and updates data if the value of the parameter is 0; otherwise, it begins a read-only transaction:

```
PROCEDURE update_or_select
          (SQLSTATE,
           :upd_sel,
           :mgrid);
BEGIN
    IF :upd_sel = 0
    THEN
       SET TRANSACTION READ WRITE;
       UPDATE DEPARTMENTS
              SET MANAGER_ID = :mgrid
              WHERE DEPARTMENT_CODE = 'SALE';
    ELSE
       SET TRANSACTION READ ONLY;
       SELECT MANAGER_ID INTO :mgrid
              FROM DEPARTMENTS
              WHERE  DEPARTMENT_CODE = 'SALE';
    END IF;
END;
```

Because you cannot refer to more than one database in a compound statement, you cannot start a transaction, or commit or roll back a transaction that includes more than one database. For example, if, outside a compound statement, you start a transaction that includes more than one database, you cannot commit or roll back that transaction within a compound statement.

The DECLARE TRANSACTION statement has the same effect on transactions in compound statements as in single SQL statements.

You can retrieve information about transactions, such as the access mode or status, in compound statements by using the GET DIAGNOSTICS statement. For more information about the GET DIAGNOSTICS statement, see Section 12.8.

For more information about transactions, see Chapter 16.

12.6 Processing Compound Statements Dynamically

You cannot include dynamic SQL statements in a compound statement. Nonetheless, SQL lets you process and execute compound statements by using the following dynamic SQL statements:

- PREPARE
- DESCRIBE
- EXECUTE
- EXECUTE IMMEDIATE

For example, if a host language program assigns a compound statement to the parameter :STMT and the statement identifier to the parameter :DYN_STMT_ID, use the following procedure to prepare the statement for dynamic execution:

```
PROCEDURE prepare_stmt
    (SQLSTATE,
     :DYN_STMT_ID INTEGER,
     :STMT CHAR(16000));

    PREPARE :DYN_STMT_ID FROM :STMT;
```

12.7 Debugging Compound Statements

Use extra care when writing and testing compound statements. Because you cannot use a programming language symbolic debugger to examine compound statement execution within Oracle Rdb, it is difficult to debug compound statements or trace their execution flow.

However, you can use the TRACE statement to monitor the contents of variables used in a compound statement. To enable trace logging, you must use the SET FLAGS 'TRACE' statement or define the RDMS$DEBUG_FLAGS logical name or RDB_DEBUG_FLAGS configuration parameter to be "Xt". The letter X must be an uppercase letter and the letter t must be a lowercase letter, and both must be enclosed in double quotes (") as shown in the following example:

```
$ DEFINE RDMS$DEBUG_FLAGS "Xt"
```

You can redirect the output from the TRACE statement by using the logical name RDMS$DEBUG_FLAGS_OUTPUT or the configuration parameter RDB_DEBUG_FLAGS_OUTPUT.

The SET FLAGS 'TRACE' statement overrides the RDMS$DEBUG_FLAGS logical name or RDB_DEBUG_FLAGS configuration parameter setting. You must be connected to the database before you use the SET FLAGS 'TRACE' statement. Note that you cannot use this statement within a compound statement; it must appear before the compound statement.

Using the TRACE statement, you can trace the value of a value expression, such as a literal, subquery, or a variable. The following example shows how to use the SET FLAGS statement to enable trace logging, how to use the TRACE statement, and shows the output from the TRACE statement:

```
-- Enable trace logging.
SET FLAGS 'TRACE';

BEGIN
    DECLARE :state_code CHAR(2);
    DECLARE :mgrid CHAR(5);
    DECLARE :cur_mgrid CHAR(5);
    SET :mgrid = '00167';

-- Trace a built-in function.
    TRACE 'Trace the current time ', CURRENT_TIMESTAMP;

    SELECT D.MANAGER_ID, E.STATE INTO :cur_mgrid, :state_code
         FROM DEPARTMENTS D, EMPLOYEES E
         WHERE DEPARTMENT_CODE = 'SALE'
               AND D.MANAGER_ID = E.EMPLOYEE_ID;
```

```
-- Trace variables.
    TRACE 'After SELECT mgrid is ', :mgrid;
    TRACE 'After SELECT cur_mgrid is ', :cur_mgrid;

    IF :cur_mgrid <> :mgrid
    THEN
       UPDATE DEPARTMENTS
              SET MANAGER_ID = :mgrid
              WHERE DEPARTMENT_CODE = 'SALE';

-- Trace variables.
       TRACE 'After UPDATE mgrid is ', :mgrid;

       IF :state_code = 'NH'
       THEN
-- Trace a literal.
          TRACE 'Entering IF loop.';

          UPDATE SALARY_HISTORY S
                 SET SALARY_AMOUNT = (salary_amount * 1.05)
                 WHERE SALARY_END IS NULL
                       AND EMPLOYEE_ID = :mgrid;
       END IF;
    END IF;
END;
~Xt: Trace the current time 1994-02-16:12:32:20.46
~Xt: After SELECT mgrid is 00167
~Xt: After SELECT cur_mgrid is 00205
~Xt: After UPDATE mgrid is 00167
~Xt: Entering IF loop.
```

To turn off the display of the prefix "~Xt:", use the SET FLAGS 'NOPREFIX' statement.

To trace the value of a column name, SQL must be able to recognize the context for the column. For example, you can trace a column name in a FOR statement by qualifying the column name with the record name, as the following excerpt shows:

```
-- Enable trace logging and turn off display of the prefix.  You can combine
-- the TRACE and NOPREFIX keywords in the same statement.
SET FLAGS 'TRACE, NOPREFIX';

BEGIN
    DECLARE :supnum INTEGER;
    SET :supnum = 0;

      FOR :job_rec
           AS EACH ROW OF
           SELECT EMPLOYEE_ID, JOB_CODE FROM JOB_HISTORY
```

```
       DO
-- Trace the values of the columns JOB_CODE and EMPLOYEE_ID.
    TRACE 'Job code: ', :job_rec.job_code;
    TRACE 'Employee_id: ', :job_rec.employee_id;
           IF :job_rec.JOB_CODE = 'DSUP'
           THEN
              SET :supnum = :supnum + 1;
           END IF;
       END FOR;
END;
Job code: SPGM
Employee_id: 00164
Job code: DMGR
Employee_id: 00164
   .
   .
   .
```

When you do not enable trace logging, the TRACE statements are inactive and add no overhead to the execution of the procedure.

12.8 Retrieving Information About Compound Statements

You can use the GET DIAGNOSTICS statement to retrieve information about the execution of the previous SQL statement in a compound statement. The GET DIAGNOSTICS statement extracts diagnostic information about the execution of the previous SQL statement. It captures the following diagnostic information from an Oracle Rdb data structure called the **diagnostics area**:

- Status information about rows

  The GET DIAGNOSTICS statement returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or the number of rows fetched by a FOR statement.

- Information about transactions

  The GET DIAGNOSTICS statement returns information about transactions, such as access mode, isolation level, and the number of transactions committed or rolled back.

- The name of the current connection

- The name of the calling routine

  The GET DIAGNOSTICS statement allows a routine to see the name of the routine that called it.

The GET DIAGNOSTIC statement also returns the value of the SQLSTATE or SQLCODE status parameters, as described in Section 12.9.

You can use the GET DIAGNOSTICS statement only within a compound statement.

The following example uses the GET DIAGNOSTICS statement to return the number of rows that have been fetched by the FOR statement:

```
SET FLAGS 'TRACE, NOPREFIX';
BEGIN
-- Declare a variable to hold the number of the current row.
    DECLARE  :currow INTEGER;

-- The :jobrec variable represents a record that holds columns from the
-- selected row.
    FOR :jobrec
        AS EACH ROW OF TABLE CURSOR JOB_CURSOR FOR
        SELECT MINIMUM_SALARY FROM JOBS
              WHERE MINIMUM_SALARY < 20000
      DO
-- Update the current row in the JOB_CURSOR.
         UPDATE JOBS
                SET MINIMUM_SALARY = MINIMUM_SALARY * 1.10
                WHERE CURRENT OF JOB_CURSOR;

-- Use the GET DIAGNOSTICS statement to retrieve the number of rows that have
-- been fetched by the FOR statement.
         GET DIAGNOSTICS
             :currow = CURRENT_ROW;

-- Use the TRACE statement to print out the number of the current row.
         TRACE 'Current Row: ', :currow;

    END FOR;

END;
Current Row: 1
Current Row: 2
Current Row: 3
Current Row: 4
Current Row: 5
Current Row: 6
```

In the preceding example, the TRACE statement prints the value of the current row to the terminal.

For more information about the GET DIAGNOSTIC statement, see *Oracle Rdb7 SQL Reference Manual*.

12.9 Handling Exception and Completion Conditions

SQL statements within a compound statement can encounter exception conditions or completion conditions. SQL treats completion conditions differently than it does exception conditions, as follows:

- When the execution of an SQL statement returns an exception condition, such as a constraint violation, SQL aborts the execution of that compound statement. You can pass information about that SQL statement to the application by using the SIGNAL statement.
- When the execution of an SQL statement results in a completion condition, such as no data returned, SQL continues execution of the compound statement. You can retrieve the status of the last SQL statement that executed by using the GET DIAGNOSTICS statement.

12.9.1 Retrieving Exception Conditions

When an SQL statement in a compound statement detects an error, you can use the SIGNAL statement to raise an exception and return the status to the application.

You specify, as an argument to the SIGNAL statement, a character value expression which Oracle Rdb uses as the value of the SQLSTATE status parameter. When Oracle Rdb encounters a SIGNAL statement, it terminates the current compound statement or routine and all calling routines and returns control to the application. In doing so, Oracle Rdb passes the SQLSTATE value you specify to the application.

The contents of the character string must conform to the ANSI/ISO SQL standard for SQLSTATE values. That is, it must contain only Latin uppercase letters (A through Z) or digits (0 through 9). For example, to signal the application if the numeric value is out of range, use the following statement:

```
SIGNAL '22003';
```

Oracle Rdb returns the SQLSTATE value of 22003 to the application. As it does so, it maps the SQLSTATE value to an SQLCODE value. In the previous example, the SQLSTATE value maps to an SQLCODE value of -304. If the SQLSTATE value can map to more than one SQLCODE value or if the SQLSTATE value is unknown, Oracle Rdb returns an SQLCODE value of -1042.

The following example shows how to use SIGNAL when a compound statement may attempt to calculate division by zero.

```
-- Calculate the difference between two job titles and put the results in the
-- table PAY_ADJUST.

BEGIN
  DECLARE :job_code1, :job_code2 CHAR (5);
  DECLARE :min_sal, :min_sal2 INTEGER;
  DECLARE :diff INTEGER;
  -- In a stored procedure, you pass the values using parameters,
  -- rather than hard coding values.
  SET :job_code1 = 'PRGM';
  SET :job_code2 = 'MENG';

  SELECT MINIMUM_SALARY INTO :min_sal FROM JOBS
       WHERE JOB_CODE = :job_code1;
  SELECT MINIMUM_SALARY INTO :min_sal2 FROM JOBS
       WHERE JOB_CODE = :job_code2;

  IF :min_sal - :min_sal2 = 0
      -- If condition is not trapped, it results in division by 0 in ELSE
      -- clause.
      THEN SIGNAL '22012';
      -- Calculate the difference in minimum salaries and store the percentage
      --   in the PAY_ADJUST table.
      ELSE SET :diff = :min_sal - :min_sal2;
           INSERT INTO PAY_ADJUST
                    (JOB_CODE1, JOB_CODE2, DIFF)
                VALUES (:job_code1, :job_code2, :min_sal / :diff);
  END IF;
END;
%RDB-E-SIGNAL_SQLSTATE, routine "(unnamed)" signaled SQLSTATE "22012"
```

In the preceding example, Oracle Rdb returns the SQLSTATE value of 22012 to indicate that division by 0 will occur.

For more information about the SIGNAL statement, see the *Oracle Rdb7 SQL Reference Manual*.

12.9.2 Retrieving Completion Conditions

The GET DIAGNOSTICS statement extracts diagnostic information about the execution of the previous SQL statement. It provides diagnostic information from the SQLSTATE or SQLCODE status parameter.

Use the EXCEPTION . . . RETURNED_SQLSTATE clause to retrieve the SQLSTATE status information, and the EXCEPTION . . . RETURNED_ SQLCODE clause to retrieve the SQLCODE status information.

The following example uses the EXCEPTION . . . RETURNED_SQLSTATE clause to return the value of the SQLSTATE status parameter:

```
SET FLAGS 'TRACE, NOPREFIX';

BEGIN ATOMIC
     DECLARE  :mgrid CHAR(5);
     DECLARE  :cur_mgrid CHAR(5);
     DECLARE  :dept_code CHAR(4);
     DECLARE  :sqlstate_var CHAR(5);

     SET :mgrid = '00166';
     SET :dept_code = 'SALE';

     SELECT MANAGER_ID INTO :cur_mgrid FROM DEPARTMENTS
            WHERE DEPARTMENT_CODE = :dept_code;

-- The GET DIAGNOSTICS statement returns an SQLSTATE code of 00000 because the
-- SELECT statement executes successfully.
     GET DIAGNOSTICS
            EXCEPTION 1 :sqlstate_var = RETURNED_SQLSTATE;

     TRACE 'After SELECT, SQLSTATE is: ', :sqlstate_var;

-- Test the value of SQLSTATE and take some action if it is 00000.
     IF :sqlstate_var = '00000'
     THEN
        IF :cur_mgrid <> :mgrid
        THEN
           UPDATE DEPARTMENTS
                  SET MANAGER_ID = :mgrid
                  WHERE DEPARTMENT_CODE = 'Ssss';

-- Because there is no department with a DEPARTMENT_CODE 'Ssss', the
-- GET DIAGNOSTICS statement returns an SQLSTATE code of 02000.

           GET DIAGNOSTICS
                  EXCEPTION 1 :sqlstate_var = RETURNED_SQLSTATE;

           TRACE 'After UPDATE, SQLSTATE is: ', :sqlstate_var;
        END IF;
    END IF;
END;
After SELECT, SQLSTATE is: 00000
After UPDATE, SQLSTATE is: 02000
```

See the *Oracle Rdb7 SQL Reference Manual* for more information about the GET DIAGNOSTICS statement and the possible values of the SQLSTATE status parameter.

13

Using Stored Routines

This chapter describes stored routines (which are stored procedures and stored functions) and explains how to store routines in a stored module within an Oracle Rdb database and how to call the stored routines for execution by an application program. In the sections that follow, you will become familiar with how to:

- Determine what a stored routine is
- Recognize the benefits of using stored routines
- Create a stored module, which contains stored routines
- Invoke a stored procedure
- Invoke a stored function
- Delete a stored routine
- Track dependencies
- Recognize when stored routines are invalidated
- Revalidate or re-create invalidated stored routines

13.1 What Are Stored Routines?

A **stored routine** is a stored procedure or stored function.

A **stored procedure** is a set of operations performed on an Oracle Rdb database by one or more SQL statements. It accepts a set of input parameters and returns results through output parameters.

A **stored function** is a set of operations performed on an Oracle Rdb database by one or more SQL statements. It accepts a set of input parameters and returns a single result through the RETURN statement.

The SQL statements included as part of stored routines execute as a unit to perform a wide variety of database operations. Stored routines use the program-like, procedural capabilities of multistatement procedures. This permits established operations, once solely the domain of applications, to be shifted to the database where they can serve the programming community in much the same way host language libraries do.

Stored routines are like 3GL application procedures. They reside within a **stored module** that is the object of compilation and they encapsulate an operation, such as an update, delete, or insert operation. Unlike application procedures, you write stored routines in SQL, rather than in a 3GL program language such as C or Fortran.

Stored modules reside as schema objects inside an Oracle Rdb database, like a table or view.

In contrast, **nonstored modules** reside in a file outside an Oracle Rdb database. Using nonstored modules, you can refer to more than one database, but using stored modules, you can refer only to the database in which the stored module resides.

In other ways, stored and nonstored modules are similar. The SQL module language you use to create nonstored modules is similar to the SQL syntax you use to create stored modules. Just as stored modules must contain at least one stored routine, nonstored modules must contain one or more procedures.

Refer to Chapter 3 and Chapter 4 for information about how to create and use modules stored in files outside an Oracle Rdb database.

13.2 Understanding the Benefits of Storing Routines in a Database

The benefits of storing routines in a database include the following:

- Encapsulated operations

  Stored routines let you place an operation (or set of operations) in the database for use by other users. A set of stored routines acts like a library of executable objects that can be linked together by an application. For example, suppose ADD EMPLOYEE is an operation of the application requiring several pieces of input information. Without knowing the intricate inner actions of a stored routine, an application can call a single routine to add an employee record to the database.

Currently, programmers do this by using object libraries; however, stored routines can relieve the user from maintaining object libraries and make the functional pieces visible to SQL queries (unlike libraries of object modules).

- Inherited privileges

  When you define a stored routine, you must have access to base objects (such as tables, views, and constraints) to which the stored routine refers. If you specify an authorization identifier when you create the stored routine, other users of the stored routine inherit the definer's access rights. This enables them to use the defined routine even if they do not have direct access to the routine's base objects. The invoker is not required to have any database object privileges other than the EXECUTE privilege required to invoke the stored routine. Users can perform fixed actions even though they have no access privileges to the underlying tables.

- Client/server processing

  Client systems cannot afford to be burdened with locally storing and maintaining database requests such as those containing SQL statements. With stored routines on the server system, client system applications can easily access and perform complex database operations contained within stored routines. Access occurs by attaching to the Oracle Rdb database in which the stored routine resides, and specifying the name of the routine and its parameters.

- Better control over metadata dependencies

  When you store routines in a database and you use those routines, Oracle Rdb tracks the metadata objects to which the routine refers.

13.3 Creating Stored Modules

You use the CREATE MODULE statement to create a stored module in an Oracle Rdb database, just as you use the CREATE TABLE statement to create a database table. Within the CREATE MODULE statement, you define one or more stored routines with the SQL language. The module serves as SQL's mechanism for storing routines in an Oracle Rdb database.

Example 13–1 shows the definition of a stored module called DATA_UPDATE.

Example 13–1 Creating a Stored Module

```
CREATE MODULE DATA_UPDATE ❶
       LANGUAGE SQL ❷
       AUTHORIZATION LAWLER ❸
       DECLARE LOCAL TEMPORARY TABLE MODULE.DATA_UPDATE_TAB
        (EMPLOYEE_ID ID_DOM,
         SALARY INTEGER(2))  ❹
       <routine definitions>  ❺
END MODULE; ❻
```

When you create a stored module you use the following clauses (keyed to the numbered callouts in Example 13–1):

❶ CREATE MODULE module-name

The CREATE MODULE statement creates a module, called DATA_UPDATE, as a persistent object in an Oracle Rdb database. Names for stored modules must be unique in an application. Applications that access routines from both stored and nonstored modules cannot have a nonstored module with the same name as a stored module used by the application.

❷ LANGUAGE SQL clause

The SQL argument to the LANGUAGE clause indicates that the routines in the stored module are invoked by an SQL statement.

In contrast, the argument to the LANGUAGE clause in a nonstored module identifies the host language in which the program calling a module's procedures is written.

❸ AUTHORIZATION clause

The authorization identifier, specified as LAWLER in the AUTHORIZATION clause, enables Oracle Rdb to identify the author or definer of the module.

When you specify an authorization identifier in the definition of a stored module, that stored module is called a **definer's rights module**. This type of module enables any user who has EXECUTE privilege on the module to execute any of the module's routines without privileges on any of the underlying schema objects that the routine references. The routines execute under the user name of the module definer, not the user name of the person executing the routine. This ability to allow users access to schema objects through a call to a stored routine without having direct access to those schema objects is a key benefit.

In contrast, when you omit the AUTHORIZATION clause in the definition of a stored module, that stored module is called an **invoker's rights module**. In this type of module, users who have EXECUTE privilege on a particular module must also have privileges to all of the underlying schema objects associated with any of the routines that they want to execute.

❹ Declared local temporary tables

You can use a declared local temporary table in a stored module. You can only refer to the table from within the module in which it is declared. The metadata and data in a declared local temporary table do not persist outside the stored module. For more information about temporary tables, see the *Oracle Rdb7 Guide to Database Design and Definition*.

❺ Routine definitions

The routine definition specifies the name of the routine, parameters, and one simple statement or one compound statement.

See Section 13.3.1 for information on creating the definition for a stored procedure. See Section 13.3.2 for information on creating the definition for a stored function.

❻ END MODULE

You must identify the end of a stored module definition with the END MODULE keywords.

13.3.1 Creating Stored Procedures

You create a stored procedure by including a stored procedure definition in a CREATE MODULE statement.

Example 13–2 shows the definition of the NEW_SALARY_PROC stored procedure, within the DATA_UPDATE module.

Example 13–2 Creating a Stored Procedure

```
CREATE MODULE DATA_UPDATE
       LANGUAGE SQL
       AUTHORIZATION LAWLER

       DECLARE LOCAL TEMPORARY TABLE MODULE.DATA_UPDATE_TAB
        (EMPLOYEE_ID ID_DOM,
        SALARY INTEGER(2))
```

(continued on next page)

Example 13–2 (Cont.) Creating a Stored Procedure

```
      PROCEDURE NEW_SALARY_PROC  ❶
        (:ID CHAR (5),           ❷
         :NEW_SALARY INTEGER (2)); ❷

      BEGIN                ❸
        UPDATE SALARY_HISTORY       ❸
            SET SALARY_END = CURRENT_TIMESTAMP
                WHERE EMPLOYEE_ID = :ID AND SALARY_END IS NULL;
        INSERT INTO SALARY_HISTORY (EMPLOYEE_ID, SALARY_AMOUNT,
            SALARY_START, SALARY_END)
            VALUES (:ID, :NEW_SALARY, CURRENT_TIMESTAMP, NULL);

       INSERT INTO MODULE.DATA_UPDATE_TAB
            (EMPLOYEE_ID, SALARY)
            SELECT EMPLOYEE_ID, SALARY_AMOUNT FROM SALARY_HISTORY
            WHERE CAST(SALARY_START AS DATE ANSI) =
                  CURRENT_DATE;
      END;                ❸
END MODULE;
```

When you create a stored procedure, you use the following clauses (keyed to the numbered callouts in Example 13–2):

❶ PROCEDURE clause

The procedure clause specifies the name of the stored procedure.

Procedure names must be unique within the module definition and across the definitions of other routines stored in modules in an Oracle Rdb database and must also be unique from external routine names defined in the same database.

❷ Parameter list

The parameter list of the procedure includes a parameter mode, a parameter name, and an SQL data type:

- Stored procedure parameters have three parameter modes: IN, OUT, and INOUT.

  If you omit the mode from the parameter declaration, (as in Example 13–2) SQL determines the mode of a parameter by the parameter's usage. To determine the parameter mode of a stored procedure parameter, ask the question, "Do the SQL statements in the stored procedure accept a value (IN parameter mode), do they produce a value (OUT parameter mode), or do they perform both operations (INOUT parameter mode)?"

- The name of each parameter in a stored procedure definition must be unique within the procedure.
- You can specify any SQL data type except LIST OF BYTE VARYING in a stored procedure. You can also specify a domain name instead of a data type in the declaration. For example, you can declare :ID CHAR (5) as :ID ID_DOM.

Note the following points about parameters in stored procedures:

- You cannot declare any status parameters (SQLCODE, SQLSTATE, or SQLCA) in a stored procedure. However, you can examine the values of SQLCODE or SQLSTATE by using the GET DIAGNOSTIC statement in the stored procedure. Section 12.9 describes how to use the GET DIAGNOSTIC statement to return the SQLCODE and SQLSTATE values and to handle completion conditions. It also describes how to use the SIGNAL statement to handle exception conditions.

  When you call a stored procedure from a nonstored procedure, as shown in Example 13–4, the nonstored procedure must declare a status parameter, as you must for any procedure in a nonstored module.
- Stored procedures support null values. As a result, you cannot use indicator parameters with stored procedure parameters.

  Be aware, however, that if you expect the called stored procedure to return a NULL value, you must be sure to declare, in the nonstored procedure that calls the stored procedure, an indicator parameter to ensure that you can process the NULL value. The indicator parameter indicates whether or not the value stored in its corresponding main parameter is NULL.

❸ BEGIN . . . END block

You can use one simple statement or one compound statement in a stored procedure. This example shows a compound statement bounded by the BEGIN . . . END block.

See Section 13.4 for information about invoking stored procedures. Refer to the *Oracle Rdb7 SQL Reference Manual* for additional information about creating stored procedures with the CREATE MODULE statement.

13.3.2 Creating Stored Functions

You create a stored function by including a stored function definition in a CREATE MODULE statement.

Example 13–3 shows the definition of the CHECK_SALARY_RANGE_FUNC stored function within the CHECK_SALARY_MOD module. The function checks to see if an employee's salary is within the correct range for the job title.

Example 13–3 Creating a Stored Function

```
CREATE MODULE CHECK_SALARY_MOD
       LANGUAGE SQL

       FUNCTION CHECK_SALARY_RANGE_FUNC ❶
         (:JOB_TITLE CHAR(20), :CUR_SALARY INTEGER(2))    ❷
         RETURNS INTEGER;                                 ❸
       BEGIN                                              ❹
         DECLARE :min_sal, :max_sal INTEGER(2);
         DECLARE :sal_check  INTEGER;

         SELECT MINIMUM_SALARY, MAXIMUM_SALARY INTO :min_sal, :max_sal
                FROM JOBS
                WHERE JOB_TITLE = :job_title;
         RETURN (CASE                                     ❺
                    WHEN(:cur_salary >= :min_sal  AND :cur_salary <= :max_sal)
                        THEN 0
                        ELSE 1
                 END );
       END;                                               ❹

END MODULE;
```

When you create a stored function, you use the following clauses (keyed to the numbered callouts in Example 13–2):

❶ FUNCTION clause

The function clause specifies the name of the stored function.

Function names must be unique within the module definition and across the definitions of other routines stored in modules in an Oracle Rdb database and must also be unique from external routine names defined in the same database.

❷ Parameter list

The parameter list of the function includes a parameter mode, a parameter name, and an SQL data type:

- Stored functions allow only the IN parameter mode.
- The name of each parameter in a stored function definition must be unique within the function.
- You can specify any SQL data type except LIST OF BYTE VARYING. You can specify a domain name instead of a data type in the declaration. For example, you can declare :JOB_TITLE CHAR(20) as :JOB_TITLE JOB_TITLE_DOM.

Note the following points about parameters in stored functions:

- You cannot declare any status parameters (SQLCODE, SQLSTATE, or SQLCA) in a stored function. However, you can examine the values of SQLCODE or SQLSTATE by using the GET DIAGNOSTIC statement in the stored function. Section 12.9 describes how to use the GET DIAGNOSTIC statement to return the SQLCODE and SQLSTATE values and to handle completion conditions. It also describes how to use the SIGNAL statement to handle exception conditions.

  When you call a stored function from a nonstored procedure, the nonstored procedure must declare a status parameter as you must for any procedure in a nonstored module.

- Stored functions support null values. As a result, you cannot use indicator parameters with stored function parameters.

  Be aware, however, that if you expect the called stored function to return a NULL value, you must be sure to declare, in the nonstored procedure that calls the stored function, an indicator parameter to ensure that you can process the NULL value. The indicator parameter indicates whether or not the value stored in its corresponding main parameter is NULL.

❸ RETURNS clause

The RETURNS clause, which is required, specifies the data type of the value returned by the function. In this example, the function returns an integer.

❹ BEGIN . . . END block

This example shows a compound statement bounded by the BEGIN . . . END block.

You can use one simple statement or one compound statement in a stored procedure. However, because stored functions return values, and you retrieve those values with a RETURN statement, you should use a compound statement with stored functions.

❺ RETURN statement

The RETURN statement returns the value of the stored function. The data type of the value returned by the RETURN statement must be compatible with the data type specified in the RETURNS clause.

In a stored function, you can use any statement that you can use in a compound statement, except SET TRANSACTION, COMMIT, or ROLLBACK. If a stored function calls a stored procedure, the procedure cannot execute any of these statements.

See Section 13.5 for information about invoking stored functions. Refer to the *Oracle Rdb7 SQL Reference Manual* for additional information about creating stored functions with the CREATE MODULE statement.

13.3.3 Creating a Stored Function to Generate New Sequence Numbers

Application programmers are often faced with the problem of generating unique sequence numbers for use as primary keys. One solution is to use stored functions to maintain a unique sequence numbered table and issue the next values.

While there are other solutions, such as using an AFTER INSERT trigger, the main advantage to using a stored function is that the final value is inserted and no subsequent update is required to the data row. This saves unnecessary validation queries for the primary key field.

To implement this solution, you can use a table, NEXT_KEY_TABLE, to maintain a list of key names and their current values. You initially load one key value into the table. Then, each time you call the stored function, it fetches the value from the NEXT_KEY_TABLE and returns the next value.

The following example shows the domain and table definitions and shows how to load the first value into the table:

```
SQL> CREATE DOMAIN KEY_NAME
cont>   CHAR(31)
cont>   CHECK (VALUE IS NOT NULL)
cont>   NOT DEFERRABLE;
SQL> --
SQL> CREATE TABLE NEXT_KEY_TABLE
cont>   ( NEXT_KEY_VAL INTEGER NOT NULL NOT DEFERRABLE,
cont>     NEXT_KEY_NAME KEY_NAME UNIQUE NOT DEFERRABLE);
SQL> --
```

```
SQL> INSERT INTO next_key_table (next_key_name, next_key_val)
cont>   VALUES ('EMPLOYEE_ID', 0);
1 row inserted
```

The following example shows the stored function definition:

```
SQL> CREATE MODULE TOOLS
cont>   LANGUAGE SQL
cont>   FUNCTION NEXT_KEY (IN :KEY_NAME KEY_NAME)
cont>   RETURNS INTEGER;
cont>   BEGIN
cont>     DECLARE :rc, :new_val INTEGER DEFAULT 0;
cont>     DECLARE :key_name_upper key_name DEFAULT UPPER(:KEY_NAME);
cont>     DECLARE :invalid_parameter CONSTANT CHAR(5) = '22023';
cont>
cont>     UPDATE NEXT_KEY_TABLE
cont>     SET NEXT_KEY_VAL = NEXT_KEY_VAL + 1
cont>         WHERE NEXT_KEY_NAME = :key_name_upper
cont>         RETURNING NEXT_KEY_VAL INTO :NEW_VAL;
cont>
cont>     GET DIAGNOSTICS :rc = ROW_COUNT;
cont>     TRACE 'NEXT_KEY is ', COALESCE(:new_val, 'NULL'), ', RC is ', ;rc;
cont>
cont>     IF :rc = 0 THEN
cont>        TRACE 'No entry exists for KEY_NAME: ', :key_name_upper;
cont>        SIGNAL :invalid_parameter;
cont>     ELSE
cont>        TRACE 'Returning new value for ', :key_name_upper,:new_val;
cont>        RETURN :new_val;
cont>     END IF;
cont>   END;
cont> END MODULE;
```

You can invoke the function by specifying it in the VALUES clause of an INSERT statement, as shown in the following example:

```
SQL> INSERT INTO employee (employee_id, last_name, birthday)
cont>   VALUES (next_key('EMPLOYEE_ID'), 'Smith', DATE'1970-1-1');
```

13.4 Invoking Stored Procedures

You can invoke a stored procedure using a CALL statement from a simple or compound statement. You can call one stored procedure from another stored procedure or from a stored function.

Example 13–4 shows a code segment from an SQL module that contains the nonstored procedure, CALL_NEW_SALARY, which calls the stored procedure defined in Example 13–2.

Example 13–4 Calling a Stored Procedure

```
PROCEDURE CALL_NEW_SALARY
  (:ID CHAR(5),     ❶
   :ID_IND SMALLINT,  ❷
   :NEW_SALARY INTEGER (2),  ❶
   :NEW_SALARY_IND SMALLINT,  ❷
    SQLCODE);  ❸

  CALL NEW_SALARY_PROC ❹
   (:ID INDICATOR :ID_IND, :NEW_SALARY INDICATOR :NEW_SALARY_IND); ❺
```

The following list is keyed to the numbered callouts in Example 13–4:

❶ The main parameters in the CALL_NEW_SALARY procedure are named :ID and :NEW_SALARY.

❷ The indicator parameters are named :ID_IND and :NEW_SALARY_IND.

Because parameters in stored procedures allow null values, each main parameter should have a corresponding indicator parameter. Otherwise, if the stored procedure returns a null value into a main parameter, SQL returns an error and the procedure fails. Providing an indicator parameter solves the problem that stored procedure parameters allow null values but nonstored parameters do not.

❸ All nonstored procedures must contain a status parameter.

Stored and nonstored procedures differ in the requirement to declare a status parameter. Nonstored procedures require them. In stored procedures, status parameters are not needed and thus are not allowed.

❹ The CALL statement invokes a stored procedure.

Invoke the NEW_SALARY_PROC stored procedure defined in Example 13–2 with the CALL statement.

❺ The CALL statement passes the :ID and :NEW_SALARY variables as arguments to the called stored procedure.

The data types of the two variables used in the CALL statement must match the data types used in the NEW_SALARY_PROC procedure that it calls.

A CALL statement in a simple statement can pass a list of literals, parameter values (parameter markers for dynamic execution), or variables to the called stored procedure, or the CALL statement can pass no arguments. A CALL statement in a compound statement can pass any value expression, except dbkeys and aggregate functions, to the called stored procedure or the CALL statement can pass no arguments.

Refer to the *Oracle Rdb7 SQL Reference Manual* for additional information about the CALL statement and the type of arguments it can pass.

13.5 Invoking Stored Functions

You invoke a stored function by using the function name as a value expression in an SQL statement. You can invoke a stored function from wherever a value expression is allowed, except from a trigger.

Example 13–5 shows a compound statement that uses an IF statement to call the stored function defined in Example 13–3. The IF statement calls the stored function CHECK_SALARY_RANGE and passes two parameters, :salrec.JOB and :salrec.SALARY, to the function. If the result of the function is equal to 1, the IF statement inserts the last name of the employee into the table SALARY_ADJUST.

Example 13–5 Invoking a Stored Function

```
BEGIN
  DECLARE :lname char(14);

  FOR :salrec
      AS EACH ROW OF TABLE CURSOR SAL_CURSOR FOR
      SELECT LAST_NAME, JOB, SALARY  FROM CURRENT_INFO
   DO
      SET :lname = :salrec.LAST_NAME;
   -- Call the stored function.
      IF CHECK_SALARY_RANGE(:salrec.JOB, :salrec.SALARY) = 1
        THEN
         INSERT INTO SALARY_ADJUST VALUES (:lname);
      END IF;
  END FOR;
END;
```

13.6 Deleting Stored Routines

The DROP MODULE statement deletes a stored module and its routines from an Oracle Rdb database. You must have the DROP privilege on a module to delete it.

Example 13–6 shows how to delete the DATA_UPDATE module.

Example 13–6 Deleting a Stored Module

```
DROP MODULE DATA_UPDATE;
```

To delete only a stored procedure, instead of the entire module, use the DROP PROCEDURE statement, as shown in the following example:

```
DROP PROCEDURE NEW_SALARY_PROC;
```

To delete only a stored function, instead of the entire module, use the DROP FUNCTION statement, as shown in the following example:

```
DROP FUNCTION NEW_FUNC;
```

When you use the RESTRICT keyword (the default), Oracle Rdb prevents the removal of a stored routine to which other stored routines within the database refer.

When you use the CASCADE keyword to drop a stored routine, Oracle Rdb executes the DROP statement and invalidates any stored routines that refer to the dropped routine.

Refer to *Oracle Rdb7 Guide to Database Design and Definition* for information about the privileges required for stored procedures. Refer to the *Oracle Rdb7 SQL Reference Manual* for additional information about the DROP MODULE, DROP PROCEDURE, and DROP FUNCTION statements.

13.7 Tracking Stored Routine Dependencies

To ensure that a stored routine can be compiled successfully at run time, Oracle Rdb tracks the underlying schema objects on which a stored routine depends. For example, Oracle Rdb might not be able to compile a stored routine properly if one of its underlying objects, such as a table or column, is deleted or altered in some way. The mechanism that Oracle Rdb uses to identify the objects on which stored routine depends for execution is called **dependency tracking**.

When you enter the CREATE MODULE statement, Oracle Rdb checks the module and each of its routines to determine if they refer to a column, constraint, domain, function, table, or view. When Oracle Rdb finds one or more of these objects, called a **referenced object**, Oracle Rdb records the dependency information in the RDB$INTERRELATIONS system table.

A stored routine, which relies on the referenced object for proper execution, is called a **dependent object**.

Table 13–1 lists the objects for which Oracle Rdb stores metadata dependency information and the dependency type for each of these objects.

Table 13–1 Dependency Tracking Table

	Information Stored in RDB$INTERRELATIONS					
Referenced Object	**Object Name**	**Subobject Name**	**Entity Name1**	**Entity Name2**	**Usage**	**Constraint Name**
Column	Table	Column	Module	Procedure or Function	P/F	
Constraint			Module	Procedure or Function	P/F	Constraint
Constraint			Module		DE	Constraint
Domain		Domain	Module	Procedure or Function	P/F	
Function (external)		Function	Module	Procedure or Function	P/F	
Function (stored)	Module	Function	Module	Procedure or Function	P/F	
Procedure (external)		Function	Module	Procedure or Function	P/F	
Procedure (stored)	Module	Function	Module	Procedure or Function	P/F	
Table	Table		Module	Procedure or Function	P/F, LS	
Table	Table		Module		DR	
View	View		Module	Procedure or Function	P/F, LS	
View	View		Module		DR	

Key to dependency types in Usage column:

DE—Default evaluating
DR—Default reserving
LS—Language semantics
P/F—Procedure or Function

Oracle Rdb stores the dependency information in RDB$INTERRELATIONS when a domain exists in a procedure block. In contrast, if a domain is defined in the parameter list of a stored routine, Oracle Rdb tracks the dependency in RDB$PARAMETERS, not in RDB$INTERRELATIONS.

A critical element of the dependency information stored in the system table RDB$INTERRELATIONS is the dependency type. Although the referenced object names enable Oracle Rdb to uniquely identify a metadata dependency, the concept of dependency type dictates how Oracle Rdb invalidates a dependent object when underlying referenced objects are added or dropped.

An invalidated routine cannot be invoked until all referenced objects are restored as they existed before invalidation occurred.

The following sections describe the four stored routine dependency types: procedure, language semantics, default reserving, and default evaluating.

13.7.1 Procedure Dependency Type

Any referenced object specified in the procedure block of a compound statement can create a **procedure dependency** or **function dependency** between that object and the stored routine that refers to that object. In the following example of a stored procedure, the CANDIDATES table and each of its specified columns (LAST_NAME, FIRST_NAME, MIDDLE_INITIAL, and CANDIDATE_STATUS) create a procedure dependency.

```
PROCEDURE SIMPLE_P

BEGIN
    INSERT INTO CANDIDATES
 (LAST_NAME,FIRST_NAME,MIDDLE_INITIAL, CANDIDATE_STATUS)
    VALUES
 ('test_lname','test_fname','t','test_status');
END;
```

When you query RDB$INTERRELATIONS as shown in Example 13–7, you can see that Oracle Rdb stores procedure dependencies for the five objects in the CANDIDATES table to which the INSERT statement in the preceding example refers.

Example 13–7 Examining Procedure Dependency Type

```
SQL> SELECT * FROM RDB$INTERRELATIONS WHERE RDB$OBJECT_NAME = 'CANDIDATES';
 RDB$OBJECT_NAME                   RDB$SUBOBJECT_NAME
   RDB$ENTITY_NAME1                  RDB$ENTITY_NAME2
      RDB$USAGE                         RDB$FLAGS
         RDB$CONSTRAINT_NAME               RDB$SECURITY_CLASS
 CANDIDATES ❶
   DEPENDENCY_LIST ❸                  SIMPLE_P ❹
      Procedure  ❺                                  5
                                          NULL
```

(continued on next page)

Example 13–7 (Cont.) Examining Procedure Dependency Type

```
CANDIDATES ❶                    LAST_NAME ❷
  DEPENDENCY_LIST ❸               SIMPLE_P ❹
     Procedure ❺                                5
                                      NULL
CANDIDATES ❶                    FIRST_NAME ❷
  DEPENDENCY_LIST ❸               SIMPLE_P ❹
     Procedure ❺                                5
                                      NULL
CANDIDATES ❶                    MIDDLE_INITIAL ❷
  DEPENDENCY_LIST ❸               SIMPLE_P ❹
     Procedure ❺                                5
                                      NULL
CANDIDATES ❶                    CANDIDATE_STATUS ❷
  DEPENDENCY_LIST ❸               SIMPLE_P ❹
     Procedure ❺                                5
                                      NULL
```

The following list is keyed to the numbered callouts in Example 13–7:

❶ Object name: Table name

❷ Subobject name: Column name

❸ Entity name1: Module name

❹ Entity name2: Procedure name

❺ Usage: Dependency type

If the columns in the CANDIDATES table are referred to by a stored function, this field would display "Function" rather than "Procedure".

13.7.2 Language Semantic Dependency Type

A procedure block of a stored routine that refers to a table or view but refers to the columns implicitly creates a **language semantic dependency** between the stored routine and the referenced object.

Natural join and SELECT * operations also cause language semantic dependencies.

In the following example, because the INSERT statement does not explicitly refer to any columns in the CANDIDATES table, Oracle Rdb sets up a language semantic dependency.

```
PROCEDURE LANG_SEMANTICS_P

BEGIN
    INSERT INTO CANDIDATES VALUES
       ('test_lname','test_fname','t','test_status');
END;
```

Because the INSERT statement does not include a column list, SQL derives the column order and names from the current table definition. The column list omission causes Oracle Rdb to establish a language semantic dependency between the CANDIDATES table and the LANG_SEMANTICS_P stored procedure.

A DELETE statement does not refer to columns explicitly, only implicitly.

Because the LANG_SEMANTICS_P stored procedure depends explicitly on the CANDIDATES table and implicitly on all of its columns, all the columns in the CANDIDATES table are considered referenced objects. Thus, dropping the table using the CASCADE keyword or adding a column to that table invalidates the stored procedure. Refer to Section 13.9.2 for information about re-creating a stored routine invalidated due to a language semantic dependency.

13.7.3 Transaction Dependency Types

Oracle Rdb recognizes two transaction dependency types:

- Default reserving dependency

  You create a **default reserving dependency** on the referenced table when you specify a table in a DECLARE TRANSACTION RESERVING statement or in a SET TRANSACTION RESERVING statement.

- Default evaluating dependency

  You create a **default evaluating dependency** on a referenced constraint when you specify a constraint in a DECLARE TRANSACTION EVALUATING statement or in a SET TRANSACTION EVALUATING statement.

In the following example, Oracle Rdb sets up a default reserving dependency for the JOBS table and a default evaluating dependency for the JOB_CODE_REQUIRED constraint.

```
CREATE MODULE DEPENDENCY_LIST LANGUAGE SQL

-- Create default reserving and default evaluating dependencies.
DECLARE TRANSACTION READ WRITE
        RESERVING JOBS
        FOR SHARED READ EVALUATING JOB_CODE_REQUIRED AT VERB TIME
```

When you query RDB$INTERRELATIONS as shown in Example 13–8, you can see that Oracle Rdb stores default reserving dependencies for the JOBS table and a default evaluating dependency for the JOB_CODE_REQUIRED constraint to which the DECLARE TRANSACTION statement refers.

Example 13–8 Examining Transaction Dependency Type

```
SQL> SELECT * FROM RDB$INTERRELATIONS WHERE RDB$OBJECT_NAME = 'JOBS';
   .
   .
   .
   DEPENDENCY_LIST ❷
      Default Txn Evaluating ❸                    1
         JOB_CODE_REQUIRED ❹                NULL
 JOBS ❶
   DEPENDENCY_LIST ❷
      Default Txn Reserving ❸                     1
                                          NULL
```

The following list is keyed to the numbered callouts in Example 13–8:

❶ Object name: Table name

❷ Entity name1: Module name

❸ Usage: Dependency type

❹ Constraint name

Unlike procedure or function and language semantic dependencies, which are procedure-based tracking mechanisms, transaction dependencies affect stored routines at the module level. Example 13–8 shows that Oracle Rdb stores a module name only for both transaction dependency types. It does not store any procedure or function name because the DECLARE TRANSACTION statement is linked to all routines in the module, not to any one in particular. This has implications when a table or constraint is deleted or altered.

If you drop a table using the CASCADE keyword and a DECLARE TRANSACTION statement of a stored module refers to that table, Oracle Rdb deletes the table and invalidates all the routines in the module. Thus, if you execute a routine that depends on the table, Oracle Rdb returns an error and the routine fails. Because the transaction dependency types are module based, each of the routines in the module fails whenever a stored routine in that module is invoked.

13.8 Invalidating Stored Routines

Invalidation of a stored routine occurs when one or more underlying schema objects on which the stored routine relies are changed or removed. Oracle Rdb stores the invalidation status in the system tables. Invalidation alerts you that a stored routine might fail if the stored routine's underlying schema objects are not re-created.

Reference Reading

Refer to the description of the Oracle Rdb system tables in Oracle_ Rdb Help. Read the descriptions of the RDB$ROUTINES and RDB$INTERRELATIONS system tables for information about which flags Oracle Rdb sets during invalidation.

Table 13–2 shows the data definition statements that can cause stored routines to be invalidated.

Table 13–2 Statements Causing Stored Routine Invalidation

Object Type	SQL Statement	Statement Fails?	Routine Invalidated?	Dependency Type
Column	ALTER TABLE DROP COLUMN	Yes	No	P/F or LS
	ALTER TABLE ADD COLUMN	No	Yes	LS
	ALTER TABLE ADD COLUMN	No	No	P/F
Constraint	ALTER TABLE DROP CONSTRAINT	Yes	No	P/F
	ALTER TABLE ADD CONSTRAINT	No	No	P/F or DE
Domain	ALTER DOMAIN (in parameter list)	Yes	No	Does not apply[1]
	ALTER DOMAIN (in procedure block)	No	No	P/F[2]

[1]Oracle Rdb stores this domain parameter list dependency in the RDB$PARAMETERS system table, not in RDB$INTERRELATIONS.

[2]Oracle Rdb stores a procedure dependency in the RDB$INTERRELATIONS system table when a domain exists in a procedure block.

Key to dependency types:

DE—Default evaluating
DR—Default reserving
LS—Language semantics
P/F—Procedure or Function

(continued on next page)

Table 13–2 (Cont.) Statements Causing Stored Routine Invalidation

Object Type	SQL Statement	Statement Fails?	Routine Invalidated?	Dependency Type
	DROP DOMAIN	Yes	No	P/F
Function	DROP FUNCTION CASCADE	No	Yes	P/F
	DROP FUNCTION RESTRICT	Yes	No	P/F
Module	DROP MODULE CASCADE	No	Yes	P/F
	DROP MODULE RESTRICT	Yes	No	P/F
Procedure	DROP PROCEDURE CASCADE	No	Yes	P/F
	DROP PROCEDURE RESTRICT	Yes	No	P/F
Table	DROP TABLE CASCADE	No	Yes	P/F or LS
	DROP TABLE RESTRICT	Yes	No	P/F, LS, or DR
View	DROP VIEW CASCADE	No	Yes	P/F or LS
	DROP VIEW RESTRICT	Yes	No	P/F, LS, or DR

Key to dependency types:

DE—Default evaluating
DR—Default reserving
LS—Language semantics
P/F—Procedure or Function

Any DROP statement that is restricted does not affect validation because a statement with the RESTRICT keyword prevents the statement from deleting any objects that have stored routine dependencies. Drop cascade operations do execute successfully but cause invalidation.

Use the SHOW PROCEDURE or SHOW FUNCTION statement in interactive SQL to see whether a stored procedure or function is valid or invalid. In the following case, deleting a table on which the P1 stored procedure relied caused Oracle Rdb to invalidate P1:

```
SQL> SHOW PROCEDURE P1
Procedure name is: P1
Status is INVALID
        Can be revalidated
```

In the above example, Oracle Rdb indicates that the procedure can be revalidated. See Section 13.9.1 for information on revalidating routines.

Stored routines that do not display a "Status is INVALID" message are valid routines.

13.9 Revalidating Stored Routines

When a routine has been invalidated, you can attempt revalidation. However, the action to take depends on the dependency type of the invalidated stored routine, as follows:

- You can revalidate invalidated stored routines that have the following dependency types:
  - Procedure
  - Default evaluating
  - Default reserving

  Section 13.9.1 describes how to revalidate the routines.
- You cannot revalidate stored routines invalidated by a language semantic dependency. You can only re-create them by following the process described in Section 13.9.2.

13.9.1 Revalidating Invalidated Stored Routines

In most cases, when a stored routine with the dependency type of procedure, default evaluating, or default reserving is invalidated, you can make the routine valid again.

You can revalidate a stored routine, such as the NEW_SALARY_PROC stored procedure defined in Example 13–9, after invalidation because its underlying object, the SALARY_HISTORY table, creates a procedure dependency on the stored routine.

Example 13–9 Stored Module Definition with Procedure Dependency Type

```
CREATE MODULE NEW_SALARY_PROC LANGUAGE SQL
      PROCEDURE NEW_SALARY_PROC
             (:ID CHAR(5),
              :NEW_SALARY INTEGER(2));
      BEGIN
         UPDATE SALARY_HISTORY
              SET SALARY_END = CURRENT_TIMESTAMP
                  WHERE EMPLOYEE_ID = :ID;
```

(continued on next page)

Example 13–9 (Cont.) Stored Module Definition with Procedure Dependency Type

```
        INSERT INTO SALARY_HISTORY (EMPLOYEE_ID, SALARY_AMOUNT,
                                    SALARY_START, SALARY_END)
              VALUES
                    (:ID, :NEW_SALARY, CURRENT_TIMESTAMP, NULL);
    END;
END MODULE;
```

If you drop the SALARY_HISTORY table using the CASCADE option, Oracle Rdb marks the NEW_SALARY_PROC as invalid. Oracle Rdb does so because the stored procedure has a procedure dependency on the SALARY_HISTORY table, upon which the stored procedure relies for proper execution.

To make a stored routine valid again, take the following steps:

1. Re-create any database objects upon which the routine is dependent. For example, to make the NEW_SALARY_PROC procedure valid, re-create the SALARY_HISTORY table.
2. Define the logical name RDMS$VALIDATE_ROUTINE or the configuration parameter RDB_VALIDATE_ROUTINE to 1 to mark an invalid routine as valid. Oracle Rdb marks each invalid routine as valid when the process calls the stored routine within a read/write transaction.
3. Invoke interactive SQL and attach to the database.

   You can use precompiled SQL and SQL module language, but these interfaces do not support the SET NOEXECUTE statement.
4. Start a read/write transaction.
5. To avoid errors if transaction statements are used in the stored routine and to avoid inadvertently modifying data, use the following SQL statement:

   ```
   SQL> SET NOEXECUTE
   ```
6. Invoke the stored routine.

   For example, the following statement calls the stored procedure NEW_SALARY_PROC:

   ```
   SQL> CALL NEW_SALARY_PROC('00196', 2);
   ```

   Because the SET NOEXECUTE statement is used, the statements in the routine do not execute, and the data is not stored in the database.

If other stored routines were marked invalid and you have re-created the database element on which they depend, you can validate these stored routines by invoking each routine.

7. Issue the SET EXECUTE statement.
8. Issue the COMMIT statement.

13.9.2 Re-Creating Invalidated Stored Routines with Language Semantic Dependencies

Any stored routine that has been invalidated due to a language semantic dependency cannot be revalidated using the method described in Section 13.9.1. Instead, you must re-create its parent module and underlying schema objects by taking the following steps:

1. Use the RMU Extract command with the Item=Module and Output qualifiers to extract the stored module definition into a file.

   Modifying the extracted module definition is easier than reentering the definition line-by-line. Refer to the *Oracle RMU Reference Manual* for further information about the RMU Extract command.

2. Delete the stored module with the DROP MODULE statement.
3. Re-create the underlying schema object or objects that caused the language semantic dependency invalidation.
4. Create the stored module again with the CREATE MODULE statement.

   Modify the extracted file (created in step 1) and store the module definition in the database with the CREATE MODULE statement.

Oracle Rdb does not allow you to revalidate a language semantic invalidation because, even if you reproduce the schema objects that caused invalidation, Oracle Rdb cannot determine whether or not the re-created objects were created as the stored routine used them. For example, you might be able to re-create a table with the same column names, but what happens if you change the order of the columns? In this case, the table is re-created and Oracle Rdb could compile and execute the stored routine. However, the routine may not execute as expected. Because Oracle Rdb cannot guarantee that the table was re-created as originally defined, Oracle Rdb disallows revalidation of stored routines invalidated by a language semantic dependency.

14
Using External Routines

This chapter describes how to create and use external routines, which are external procedures or external functions written in a 3GL language, linked into a shareable image or shared object, and registered in a database. In the sections that follow, you will become familiar with:

- The concept of external routines
- Developing applications that use external routines
- Creating external routine definitions in the database
- Modifying and deleting external routine definitions from the database
- Writing external routines and writing programs to invoke the routines
- Creating OpenVMS shareable images for external routines
- Creating Digital UNIX shared objects for external routines
- Invoking external routines in SQL statements
- Specifying the execution characteristics of routines
- The concepts of routine activation and deactivation
- Declaring and passing parameters to and from external routines and declaring return values
- Coding external routines in particular host languages
- Using notify routines for initialization and cleanup operations
- Handling exceptions that external routines can encounter
- The limitations of external routines
- Troubleshooting problems with external routines
- Writing external routines that are portable and efficient

14.1 Introducing External Routines

External routines provide the procedural capabilities of programming languages. As a result, they extend the effects of SQL statements into the realm of tasks that can be performed through 3GL routines. External routines can perform tasks such as computing the square root of a value or attaching to a database and performing data manipulation operations on the database.

SQL implements the following types of external routines:

- External functions

  An **external function** is a 3GL program (written in a language such as C or COBOL) that you invoke by using the function name as a value expression in an SQL statement. For example, the following INSERT statement invokes the OpenVMS Runtime Library (RTL) routine MTH$SQRT to store square root values in a column of the SQUARE_ROOTS table:

  ```
  INSERT INTO SQUARE_ROOTS VALUES (P_NUM, SQRT(CAST(P_NUM AS REAL)));
  ```

  Like a built-in or stored function, an external function may accept a list of input arguments and always returns a single value that SQL uses in its evaluation of an SQL statement. Unlike a built-in or stored function, however, the code for external functions is stored in files *external* to SQL and Oracle Rdb.

  To register the name of the external function in the database, specify the function definition using the SQL CREATE FUNCTION statement.

- External procedures

  An **external procedure** is a 3GL program (written in a language such as C or COBOL) that you invoke using the SQL CALL statement. Like a stored procedure, an external procedure may accept a list of input, output, or input/output arguments, and does not return a value. Unlike a stored procedure, however, the code for external procedures is stored in files *external* to SQL and Oracle Rdb.

  The following example shows a CALL statement that invokes the external procedure ADD_SOUNDEX_NAME:

  ```
  CALL ADD_SOUNDEX_NAME (:error);
  ```

  To register the name of the external procedure in the database, specify the routine definition using the SQL CREATE PROCEDURE statement.

External routines can be based on existing routines from run-time libraries, operating system services, or other existing libraries, or you can write new external routines. For example, you might want to create an external function that lets you identify names in a database that sound alike; the English pronunciation of Barns, Barnes, and Barnse is the same. The following statement invokes the SOUNDEX user-defined external function that lets you encode a name into a string, which you can then use to search a database for sound-alike names:

```
SELECT LAST_NAME, SOUNDEX(LAST_NAME) FROM EMPLOYEES;
```

14.2 Developing External Routines

To develop an external routine and use it with an Oracle Rdb database, take the following steps:

1. If the external routine does not exist, create it using a 3GL language.
2. Compile the routine.

   If the routine does not contain any calls to an Oracle Rdb database, compile the source code with a host language compiler. If the routine contains calls to an Oracle Rdb database, compile the source code with the SQL precompiler. If the routine refers to SQL module procedures, which make calls to an Oracle Rdb database, compile the routine source code with the host language compiler and the SQL module procedures with the SQL module processor.
3. Create a shareable image or a shared object.

   OpenVMS VAX OpenVMS Alpha

   On OpenVMS, you must create a shareable image to use an external routine, unless an existing routine already resides in an existing shareable image. ♦

   Digital UNIX

   On Digital UNIX, you must create a shared object to use an external routine, unless an existing routine already resides in an existing shared object.♦
4. Test the routine

   After coding, compiling, and linking an external routine, test the routine independently to ensure that it does what you intend. This step usually requires that you write a test program.
5. Create the routine definition in the database

You create the external routine definition in an Oracle Rdb database using the CREATE FUNCTION or CREATE PROCEDURE statement. The external routine definition contains information about the routine, such as the parameters that the external routine uses, the external routine name, the location of the executable image or object containing the routine, and the language in which the routine is coded.

6. Invoke the routine in an SQL statement

   You can invoke an external function wherever a value expression is allowed in an SQL statement. You invoke an external procedure with an SQL CALL statement within a compound statement.

   Refer to Section 14.8 for examples of where you might commonly invoke external routines.

The active scope of an external routine is generally the scope of the active attach on the database that invoked the external routine. In some environments, external routines might be physically deactivated (code removed) when you detach from the invoking database.

Note that operations performed by external routines occur outside the context of any database transaction in effect when the routine is invoked. If an external routine performs an operation, such as writing to a file or spawning a process, but a rollback occurs in the database, the operation performed by the external routine will not be undone as part of the transaction roll back.

14.3 Creating External Routine Definitions

Before you can invoke an external routine from an SQL statement, you must register the external routine name, creating the external routine definition, in an Oracle Rdb database. Although this step is not logically the first step you take in creating external routines, it is useful to understand the type of information about external routines that you store in an Oracle Rdb database.

Section 14.3.1 describes how to create an external function definition. Section 14.3.2 describes how to create an external procedure definition. You use many of the same clauses to create external functions and external procedures.

14.3.1 Creating External Function Definitions

The *definition* of an external function resides in an Oracle Rdb database like other schema objects, such as tables or views. To create an external function definition, use the CREATE FUNCTION statement, as shown in Example 14–1.

Example 14–1 Defining an External Function with the CREATE FUNCTION Statement

```
-- SQRT external function.
CREATE FUNCTION SQRT ❶ (IN :PARAM1 REAL) ❷
    RETURNS REAL; ❸
    EXTERNAL NAME MTH$SQRT ❹
      LOCATION 'SYS$SHARE:MTHRTL.EXE' ❺
      LANGUAGE GENERAL ❻
      GENERAL PARAMETER STYLE ❼
    NOT VARIANT ❽
    COMMENT IS 'Square Root of an F-floating value'; ❾
```

The numbered callouts in Example 14–1 are keyed to the following list:

❶ CREATE FUNCTION SQRT

Creates an external function as an object in an Oracle Rdb database and specifies the name of the external function definition.

The name must be unique among all external routine definitions and stored routine names.

❷ IN :PARAM1 REAL

Specifies the parameter name and the SQL data type (REAL) for the IN parameter used by the external function. The parameter name is optional, but if you use it, you must precede it with a colon (:). You can use only IN parameters for external functions.

You can specify any SQL data type, except LIST OF BYTE VARYING. Alternatively, you can use a domain name to specify the data type.

Also, you can specify a passing mechanism for each parameter. When you do not specify a passing mechanism, SQL passes the parameter by REFERENCE. The passing mechanism you specify depends on the data type of the parameter and the language of the external function.

For more information about parameters and supported passing mechanisms, see Section 14.11 and Section 14.12.

❸ RETURNS REAL

Describes the SQL data type for the return value of the external function.

You can specify any SQL data type, except LIST OF BYTE VARYING. Alternatively, you can use a domain name to specify the data type.

You can specify a passing mechanism for the return value. If you do not specify a passing mechanism, SQL passes return values of numeric data type by VALUE and return values of character data type by REFERENCE. The passing mechanism you specify depends on the data type of the return value and the language of the external function. For information on the supported passing mechanisms, see Section 14.11 and Section 14.12.

❹ EXTERNAL NAME MTH$SQRT

Specifies the name of the external function as it is declared in the external routine program.

❺ LOCATION 'SYS$SHARE:MTHRTL.EXE'

Specifies the file specification of the external function.

OpenVMS

On OpenVMS, you can use a file specification or a logical name to identify the image location of the external function. ♦

Digital UNIX

On Digital UNIX, you can use an absolute or relative pathname to identify the shared object location of the external function. ♦

If you do not specify a location clause, or if you specify the DEFAULT LOCATION clause, SQL uses the name RDB$ROUTINES as the image location.

❻ LANGUAGE GENERAL

Identifies the host language in which the external function is written.

You can specify the ADA, C, COBOL, FORTRAN, PASCAL, or GENERAL keyword. The GENERAL keyword enables SQL to call an external function written in any language, if the external function uses the data types and parameter mechanisms supported by Oracle Rdb. Use this keyword if you do not know the language in which the function is written or if the function is written in a language that is not supported by SQL.

❼ GENERAL PARAMETER STYLE

Identifies the method used to pass arguments to and return values from external functions. SQL currently restricts this clause to style GENERAL, which most closely corresponds to the OpenVMS calling conventions. To comply with the evolving SQL standard, future versions of Oracle Rdb might implement other parameter styles.

❽ NOT VARIANT

Specifies whether or not to invoke a function each time it occurs within the scope of a single query. The NOT VARIANT clause can (but does not always) result in a single evaluation of corresponding function expressions in a single query. The resulting value is used in all occurrences of the corresponding function expression. The final decision to evaluate an external function only once depends on the Oracle Rdb query optimizer.

The VARIANT clause forces the evaluation of the function every time the function is invoked.

❾ COMMENT IS 'Square Root of an F-floating value'

Lets you add a descriptive comment about the external function.

For a complete description of the CREATE FUNCTION statement, its syntax and arguments, refer to the Create Routine Statement in the *Oracle Rdb7 SQL Reference Manual*.

14.3.2 Creating External Procedure Definitions

The *definition* of an external procedure resides in an Oracle Rdb database like other schema objects, such as tables or views. To create an external procedure definition, use the CREATE PROCEDURE statement, as shown in Example 14–2.

Example 14–2 Defining an External Procedure with the CREATE PROCEDURE Statement

```
CREATE PROCEDURE ADD_SOUNDEX_NAME ❶
       (INOUT :PARAM1 INTEGER BY REFERENCE) ❷;
       EXTERNAL NAME ADD_SOUNDEX_NAME  ❸
       LOCATION 'ADD_SOUNDEX.EXE' ❹
       LANGUAGE FORTRAN   ❺
       GENERAL PARAMETER STYLE   ❻
       BIND ON SERVER SITE       ❼
       BIND SCOPE TRANSACTION    ❽
       NOTIFY ADD_SOUNDEX_NOTIFY ON BIND, TRANSACTION;  ❾
```

The numbered callouts in Example 14–2 are keyed to the following list:

❶ CREATE PROCEDURE ADD_SOUNDEX_NAME

Creates an external procedure as an object in an Oracle Rdb database and specifies the name of the external procedure definition.

The name must be unique among all external routine definitions and stored routine names.

❷ INOUT :PARAM1 INTEGER BY REFERENCE

Specifies the parameter name, the SQL data type (INTEGER) for the INOUT parameter used by the external procedure, and the passing mechanism. The parameter name is optional, but if you use it, you must precede it with a colon (:). You can use IN, INOUT, and OUT parameters for external procedures.

You can specify any SQL data type, except LIST OF BYTE VARYING. Alternatively, you can use a domain name to specify the data type.

You can also specify a passing mechanism for each parameter. When you do not specify a passing mechanism, SQL passes the parameter by REFERENCE. The passing mechanism you specify depends on the data type of the parameter and the language of the external procedure.

For more information about parameters and supported passing mechanisms, see Section 14.11 and Section 14.12. .

❸ EXTERNAL NAME ADD_SOUNDEX_NAME

Specifies the name of the external procedure as it is declared in the external routine program.

❹ LOCATION 'ADD_SOUNDEX.EXE'

OpenVMS

On OpenVMS, you can use a file specification or a logical name to identify the image location of the external procedure. ♦

Digital UNIX

On Digital UNIX, you can use an absolute or relative pathname to identify the shared object location of the external procedure. ♦

If you do not specify a location clause, or if you specify the DEFAULT LOCATION clause, SQL uses the name RDB$ROUTINES as the image location.

❺ LANGUAGE GENERAL

Identifies the host language in which the external procedure is written.

You can specify the ADA, C, COBOL, FORTRAN, PASCAL, or GENERAL keyword. The GENERAL keyword enables SQL to call an external procedure written in any language, if the external procedure uses the data types and parameter mechanisms supported by Oracle Rdb. Use this keyword if you do not know the language in which the function is written or if the function is written in a language that is not supported by SQL.

❻ GENERAL PARAMETER STYLE

Identifies the method used to pass arguments to and return values from an external procedure. SQL currently restricts this clause to style GENERAL, which most closely corresponds to the OpenVMS calling conventions. To comply with the evolving SQL standard, future versions of Oracle Rdb might implement other parameter styles.

❼ BIND ON SERVER SITE

You can specify SERVER SITE or CLIENT SITE binding.

CLIENT SITE binding is available only on OpenVMS. When you specify this binding, Oracle Rdb activates and executes the external routine in the same process as the Oracle Rdb server.

When you specify SERVER SITE binding, Oracle Rdb activates the external routine in a separate executor process on the same processing node as the Oracle Rdb server.

For more information about client-site and server-site binding, see Section 14.10.

❽ BIND SCOPE TRANSACTION

Specifies the scope during which an external routine is active and when the routine is deactivated. The TRANSACTION keyword indicates that the external routine is deactivated when a transaction is terminated. Alternatively, you can specify the CONNECT keyword, which indicates that the external routine is deactivated when the database that contains the routine definition is disconnected. The default is CONNECT.

❾ NOTIFY ADD_SOUNDEX_NOTIFY ON BIND, TRANSACTION

Specifies the name of the entry point for a notify routine in the external routine image or shared object and the events for which the notify routine is invoked. The notify routine can perform initialization and cleanup operations, such as initializing variables or attaching or detaching from a database. It can share information about database-related events with the body of the external routine.

The NOTIFY clause specifies the types of events that invoke the notify routine. You can specify BIND, CONNECT, or TRANSACTION as the events.

For more information about using the NOTIFY clause, see Section 14.13.

For a complete description of the CREATE PROCEDURE statement, its syntax and arguments, refer to the Create Routine Statement in the *Oracle Rdb7 SQL Reference Manual*.

14.4 Modifying and Deleting External Routine Definitions

To modify an external routine definition, you must first delete it and then create it again with the changes applied. SQL does not provide an ALTER FUNCTION or ALTER PROCEDURE statement to modify external routine definitions.

To remove an external function definition from an Oracle Rdb database, use the DROP FUNCTION statement, as shown in the following example:

```
DROP FUNCTION SOUNDEX;
```

To remove an external procedure definition from an Oracle Rdb database, use the DROP PROCEDURE statement, as shown in the following example:

```
DROP PROCEDURE CLEAR_SOUNDEX;
```

When you use the RESTRICT keyword (the default) Oracle Rdb prevents the removal of an external routine definition when any other object within an Oracle Rdb database refers to that routine.

SQL returns an exception and prevents the removal of an external routine definition if it is referenced in:

- COMPUTED BY clause of a table definition
- Constraint definitions
- Trigger definitions
- Stored procedures
- Active requests

When you use the CASCADE keyword to drop the external routine, Oracle Rdb executes the drop operation and invalidates any stored routines that refer to the dropped routine. See the *Oracle Rdb7 SQL Reference Manual* for more information about the effects of the CASCADE keyword with the DROP FUNCTION and DROP PROCEDURE statement.

14.5 Creating External Routines

This section describes how to create the following types of external routines:

- An external routine based on existing routines

  See Section 14.5.1.
- An external routine that you code yourself (user-defined)

  See Section 14.5.2.

- An external routine that makes calls into the database

  See Section 14.5.3.

- An external routine that requires a jacket routine

  See Section 14.5.4.

14.5.1 Creating External Routines Based on Existing Routines

This section shows you how to use an existing external function, which calculates the square root of a series of numbers.

Because the function already exists and the executable image is a shareable image or shared object, you only need to create the function definition in SQL and create an SQL program or interactive SQL script to invoke the function.

OpenVMS VAX

Example 14–3 shows how to create the function definition, which uses the OpenVMS VAX run-time library routine MTH$SQRT to calculate the square root of a series of numbers.

Example 14–3 Defining an External Function for an Existing Function on OpenVMS VAX

```
CREATE FUNCTION SQRT (IN :PARAM1 REAL)
       RETURNS REAL;
       EXTERNAL NAME MTH$SQRT LOCATION 'SYS$SHARE:MTHRTL.EXE'
       LANGUAGE GENERAL
       GENERAL PARAMETER STYLE;                    ♦
```

OpenVMS Alpha

Example 14–4 shows how to create the function definition, which uses the OpenVMS Alpha run-time library routine MTH$SQRT to calculate the square root of a series of numbers.

Example 14–4 Defining an External Function for an Existing Function on OpenVMS Alpha

```
CREATE FUNCTION SQRT (IN :PARAM1 REAL)
       RETURNS REAL;
       EXTERNAL NAME MTH$SQRT LOCATION 'SYS$SHARE:DPML$SHR.EXE'
       LANGUAGE GENERAL
       GENERAL PARAMETER STYLE;
```

The only difference from the OpenVMS VAX example is the location of the routine. ♦

Digital UNIX

On Digital UNIX, you need jacket routines to convert between F-float and IEEE S-float data. The sqrt.c jacket routine, shown in the following example, calls utility routines in ftof.c and uses the system-provided sqrtf routine:

```
#include <math.h>

extern void cvt_ff_to_fs( unsigned int *, float * );
extern void cvt_fs_to_ff( float *, unsigned int * );

extern void SQRT( unsigned int *ffout, unsigned int *ffin ) {
float fsin,fsout;
cvt_ff_to_fs( ffin, &fsin );
fsout = sqrtf( fsin );
cvt_fs_to_ff( &fsout, ffout ); }
```

The following example shows the utility routine ftof.c, which calls system-provided floating-point conversion (cvt) routines:

```
#include <cvt.h>
#include <excpt.h>
static int sts;

extern void cvt_ff_to_fs( unsigned int *ff, float *fs ) {
  sts = cvt_ftof( ff,CVT_VAX_F, fs,CVT_IEEE_S, CVT_FORCE_ALL_SPECIAL_VALUES);
  if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
extern void cvt_fg_to_ft( unsigned long *fg, double *ft ) {
  sts = cvt_ftof( fg,CVT_VAX_G, ft,CVT_IEEE_T, CVT_FORCE_ALL_SPECIAL_VALUES);
  if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
extern void cvt_fs_to_ff( float *fs, unsigned int *ff ) {
  sts = cvt_ftof( fs,CVT_IEEE_S, ff,CVT_VAX_F, CVT_FORCE_ALL_SPECIAL_VALUES);
  if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
extern void cvt_ft_to_fg( double *ft, unsigned long *fg ) {
  sts = cvt_ftof( ft,CVT_IEEE_T, fg,CVT_VAX_G, CVT_FORCE_ALL_SPECIAL_VALUES);
  if (sts != CVT_NORMAL) exc_raise_status_exception( sts ); }
```

To compile and link the jacket and utility routines to create a shared object, use the following commands:

```
$ cc -c -o sqrt.o sqrt.c
$ cc -c -o ftof.o ftof.c
$ ld -shared -soname $HOME/sqrt.so -o $HOME/sqrt.so sqrt.o ftof.o -lexc -lm -lc
```

Example 14–5 shows how to create the function definition in SQL.

Example 14–5 Defining an External Function for an Existing Function on Digital UNIX

```
CREATE FUNCTION SQRT (IN :PARAM1 REAL)
       RETURNS REAL BY REFERENCE;
       EXTERNAL NAME SQRT LOCATION 'sqrt.so'
       LANGUAGE C
       GENERAL PARAMETER STYLE
       BIND ON SERVER SITE;                          ♦
```

Example 14–6 shows an excerpt of the SQL module file, sql_sqrt_c.sqlmod, that contains a procedure that invokes the external function.

Example 14–6 Invoking a Predefined External Function Using an SQL Module

```
   .
   .
   .
-------------------------------------------------------------------------
-- Procedure to invoke the SQRT external function and insert values
-- into the SQUARE_ROOTS table
-------------------------------------------------------------------------
PROCEDURE INSERT_SQUARE_ROOTS_TABLE
    (SQLCODE,
     :P_NUM  INTEGER);

    INSERT INTO SQUARE_ROOTS VALUES (:P_NUM, SQRT(CAST(:P_NUM AS REAL)));
   .
   .
   .
```

Example 14–7 shows the C program, sql_sqrt_mod.c, that calls the procedure in sql_sqrt_c.sqlmod.

Example 14–7 Invoking a Predefined Function with a C Program

```
   .
   .
   .
/* Declarations of entry points in the SQL module. */
extern void INSERT_SQUARE_ROOTS_TABLE(int *sqlcode, int *p_num);
```

(continued on next page)

Example 14–7 (Cont.) Invoking a Predefined Function with a C Program

```
main()
{
    int sqlcode = 0;
    int i, n;
    float s;

   .
   .
   .
/* Call the SQL module procedure to insert the square root values into the
   database.  */

    for(i=0;i<10;i++)
        INSERT_SQUARE_ROOTS_TABLE(&sqlcode, &i);

    OPEN_E1(&sqlcode);
    do  {
        FETCH_E1(&sqlcode, &n, &s);
        if (sqlcode==0)
            printf("number=%d square_root=%f\n", n, s);
        } while (sqlcode==0);
   .
   .
   .
}
```

14.5.2 Writing User-Defined External Routines

This section uses a set of examples to show you how to write a user-defined external function, which is an external function based on code that you write yourself.

Table 14–1 describes the components that you need to invoke the user-defined SOUNDEX function.

Table 14–1 Components for Building a User-Defined External Function

Component	Description
soundex_c.c	The C program containing the SOUNDEX function
soundex_c.opt	The options file that defines any universal symbols or symbol vectors (OpenVMS)
sql_soundex_c.sqlmod	The SQL module file containing procedures to call user-defined SOUNDEX function
sql_soundex_mod.c	The main C program that calls the associated SQL module file

Example 14–8 shows an excerpt of the user-defined external function that generates a list of four-character codes for the last names of employees in the mf_personnel database.

Example 14–8 Writing a User-Defined External Function in C

```
/* This function is an implementation of the SOUNDEX routine,
   originally developed by Margaret K. Odell and Robert C. Russell
   and described in Knuth's "Sorting and Searching, Vol. 3, The
   Art of Computer Programming."  */

#include <stdio.h>
   .
   .
   .
void SOUNDEX (char *out_string, char *source_str)
   .
   .
   .
```

The following example shows the contents of the soundex_c.opt options file as declared for an OpenVMS VAX system.

```
UNIVERSAL = SOUNDEX
```

The name you use in the options file must match the name given to the external function when you define the function with the CREATE FUNCTION statement.

You must link the object file of the program and the options file to create a shareable image. Example 14–9 shows how to compile the program that contains the SOUNDEX function and and how to link the program and the options file to create a shareable image.

Example 14–9 Compiling and Linking a User-Defined External Function

```
$ CC/G_FLOAT SOUNDEX_C.C
$ LINK/SHAREABLE=SOUNDEX_C_IMAGE.EXE SOUNDEX_C,-
_$     SOUNDEX_C.OPT/OPT
```

Example 14–10 shows how to create the SOUNDEX function definition in SQL.

Example 14–10 Defining an External Function for a User-Defined Function

```
CREATE FUNCTION SOUNDEX (IN CHAR(32)) RETURNS CHAR(4);
     EXTERNAL LOCATION 'SOUNDEX_C_IMAGE.EXE'
     LANGUAGE C
     GENERAL PARAMETER STYLE;
```

Example 14–11 shows an SQL module file that contains procedures that invoke the SOUNDEX function.

Example 14–11 Invoking a User-Defined External Function from an SQL Module

```
   .
   .
   .
------------------------------------------------------------
-- Declare Statement Section
------------------------------------------------------------
DECLARE E1_CURSOR CURSOR FOR SELECT LAST_NAME, SOUNDEX_LAST_NAME
        FROM EMPLOYEES
DECLARE E2_CURSOR CURSOR FOR SELECT LAST_NAME, SOUNDEX_LAST_NAME
        FROM EMPLOYEES WHERE SOUNDEX_LAST_NAME=SOUNDEX('Varmelker')
   .
   .
   .
------------------------------------------------------------
-- Procedure to fetch a row.
------------------------------------------------------------
PROCEDURE FETCH_E2
    (SQLCODE,
     :P_LAST_NAME CHAR(32),
     :P_SOUNDEX_LAST_NAME CHAR(4));

    FETCH E2_CURSOR INTO  :P_LAST_NAME, :P_SOUNDEX_LAST_NAME;
------------------------------------------------------------
-- Procedure to alter table.
------------------------------------------------------------
PROCEDURE ALTER_EMPLOYEES_TABLE
    (SQLCODE);

    ALTER TABLE EMPLOYEES ADD SOUNDEX_LAST_NAME CHAR(4);
```

(continued on next page)

Example 14–11 (Cont.) Invoking a User-Defined External Function from an SQL Module

```
------------------------------------------------------------
-- Procedure to update values in employees table.
------------------------------------------------------------
PROCEDURE UPDATE_EMPLOYEES
    (SQLCODE);

    UPDATE EMPLOYEES SET SOUNDEX_LAST_NAME = SOUNDEX(LAST_NAME);
```

Example 14–12 shows a C program that calls the SQL module procedures that invoke the SOUNDEX function.

Example 14–12 Invoking a User-Defined External Function with a C Program

```
#include <stdio.h>
typedef char STR33[33];
typedef char STR5[5];
typedef int *sqlcode;

/* Declarations of entry points in the SQL module. */
   .
   .
   .

extern void OPEN_E1(int *sqlcode);
extern void CLOSE_E1(int *sqlcode);
extern void FETCH_E1(int *sqlcode,  STR33 last_name,  STR5 soundex_last_name);
extern void ALTER_EMPLOYEES_TABLE(int *sqlcode);
extern void UPDATE_EMPLOYEES(int *sqlcode);

main()
{
    char last_name[33];
    char slast_name[5];
    int sqlcode = 0;
   .
   .
   .
    printf("\nAltering EMPLOYEES table to add SOUNDEX_LAST_NAME column\n");
    ALTER_EMPLOYEES_TABLE(&sqlcode);

    printf("\nUpdating EMPLOYEES table, setting SOUNDEX_LAST_NAME equal to\n");
    printf("SOUNDEX(LAST_NAME)\n");
    UPDATE_EMPLOYEES(&sqlcode);

    printf("\nPrint LAST_NAME and SOUNDEX_LAST_NAME\n");
    printf("LAST_NAME                    SOUNDEX_LAST_NAME\n");
```

(continued on next page)

Example 14–12 (Cont.) Invoking a User-Defined External Function with a C Program

```
    OPEN_E1(&sqlcode);
    do  {
        FETCH_E1(&sqlcode, last_name, slast_name);
        if (sqlcode==0)
            printf("%s %s\n",
                    last_name, slast_name);
        } while (sqlcode==0);

    if (sqlcode!=100)
        printf("SQL error code = %d\n", sqlcode);
    CLOSE_E1(&sqlcode);

   printf("\nPrint LAST_NAME and SOUNDEX_LAST_NAME where SOUNDEX_LAST_NAME\n");
    printf(" sounds like 'Varmelker')\n");
    printf("LAST_NAME                    SOUNDEX_LAST_NAME\n");
    OPEN_E2(&sqlcode);

    do  {
        FETCH_E2(&sqlcode, last_name, slast_name);
        if (sqlcode==0)
            printf("%s %s\n",
                    last_name, slast_name);
        } while (sqlcode==0);

    if (sqlcode!=100)
        printf("SQL error code = %d\n", sqlcode);
    CLOSE_E2(&sqlcode);
   .
   .
   .
}
```

The following example shows how to compile and link the SQL module and the host language program that invoke the SOUNDEX function:

```
$ CC/G_FLOAT SQL_SOUNDEX_MOD
$ SQL$MOD SQL_SOUNDEX_C
$ LINK SQL_SOUNDEX_MOD, SQL_SOUNDEX_C
$ RUN SQL_SOUNDEX_MOD
```

When the SQL_SOUNDEX_MOD program executes, it displays the following results:

```
Altering EMPLOYEES table to add SOUNDEX_LAST_NAME column

Updating EMPLOYEES table, setting SOUNDEX_LAST_NAME equal to
SOUNDEX(LAST_NAME)
```

```
Print LAST_NAME and SOUNDEX_LAST_NAME
LAST_NAME                    SOUNDEX_LAST_NAME
Smith                           S530
O'Sullivan                      O024
Lasch                           L200
   .
   .
   .
Keisling                        K245
Vormelker                       V654
   .
   .
   .

Print LAST_NAME and SOUNDEX_LAST_NAME where SOUNDEX_LAST_NAME
sounds like 'Varmelker'
LAST_NAME                    SOUNDEX_LAST_NAME
Vormelker                       V654
```

Section 14.6 and Section 14.7 describe how to compile and link external routines on all supported platforms.

14.5.3 Writing External Routines That Call into the Database

You can write an external routine that contains calls into an Oracle Rdb database. That is, an external routine can contain SQL data manipulation statements in precompiled SQL programs or SQL module procedures and can use those statements to insert, update, and delete data in a database.

This section illustrates using an external routine that adds a document to a database, and in doing so, indexes the document. The external routine is created using an SQL module, sql_add_doc_mod.sqlmod, and a C language program, sql_add_doc.c. The external routine uses a database called docdb that contains two tables, DOC and DOC_INDEX. The following example shows the definition of the tables:

```
SQL> -- Create the table DOC to hold the document.
SQL> CREATE TABLE DOC (DOC_ID INTEGER, DOC_TEXT VARCHAR (500));
SQL>
SQL> -- Create the table DOC_INDEX to hold the index and information
SQL> -- about the occurrence of the index tokens.
SQL> CREATE TABLE DOC_INDEX
cont>       (DOC_ID INTEGER, TOKEN CHAR (32),
cont>        TOKEN_OCC_USED INTEGER, TOKEN_OCC VARCHAR (100),
cont>        PRIMARY KEY (TOKEN, DOC_ID) NOT DEFERRABLE);
```

Example 14–13 shows the C language program sql_add_doc.c.

Example 14–13 Writing an External Routine to Call into a Database

```
/* External routine ADD_DOC

   Given the text of a document, create tokens by breaking the document into
   words, create an occurrence record for each unique token, and store a
   uniformly space-delimited form of the document.

   This external routine uses the SQL module, sql_add_doc_mod.sqlmod, for
   database operations, including a notify routine to control the attach
   and detach on routine binding and unbinding.   */

#include <string.h>
#include <stdio.h>

extern int ADD_DOC( int *, char * );
int add_token( int *, char *, int );
extern void ADD_DOC_NOTIFY( int *, int *, int *, int * );

void ATTACH_DB( int * );
void DETACH_DB( int * );
void START_TRAN( int * );
void COMMIT_TRAN( int * );
void ROLLBACK_TRAN( int * );
void STORE_DOC( int *, char *, int * );

void GET_DOC_INDEX_BY_TOKEN_ID( char *, int *, int *, char *, int * );
void STORE_DOC_INDEX( int *, char *, int *, char *, int * );
void UPDATE_DOC_INDEX( int *, char *, int *, char *, int * );
void sql_signal( void );
extern int ADD_DOC( int *doc_id, char *doc_text ) {

    int tokoff = 0, toklen = 0, docoff = 0, ndocoff = 0, dummy = 0, status = 0;
    char new_text[500+1], token[32+1];

/* Start the transaction. */
    START_TRAN( &status );
    if (status !=0) return status;
    for (docoff =0; docoff<= strlen( doc_text ); docoff++) {

        /* At token break, capture token from new text and add to index. */
        if (doc_text[docoff] == ' ' || doc_text[docoff] == '    ' ||
            doc_text[docoff] == '\0') {
            if (toklen > 0) {
                strncpy( token, &new_text[tokoff], toklen );
                token[toklen] = '\0';
                status = add_token( doc_id, token, tokoff );
                if (status != 0) break;
                toklen = 0; tokoff = 0;
            }
```

(continued on next page)

Example 14–13 (Cont.) Writing an External Routine to Call into a Database

```
        } else {
            /* Move document text to uniformly delimited new doc text. */
            if (ndocoff > 0 && toklen == 0)
                { new_text[ndocoff++] = ' '; tokoff = ndocoff; }
            new_text[ndocoff++] = doc_text[docoff]; toklen++;
        }
    }
    if (status !=0) { ROLLBACK_TRAN( &dummy ); return status; }

    /* Save uniformly delimited doc text.  Store it in the database. */
    new_text[ndocoff] = '\0';
    STORE_DOC ( doc_id, new_text, &status );
    if (status !=0) { ROLLBACK_TRAN( &dummy ); return status; }

    /* Commit the transation. */
    COMMIT_TRAN( &status );
    if (status !=0) { ROLLBACK_TRAN( &dummy ); return status; }

    return status;
}

int add_token( int *doc_id, char *token, int tokoff ) {

    int    used, status = 0;
    char   occ[100+1];

    /* Insert or update token occurrence index. */
    GET_DOC_INDEX_BY_TOKEN_ID( token, doc_id, &used, occ, &status );
    if (status == 100) { status = 0; used = 1; }
    else               { if (status == 0) used++; else return status; }
    sprintf( &occ[(used-1)*4], "%04d", tokoff );
    if (used == 1) STORE_DOC_INDEX(  doc_id, token, &used, occ, &status );
    else           UPDATE_DOC_INDEX( doc_id, token, &used, occ, &status );
    return status;
}

extern void ADD_DOC_NOTIFY( int *func, int *u1, int *u2, int *u3 ) {
    int status;

    /* Attach to database on routine activation.
       Detach on routine deactivation. */
    switch (*func) {
        case 1: ATTACH_DB( &status ); if (status != 0) sql_signal(); break;
        case 2: DETACH_DB( &status ); if (status != 0) sql_signal(); break;
    }
    return;
}
```

Example 14–13 calls the SQL module procedures shown in Example 14–14. The procedures attach to the database, control transactions, insert documents into the database, and index the documents.

Example 14–14 Calling into a Database from an SQL Module

```
-- This SQL module, linked together with the C language program sql_add_doc.c
-- forms an external function, ADD_DOC.  The SQL module procedures perform
-- database operations, including attaching to the database,
-- controlling transactions, inserting, retrieving, and updating data.
--
-- This SQL module is also used by the C program, sql_locate_doc.c to retrieve
-- information from the docdb database.

MODULE ADDDOC
LANGUAGE C
PARAMETER COLONS

DECLARE ALIAS FOR FILENAME docdb
DECLARE FIND_DOC_INDEX_BY_TOKEN CURSOR FOR
    SELECT DOC_ID FROM DOC_INDEX WHERE TOKEN = :tok

PROCEDURE ATTACH_DB
    (SQLCODE);
    ATTACH 'FILENAME docdb';

PROCEDURE DETACH_DB
    (SQLCODE);
    DISCONNECT DEFAULT;

PROCEDURE START_READ_TRAN
    (SQLCODE);
    SET TRANSACTION READ ONLY;

PROCEDURE START_TRAN
    (SQLCODE);
    SET TRANSACTION READ WRITE;

PROCEDURE COMMIT_TRAN
    (SQLCODE);
    COMMIT;

PROCEDURE ROLLBACK_TRAN
    (SQLCODE);
    ROLLBACK;

-- Insert a document into the database.
PROCEDURE STORE_DOC
    (:ID INTEGER, :TEXT CHAR(500), SQLCODE);
    INSERT INTO DOC VALUES (:ID, :TEXT);
```

(continued on next page)

Example 14–14 (Cont.) Calling into a Database from an SQL Module

```
-- Retrieve the index to determine if the index token already exists
-- or if it is a new token.
PROCEDURE GET_DOC_INDEX_BY_TOKEN
    (:TOK CHAR (32), :ID INTEGER, :USED INTEGER, :OCC CHAR (100), SQLCODE);
    SELECT DOC_ID, TOKEN_OCC_USED, TOKEN_OCC INTO :ID, :USED, :OCC
        FROM DOC_INDEX WHERE :TOK = TOKEN;

PROCEDURE GET_DOC_INDEX_BY_TOKEN_ID
    (:TOK CHAR (32), :ID INTEGER, :USED INTEGER, :OCC CHAR (100), SQLCODE);
    SELECT TOKEN_OCC_USED, TOKEN_OCC INTO :USED, :OCC
        FROM DOC_INDEX WHERE :TOK = TOKEN AND :ID = DOC_ID;

-- Insert new index tokens.
PROCEDURE STORE_DOC_INDEX
    (:ID INTEGER, :TOK CHAR (32), :USED INTEGER, :OCC CHAR (100), SQLCODE);
    INSERT INTO DOC_INDEX VALUES (:ID, :TOK, :USED, :OCC);

-- Update the occurrence count if the index token already exists.
PROCEDURE UPDATE_DOC_INDEX
    (:ID INTEGER, :TOK CHAR (32), :USED INTEGER, :OCC CHAR (100), SQLCODE);
    UPDATE DOC_INDEX SET TOKEN_OCC_USED = :USED, TOKEN_OCC = :OCC
        WHERE :TOK = TOKEN AND :ID = DOC_ID;
```

Example 14–15 shows the external routine definition as it is stored in the database docdb.

Example 14–15 Defining an External Routine That Calls into the Database

```
SET QUOTING RULES 'SQL92';
--  Define the external routine that loads the index.
CREATE FUNCTION ADD_DOC
            (IN INTEGER, IN VARCHAR (500))
            RETURNS INTEGER;
            EXTERNAL NAME "ADD_DOC" LOCATION 'ADDDOC.EXE'
            LANGUAGE C GENERAL PARAMETER STYLE
            BIND ON SERVER SITE
            NOTIFY "ADD_DOC_NOTIFY" ON BIND;
```

For information about the limitations SQL imposes when you call into a database, see Section 14.15.

To compile and link the ADD_DOC external function, take the following steps. (This sequence shows how to compile and link on OpenVMS VAX. Section 14.6 and Section 14.7 explain how to compile and link on all supported platforms.)

1. Compile the SQL module using the SQL module processor:

```
$ SQLMOD
INPUT FILE> SQL_ADD_DOC_MOD
```

2. Compile the C host language program:

```
$ CC SQL_ADD_DOC
```

3. Create a linker options file, sql_add_doc.opt, which contains the following lines:

```
UNIVERSAL = ADD_DOC
UNIVERSAL = ADD_DOC_NOTIFY
PSECT_ATTR=RDB$MESSAGE_VECTOR,NOSHR
PSECT_ATTR=RDB$DBHANDLE,NOSHR
PSECT_ATTR=RDB$TRANSACTION_HANDLE,NOSHR
```

4. Create the shareable image:

```
$ LINK/SHARE=SYS$LOGIN:ADDDOC.EXE SQL_ADD_DOC.OBJ, SQL_ADD_DOC_MOD.OBJ, -
$_ SQL_ADD_DOC.OPT/OPT, -
$_ SQL$USER/LIB
```

The following example shows how to use the ADD_DOC external function in interactive SQL to load text from documents into the database:

```
SQL> ATTACH 'FILENAME docdb';
SQL>
SQL> -- Declare a variable to retrieve the status of the statement.
SQL> DECLARE :STATUS INTEGER;
SQL> -- Populate the database with documents and index them.
SQL>  BEGIN
cont>  SET :STATUS =
cont>      ADD_DOC (1, 'An external function is a 3GL program that you invoke
by using the function name as a value expression in an SQL statement.');
cont> END;
SQL> PRINT 'The SQL error code is ', :status;
                              STATUS
 The SQL error code is             0
SQL>  BEGIN
cont>   SET :STATUS =
cont>      ADD_DOC (2, 'An external procedure is a 3GL program that you invoke
using the SQL CALL statement.');
cont> END;
SQL> PRINT 'The SQL error code is ', :status;
                              STATUS
 The SQL error code is             0
SQL>
```

```
SQL>  BEGIN
cont>   SET :STATUS =
cont>    ADD_DOC (3, 'An external routine is an external procedure or external
function, linked into a shareable image or shared module, and registered in the
database.');
cont> END;
SQL> PRINT 'The SQL error code is ', :status;
                               STATUS
 The SQL error code is              0
```

The following example shows an excerpt from a C language program that calls procedures in an SQL module to search for strings in the database:

```
   .
   .
   .
int main (void) {
    int seaidx, toklen=0, status = 0, exitval = 0, doc_id;
    char search_text[101];

    /* Get the search string. */
    printf( "Search string: " );
    scanf( "%[^\n]", search_text );
    printf( "\n" );
    for (seaidx=0; seaidx<=strlen( search_text ); seaidx++) {
        if (search_text[seaidx] == ' ' || search_text[seaidx] == '      ' ||
            search_text[seaidx] == '\0') {
            if (toklen > 0) { token[tokcnt][toklen] = '\0';
                              toksiz[tokcnt++] = toklen; toklen = 0; }
        } else { token[tokcnt][toklen++] = search_text[seaidx]; }
    }
    /* Search for relevant documents. */
    if (tokcnt > 0) {
        ATTACH_DB( &status );
        if (status != 0) { exitval = 1; goto done; }
        START_READ_TRAN( &status );
        if (status != 0) { exitval = 1; goto det_db; }
        OPEN_FIND_DOC_INDEX_BY_TOKEN( (char *)&token[0][0], &status );
        if (status == 100) goto rbk_db;
        if (status != 0) { exitval = 1; goto rbk_db; }
        while (1) {
            FETCH_ID_FROM_DOC_INDEX( &doc_id, &status );
            if (status == 100) break;
            if (status != 0) { exitval = 1; goto rbk_db; }
            status = check_doc_occ( 0, doc_id, 0 );
            if (status != 0) { exitval = 1; goto rbk_db; }
        }
   .
   .
   .
int check_doc_occ( int tokidx, int doc_id, int curoff ) {
```

```
    int idx, fndlen, nxtidx, nxtoff = 0, occoff = 0, status = 0, used = 0;
    char occ[100+1], doc[500+1];

    /* Match next token in document (with the id) using prior end offset. */
    GET_DOC_INDEX_BY_TOKEN_ID( (char *)&token[tokidx][0], &doc_id, &used, occ,
                               &status );
    if (status == 100 && tokidx != 0) return 0;
    if (status != 0) return status;
    for (idx=0; idx<used; idx++) {
        sscanf( &occ[idx*4], "%04d", &occoff );
        if (tokidx == 0) begoff = occoff;
        if (tokidx == 0 || curoff == occoff) {
            if (tokidx < tokcnt-1) {
                nxtoff = occoff + toksiz[tokidx] + 1; nxtidx = tokidx + 1;
                status = check_doc_occ( nxtidx, doc_id, nxtoff );
            } else {
                GET_DOC_BY_ID( &doc_id, doc, & status );
                if (status != 0) return status;
                fndlen = occoff + toksiz[tokidx] - begoff;
                printf( "Found doc %d, offset %d: %*.*s\n", doc_id, begoff,
                        fndlen, fndlen, &doc[begoff] );
            }
        }
        if (tokidx != 0 && occoff >= curoff) break;
```

The following example shows SQL module procedures, called from the previous C program, which search for strings in the database:

```
PROCEDURE GET_DOC_BY_ID
     (:ID INTEGER, :TEXT CHAR(500), SQLCODE);
     SELECT DOC_TEXT INTO :TEXT
         FROM DOC WHERE DOC_ID = :ID;

PROCEDURE GET_DOC_INDEX_BY_TOKEN_ID
    (:TOK CHAR (32), :ID INTEGER, :USED INTEGER, :OCC CHAR (100), SQLCODE);
    SELECT TOKEN_OCC_USED, TOKEN_OCC INTO :USED, :OCC
        FROM DOC_INDEX WHERE :TOK = TOKEN AND :ID = DOC_ID;

PROCEDURE OPEN_FIND_DOC_INDEX_BY_TOKEN
    (:TOK CHAR (32), SQLCODE);
    OPEN FIND_DOC_INDEX_BY_TOKEN;

PROCEDURE FETCH_ID_FROM_DOC_INDEX
    (:ID INTEGER, SQLCODE);
    FETCH FIND_DOC_INDEX_BY_TOKEN INTO :ID;

PROCEDURE CLOSE_FIND_DOC_INDEX_BY_TOKEN
     (SQLCODE);
    CLOSE FIND_DOC_INDEX_BY_TOKEN;
```

14.5.4 Writing Jacket Routines to Invoke External Routines

Some external routines require additional processing to return the information that you want. An application program that performs such additional processing on a subprogram (in this case, an external function) is called a **jacket routine**. When you execute a jacket routine, in effect you execute the external function indirectly, within the enveloping program. For example, OpenVMS system services routines and some OpenVMS RTL routines must be called from a jacket routine.

Some external routines, such as a random number routine, cannot calculate random numbers unless a seed (or starting value) is passed to them by an enveloping jacket routine. For example, the MYRANDOM jacket routine passes a seed value to the OpenVMS RTL MTH$RANDOM routine, which then calculates a series of random numbers that are inserted into the database.

The additional processing that a jacket routine performs can vary widely from very simple to extremely complex. The MYRANDOM jacket routine shown in this section represents the simplest case.

Table 14–2 describes the components that you need to invoke an OpenVMS RTL routine through a jacket routine.

Table 14–2 Components for Building a Jacket Routine to Invoke an External Function

Component	Description
myrandom_c.c	The C program containing the jacket routine MYRANDOM that calls the OpenVMS RTL MTH$RANDOM routine
myrandom_c.opt	The options file that defines any universal symbols
sql_myrandom_c.sqlmod	The SQL module file containing procedures to call the OpenVMS RTL MTH$RANDOM routine through a jacket routine
sql_myrandom_mod.c	The C program that calls the associated SQL module file

Example 14–16 shows the jacket routine, myrandom_c.c, that passes a seed value to the MTH$RANDOM routine.

Example 14–16 Writing a Jacket Routine in C

```
static seed = 0;
float MYRANDOM()
{
    return (MTH$RANDOM( &seed));
}
```

The following example shows the contents of the myrandom_c.opt options file as declared for an OpenVMS VAX system.

```
UNIVERSAL = MYRANDOM
```

The name you use in the options file must match the name given to the external function when you define the function with the CREATE FUNCTION statement. Example 14–17 shows how to compile and link the jacket routine with the MTH$RANDOM routine and the options file to create a shareable image.

Example 14–17 Compiling and Linking a Jacket Routine

```
$ CC/G_FLOAT myrandom_c.c
$ LINK/SHAREABLE=myrandom_c_image.exe myrandom_c,-
_$  myrandom_c.opt/OPT
```

Example 14–18 shows how to create the function definition in SQL.

Example 14–18 Defining an External Function That Calls a Jacket Routine

```
CREATE FUNCTION MYRANDOM() RETURNS REAL;
    EXTERNAL LOCATION 'DB_DISK:[DB_FUNCT]MYRANDOM_C_IMAGE.EXE'
    LANGUAGE C
    GENERAL PARAMETER STYLE;
```

Example 14–19 shows an SQL module file, sql_myrandom_c.sqlmod, that defines a set of procedures for invoking the MTH$RANDOM routine.

Example 14–19 Invoking a Jacket Routine from an SQL Module

```
   .
   .
   .
-------------------------------------------------------------
-- Procedure to insert random numbers
-------------------------------------------------------------
PROCEDURE INSERT_RANDOM_NUMBER
    (SQLCODE);

    INSERT INTO RANDOM_NUMBERS VALUES(MYRANDOM());
```

Example 14–20 shows the C program, sql_myrandom_mod.c, that calls the SQL module procedure.

Example 14–20 Invoking a Jacket Routine from a C Program

```
#include <stdio.h>
typedef int *SQLCODE;

/* Declarations of entry points in the SQL module. */
extern void INSERT_RANDOM_NUMBER(int *sqlcode);
   .
   .
   .

main()
{
    int sqlcode = 0;
    int i;
    float r;

/* Invoke the external function. */
    for(i=0;i<10;i++)
        INSERT_RANDOM_NUMBER(&sqlcode);
    OPEN_E1(&sqlcode);
    do{
        FETCH_E1(&sqlcode, &r);
        if (sqlcode==0)
            printf("Random number = %f\n", r);
      } while (sqlcode==0);
   .
   .
   .
}
```

See Section 14.5.1 for another example of using a jacket routine with external routines.

14.6 Creating Shareable Images for External Routines

OpenVMS VAX OpenVMS Alpha

On OpenVMS, you must create a shareable image to use an external routine, unless an existing routine already resides in a shareable image.

You must ensure that the location of the routine as specified in the database is the same as the actual location of the shareable image. If you use a logical name in the CREATE FUNCTION or CREATE PROCEDURE statement, check to see that the logical is pointing to the correct image.

When you use server-site binding, any logical name defined for the database must be relevant in the executor process where it will be used to access the database.

Before you link the images to create a shareable image, compile the program that contains the external routine.

If the routine does not contain any calls into the database, compile the host language program. For example, the following command line shows how to compile an external routine written in the C language:

```
$ CC soundex_c.c
```

If the routine uses embedded SQL to make calls to an Oracle Rdb database, compile the source code with the SQL precompiler. For example, the following command compiles a C language program, sql_test.sc, that contains embedded SQL statements:

```
$ SQL$PRE
SQLPRE> sql_test/CC/SQLOPTIONS=ROLLBACK_ON_EXIT
```

If the routine refers to SQL module procedures, which make calls to an Oracle Rdb database, compile the routine source code with the host language compiler and the SQL module procedures with the SQL module processor. For example, the following command lines show how to compile an external routine using the SQL module processor and the C language compiler:

```
$ SQLMOD
INPUT FILE> sql_add_doc_mod
$
$ CC sql_add_doc
```

The following sections describe how to create shareable images on OpenVMS VAX and OpenVMS Alpha systems. ♦

14.6.1 Creating Shareable Images for External Routines on OpenVMS VAX

OpenVMS VAX

On OpenVMS VAX, to link a shareable image containing an external routine that does not refer to the database, take the following steps:

1. Create an options file that specifies the external routine name as a universal symbol by using the following format for each external routine:

   ```
   UNIVERSAL = routine-name
   ```

   When you build a shareable image that contains global variables and you do not specify any options, all program sections (PSECTS) corresponding to global variables are created as shareable. As a result, you must install the image. To avoid installing the image, define the global variables as nonshareable (NOSHR).

   For an external routine called SOUNDEX, the options file, called SOUNDEX_C.OPT, contains the following line:

   ```
   UNIVERSAL = SOUNDEX
   ```

2. Link the object file of the program and the options file to create a shareable image:

   ```
   $ LINK/SHAREABLE=soundex_c_image.exe soundex_c,-
   _$     soundex_c.opt/OPT
   ```

To link a shareable image containing an external routine that calls into the database, take the following steps:

1. Create an options file that specifies the external routine name as a universal symbol by using the following format for each external routine;

   ```
   UNIVERSAL = routine-name
   ```

   If the external routine contains notify routines, add a universal symbol for each notify routine contained in the external routine:

   ```
   UNIVERSAL = notify-routine-name
   ```

   When you build a shareable image that contains global variables and you do not specify any options, all program sections (PSECTS) corresponding to global variables are created as shareable. As a result, you must install the image. To avoid installing the image, define the global variables as nonshareable (NOSHR).

   Because the ADDDOC external routine contains a notify routine called ADD_DOC_NOTIFY, the options file, sql_add_doc.opt, contains a universal symbol for it as well as for the routine. The options file contains the following lines:

```
UNIVERSAL = ADD_DOC
UNIVERSAL = ADD_DOC_NOTIFY
PSECT_ATTR=RDB$MESSAGE_VECTOR,NOSHR
PSECT_ATTR=RDB$DBHANDLE,NOSHR
PSECT_ATTR=RDB$TRANSACTION_HANDLE,NOSHR
```

2. Link the object files of the program and the options file to create a shareable image:

```
$ LINK/SHARE=sys$login:adddoc.exe sql_add_doc.obj, sql_add_doc_mod.obj, -
$_ sql_add_doc.opt/OPT, -
$_ SQL$USER/LIB                                          ♦
```

14.6.2 Creating Shareable Images for External Routines on OpenVMS Alpha

OpenVMS Alpha

On OpenVMS Alpha, to link a shareable image containing an external routine that does not refer to the database, take the following steps:

1. Create an options file that specifies the external routine name as a symbol vector by using the following format for each external routine:

   ```
   SYMBOL_VECTOR = (routine-name = PROCEDURE )
   ```

   When you build a shareable image that contains global variables and you do not specify any options, all program sections (PSECTS) corresponding to global variables are created as shareable. As a result, you must install the image. To avoid installing the image, define the global variables as nonshareable (NOSHR).

   For an external routine called SOUNDEX, the options file, called SOUNDEX_C.OPT, contains the following line:

   ```
   SYMBOL_VECTOR = (SOUNDEX = PROCEDURE)
   ```

2. Link the object file of the program and the options file to create a shareable image:

   ```
   $ LINK/SHAREABLE=soundex_c_image.exe soundex_c,-
   _$     soundex_c.opt/OPT
   ```

To link a shareable image containing an external routine that calls into the database, take the following steps:

1. Create an options file that specifies the external routine name as a symbol vector by using the following format for each external routine:

   ```
   SYMBOL_VECTOR = (routine-name = PROCEDURE)
   ```

   If the external routine contains notify routines, add a symbol vector for each notify routine contained in the external routine:

   ```
   SYMBOL_VECTOR = (notify-routine-name = PROCEDURE)
   ```

When you build a shareable image that contains global variables and you do not specify any options, all program sections (PSECTS) corresponding to global variables are created as shareable. As a result, you must install the image. To avoid installing the image, define the global variables as nonshareable (NOSHR).

Because the ADDDOC external routine contains a notify routine called ADD_DOC_NOTIFY, the options file, sql_add_doc.opt, contains a symbol vector for it as well as for the routine. The options file contains the following lines:

```
SYMBOL_VECTOR = (ADD_DOC = PROCEDURE)
SYMBOL_VECTOR = (ADD_DOC_NOTIFY = PROCEDURE)
PSECT_ATTR=RDB$MESSAGE_VECTOR,NOSHR
PSECT_ATTR=RDB$DBHANDLE,NOSHR
PSECT_ATTR=RDB$TRANSACTION_HANDLE,NOSHR
```

2. Link the object file of the program and the options file to create a shareable image:

```
$ LINK/SHARE=sys$login:adddoc.exe sql_add_doc.obj, sql_add_doc_mod.obj, -
$_ sql_add_doc.opt/OPT, -
$_ SQL$USER/LIB                                                    ♦
```

14.7 Creating Shared Objects on Digital UNIX

Digital UNIX

On Digital UNIX, you must build a shared object to use an external routine, unless the existing routine already resides in a shared object.

If the routine does not contain any calls into the database, take the following steps to build a shared object:

1. Before you link the images to create a shared object, compile the program that contains the external routine. For example, the following command line shows how to compile an external routine written in the C language:

```
$ cc -c -o soundex_c.o soundex_c.c
```

2. Link the modules, using the `-shared` option on the command line. The following example shows how to create the shared object for the SOUNDEX external routine:

```
$ ld -shared -soname ${HOME}/soundex_c_image.exe -o \
>     ${HOME}/soundex_c_image.exe soundex_c.o \
>     -no_archive -nocount -lots -lc
```

To create a shared object that contains an external routine that calls into the database, take the following steps:

1. If the routine uses embedded SQL to make calls to an Oracle Rdb database, compile the source code with the SQL precompiler. For example, the following command shows how to use the SQL precompiler to compile a C language program, sql_test.sc, that contains embedded SQL statements:

   ```
   $ sqlpre -l cc sql_test.sc
   ```

   If the routine refers to SQL module procedures, which make calls to an Oracle Rdb database, compile the routine source code with the host language compiler and the SQL module procedures with the SQL module processor. For example, the following command lines show how to compile the ADD_DOC external routine using the SQL module processor and the C language compiler:

   ```
   $ sqlmod -int32 sql_add_doc_mod.sqlmod
   $
   $ cc -c -o sql_add_doc.o sql_add_doc.c
   ```

2. Link the modules, using the `-shared` option on the command line. The following example shows how to link to create the shared object for the ADDDOC external routine:

   ```
   $ ld -hidden_symbol 'RDB$TRANSACTON_HANDLE' -hidden_symbol 'RDB$DBHANDLE' \
   >    -hidden_symbol 'RDB$MESSAGE_VECTOR' -shared \
   >    -soname ${HOME}/adddoc.exe -o ${HOME}/adddoc.exe \
   >    sql_add_doc.o sql_add_doc_mod.o -no_archive -nocount \
   >    -lsql -lrdbshr -lcosi -lots -lc                          ♦
   ```

14.8 Invoking External Routines

To invoke an external routine, you must have explicit execute access to the external routine. System privileges, such as the OpenVMS SYSPRV or the Digital UNIX superuser accounts, do not serve as overrides.

14.8.1 Invoking External Functions

You can invoke an external function from anywhere a value expression can be specified in an SQL statement.

The SQL statements in Example 14–21 show how to invoke an external function called SOUNDEX from a variety of value expression locations. The SOUNDEX external function definition is shown in the following example:

```
CREATE FUNCTION SOUNDEX(IN CHAR(32))
       RETURNS CHAR(4);
       EXTERNAL LOCATION 'SOUNDEX$EXE' LANGUAGE C
       GENERAL PARAMETER STYLE;
```

Example 14–21 Invoking External Functions in SQL Statements

```
-- Invoke the external function in the select list of a SELECT statement
-- to display an employee's last name and the last name generated by the
-- SOUNDEX external function.
SELECT  LAST_NAME, SOUNDEX(LAST_NAME)
        FROM EMPLOYEES;

-- Invoke the external function in a WHERE clause to find employees
-- whose name sounds like "Barns".
SELECT LAST_NAME
       FROM EMPLOYEES
       WHERE SOUNDEX(LAST_NAME) = SOUNDEX('Barns');

-- Invoke the external function to insert column values into rows of
-- a table.
INSERT  INTO  EMPLOYEES
        (LAST_NAME, SOUNDEX_LAST_NAME)
        VALUES
        ('Barnes', SOUNDEX('Barnes'));

-- Invoke the external function twice in a column constraint definition.
-- You can also use an external function in a table constraint definition.
CREATE TABLE EMPLOYEES
       (LAST_NAME CHAR(32),
        FIRST_NAME CHAR(32),
        CHECK(SOUNDEX(LAST_NAME) <> SOUNDEX(FIRST_NAME)
        );

-- Invoke an external function in a COMPUTED BY clause.
CREATE TABLE EMPLOYEES_CACHE
       (LAST_NAME,
        SOUNDEX_LAST_NAME COMPUTED BY SOUNDEX(LAST_NAME)));
```

The parameters you pass to the external function must agree with the formal parameters in number, data type, and length.

14.8.2 Invoking an External Function Within a Trigger

You can invoke an external function from within a trigger, as a predicate in a triggered action, or as an action in a triggered statement (INSERT, DELETE, and UPDATE).

Example 14–22 invokes the SOUNDEX external function in a triggered INSERT statement.

Example 14–22 Invoking External Functions Within a Trigger Definition

```
CREATE TRIGGER LOG_UNUSUAL_NAME_CHANGES
       AFTER UPDATE OF LAST_NAME
       ON EMPLOYEES
       REFERENCING OLD AS OEMP
                   NEW AS NEMP
       WHEN (SOUNDEX(OEMP.LAST_NAME) = SOUNDEX(NEMP.LAST_NAME))
       (INSERT INTO UNUSUAL_NAME_CHANGES_LIST
            (OEMP.LAST_NAME,
             NEMP.LAST_NAME,
             SOUNDEX(OEMP.LAST_NAME))
       FOR EACH ROW;
```

Suppose that you want to receive a mail message whenever an employee record is removed from your personnel database. The mail message might contain both the type of operation (DELETE) performed on the employee's record and the employee's identification number. The following set of OpenVMS examples shows one way to use an external function within a trigger definition to send a mail message. Refer to the *Oracle Rdb7 Guide to Database Design and Definition* for more information about using external functions in trigger definitions.

Example 14–23 shows the external function definition.

Example 14–23 Tracking Database Activity with External Functions

```
CREATE FUNCTION SEND_MAIL
               (IN CHAR(32),
                IN CHAR(32),
                IN CHAR(256))
       RETURNS INTEGER;
       EXTERNAL NAME   SEND_MAIL
       LOCATION 'SUPPORT_FUNCTIONS'
       LANGUAGE C
       GENERAL PARAMETER STYLE;
```

The first argument passes the user name of the person to whom the mail is sent, the second argument passes the mail subject text, and the third argument passes the message content.

Example 14–24 shows the trigger definition.

Example 14–24 Using Triggers and External Functions to Track Database Activity

```
CREATE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
       BEFORE DELETE ON EMPLOYEES
       WHEN SEND_MAIL ('godfrind',
           'Employee '|| EMPLOYEES.EMPLOYEE_ID || ' deleted',
            'User ' || CURRENT_USER || ' just deleted employee ' ||
             EMPLOYEES.EMPLOYEE_ID ) <> 0
       ( ERROR )
       FOR EACH ROW;
```

The SEND_MAIL external function uses the return value as a status to indicate success or failure of the mail operation. Before an employee record is deleted, the SEND_MAIL external function is called. The trigger passes information about the database activity to the external function.

14.8.3 Invoking External Procedures

You invoke an external procedure using the CALL statement, as shown in Example 14–25.

Example 14–25 Invoking External Procedures

```
BEGIN
  SET :ERROR = 0;
  CALL ADD_SOUNDEX_NAME (:ERROR);
END;
```

The parameters you pass to the external procedure must agree with the formal parameters in number, data type, and length.

14.9 Specifying Execution Characteristics of Routines

All external routines are controlled by an executor manager. The executor manager, as well as all external routines, operates in a non-privileged execution mode, such as OpenVMS user mode.

When you create an external routine definition, you can control many of the execution characteristics of the routine by specifying the BIND ON CLIENT SITE or BIND SERVER SITE clauses.

OpenVMS VAX OpenVMS Alpha

You can specify CLIENT SITE binding only on OpenVMS. When you specify this binding, Oracle Rdb activates and executes the external routine in the same process as the server. This binding offers the most efficient execution characteristics. When the server resides in the same process with the client application, this binding allows sharing of client resources and debugging of external routines as if they are part of the client application. However, it has address space limitations and is restricted to the SYSTEM_USER execution environment.

In addition, Oracle Rdb returns an error if an application process has higher privileges than the user running the application. This situation usually occurs when one of the application images is installed with privileges that the user has not been granted. Oracle Rdb generates an exception to prevent the security problem of a random external routine using elevated privileges to which it would not normally have access. ♦

You can specify SERVER SITE binding on any platform. When you specify this binding, Oracle Rdb activates and executes the the external routine in a separate executor process on the same processing node as the Oracle Rdb server. SERVER SITE binding offers reasonable execution characteristics, a larger address space than client-site binding, a true SESSION_USER execution environment, and has no restrictions regarding elevated privileges for the client process. However, this binding does not permit sharing of client resources and it can be difficult to debug routines.

When you specify SERVER SITE binding, a server-site executor process is initialized on behalf of the SESSION_USER who invokes the external routine. That is, the executor process uses the user name and user identification, the default login device and directory, and on OpenVMS, the privileges and quotas of the session user.

The process environment is similar to that established for a login with no login command script available. In other words, the environment is similar to logging on an OpenVMS system using the DCL LOGIN/NOCOMMAND command or logging on a Digital UNIX system with no .login, .profile, or resource configuration (.*rc) files. The process is non-interactive (that is, detached on OpenVMS), has no command line interpreter or command shell, has all standard I/O devices connected to the null device, and uses system defaults for file creation protection and other masks.

As stated earlier, when you specify server-site binding, the external routine uses the session user's environment. The session user is the user specified by the authorization identifier of the current SQL session. Consider the following scenario to understand when and how the session user is used:

- Your user name is USERA. When you log on to an OpenVMS system, you establish the process system user as USERA.
- You attach to database D1 and do not specify the USER clause. As a result, when you query database D1, the session user is USERA, the same as the SYSTEM_USER. External routines at either the client site or server site operate in the system environment of USERA.
- You attach to database D2 specifying the USER 'USERX' clause. As a result, when you query database D2, the session user is USERX, but the SYSTEM_USER is USERA. External routines at the server site operate in the system environment of USERX because USERX was the declared session user when the server site executor process was created. External routines at the client site operate in the system environment of USERA, because the OpenVMS process system user is USERA.

If the server-site binding executor manager encounters an unexpected error, the current routine execution is aborted and an exception is returned. All subsequent attempts to use that executor process generate the same exception, until all databases for the session user are disconnected. At that point, the executor process is discarded and the next execution of a routine with server-site binding initiates a new executor process.

Digital UNIX

On Digital UNIX, the server-site process is initialized so that any trappable signal that normally stops or terminates the process is converted to an exception when a signal is raised. ♦

The database server and the server-site binding executor manager manipulate routine parameter data and control information by using shared memory. Shared memory has a default size of 31 pages. You can select a different shared memory size by setting the RDMS$RTX_SHRMEM_PAGE_CNT logical name or the RDB_RTX_SHRMEM_PAGE_CNT configuration parameter to a different value.

14.10 Understanding Routine Activation and Deactivation

When you create the external routine definition, ensure that the proper external routine image is activated. Do that by using one of the following methods:

- Specify an image file specification; do not use the default image name.
- Use a full file specification; do not use any of the default components in the image file specification. On OpenVMS, specify a device and specific directory or a logical name that translates to both. On Digital UNIX, specify an absolute path.

OpenVMS VAX OpenVMS Alpha

- On OpenVMS, follow these rules:
  - Do not use logical names in the file specification. Specify the physical device names.
  - Use only logical names defined as /SYSTEM/EXECUTIVE_MODE /CONCEALED.
  - Specify SYSTEM LOGICAL_NAME TRANSLATION in the routine definition and ensure that all logical names are defined as /SYSTEM /EXECUTIVE_MODE and have only one translation.

    If you use the WITH SYSTEM LOGICAL_NAME TRANSLATION clause, the logical name is expanded. The resultant expanded file specification must have no more than 255 characters.
  - Make sure any logicals are defined on all nodes where the image is used.

  Use user mode logical names only while testing. User mode logical names allow the flexibility of using different routine images by redefining the logical name and, as such, are appropriate only in a non-production environment. ♦

Oracle Rdb recommends that, whenever possible, you place routine images to be used in a production environment in a protected library.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, place images in a directory referenced by the system-level, executive-mode logical name RDB$LIBRARY and specify SYSTEM LOGICAL_NAME TRANSLATION in the routine definition.

Example 14–26 adds a secure external function definition to your Oracle Rdb database.

Example 14–26 Securing an External Function Definition

```
CREATE FUNCTION SECURE_FUNC (IN CHAR(32) BY REFERENCE)
       RETURNS TINYINT;
       EXTERNAL LOCATION 'RDB$LIBRARY:SECURE_FUNC.EXE'
       WITH SYSTEM LOGICAL_NAME TRANSLATION
       LANGUAGE C
       GENERAL PARAMETER STYLE;
```

On OpenVMS, generally it is not necessary for an external routine image to be installed, particularly for server-site binding. When you use client-site binding and the application image is a privileged image, the routine image must be installed and the location file specification must contain only system-level, executive-mode logical names. ♦

Digital UNIX

On Digital UNIX, place shared objects in the following directory:

```
/usr/local/dbs/shlib
```
♦

Oracle Rdb dynamically activates the shareable image containing the external routine code at the time of the initial routine invocation. While the image containing the routine is active, the dynamic activation overhead is incurred only once for a routine.

When you physically replace an external routine image, any associated *active* external routine is not invalidated. Invocations of routines defined in these already active routine images continue to execute code from the old image. Only those invocations that cause physical activation after you replace the code use the new image and routine code.

Deactivation of a routine from the executor process occurs logically and, depending on the platform, sometimes physically. Deactivation occurs at the end of the bind scope and only if the external routine is in the activated state.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, physical deactivation (removal from the address space) can occur only for routines defined with the SERVER SITE binding. The deactivation occurs in bulk when all databases are detached and the SERVER SITE executor process is destroyed. ♦

If an external routine is still attached to one or more databases during the routine's deactivation, Oracle Rdb rolls back all active transactions and disconnects all databases. Oracle Rdb ignores any exceptions that occur during these cleanup operations.

14.11 Declaring and Passing Parameters and Return Values

How you declare parameters and function return values, and the mechanism you use to pass them, depends on the language of the external routine. Section 14.12 discusses the language-specific guidelines. However, the following points pertain to all languages:

- You cannot pass INOUT or OUT parameters by VALUE.
- If a function definition declares the function value passing mechanism to be by REFERENCE, LENGTH, or, on OpenVMS, DESCRIPTOR, the corresponding 3GL routine code must represent a procedure that accesses the output function value parameter as the first argument. Some languages, such as FORTRAN, provide function-specific syntax that handles this situation automatically.

- If character string data items are longer than the corresponding parameter, SQL truncates the data on input and blank-fills it on output. If character string data items are shorter than the corresponding parameter, SQL blank-fills or null-terminates the data on input and truncates it on output.
- External functions support only OpenVMS F-floating point and G-floating point data types. Routines that require IEEE floating point data types (S-floating and T-floating) must convert the input and output floating point data before and after use. As a result, on Digital UNIX you must pass the "foreign format floating" data as nonfloating point parameters.
- When the values associated with input or output parameters or function values are uninitialized, the results are indeterminate.
- If a routine invocation generates an exception, the output values for any OUT and INOUT access mode parameters (including a function value) is indeterminate.
- Output data (including the function value data) cannot represent an address.

In selecting and using a passing mechanism, keep in mind the following points:

- BY REFERENCE
  - The input and output data is passed as an address argument referencing the actual data. The address argument is 32 bits on OpenVMS and 64 bits on Digital UNIX.
  - If you specify the routine host language as C, SQL passes the character data types CHAR(n), NCHAR(n), VARCHAR(n), NCHAR VARYING(n), and LONG VARCHAR as null-terminated character strings, and the VARCHAR actual strings do not include the 16-bit length field.
- BY VALUE

  The input data and function value data is passed as a value. On OpenVMS, the valid input data types are SMALLINT, TINYINT, INTEGER, and REAL (32 bits). On Digital UNIX, and as a function value, valid data types are SMALLINT, TINYINT, INTEGER, and REAL (32 bits), BIGINT, DOUBLE PRECISION, DATE VMS, DATE ANSI, TIME, TIMESTAMP, and INTERVAL (64 bits).

OpenVMS VAX OpenVMS Alpha

- BY DESCRIPTOR
  - The input and output character data are each passed as an address argument referencing an OpenVMS fixed-length (class S) string (type T or VT) descriptor, which in turn references the actual character data. The descriptor is read-only and both addresses are 32 bits.
  - For fixed-length character string (type T) descriptors, the descriptor length value is the minimum of the length of the actual argument and the length of the declared parameter. For varying character string (type VT) descriptors, the descriptor length value is the length of the declared parameter. ♦
- BY LENGTH

OpenVMS VAX OpenVMS Alpha

  - On OpenVMS, BY LENGTH is the same as mechanism BY DESCRIPTOR. ♦

Digital UNIX

  - On Digital UNIX, input and output data are each passed as an address argument referencing, either directly or indirectly, the actual character data.

    For Digital UNIX Pascal, the parameter argument is an address referencing a private descriptor which consists of an address that references the character string and a length. For Digital UNIX FORTRAN, the parameter argument is an address referencing the character string; an extra hidden argument is passed representing the string length. All addresses are 64 bits.
  - On Digital UNIX, for fixed-length character strings, any passed length value is the minimum of the length of the actual argument and the length of the declared parameter. For all other character data types, the passed length value is the length of the declared parameter. ♦

14.12 Language-Specific Guidelines for Coding External Routines

Oracle Rdb supports external routines written in any programming language; however, if the external routine is written in a language other than Ada, C, COBOL, FORTRAN, or Pascal, you must specify the GENERAL keyword in the LANGUAGE clause of the CREATE FUNCTION or CREATE PROCEDURE statement. Although you can call external functions written in any language when you specify GENERAL, the external functions can only use the data types and parameter mechanisms that Oracle Rdb supports.

For information about the supported data types and parameter mechanisms, see the following sections.

14.12.1 Using External Routines with Ada

Keep in mind the following guidelines when you use external routines with the Ada language:

- SQL data types that have direct Ada equivalents are: TINYINT, SMALLINT, INTEGER, REAL, DOUBLE PRECISION, CHAR(n), and VARCHAR(n).
- The passing mechanism for parameters depends on the data type of the parameter:
  - Pass numeric arguments by REFERENCE.
  - On OpenVMS, pass fixed-length character strings by DESCRIPTOR or LENGTH.
  - On Digital UNIX, pass fixed-length character strings by REFERENCE.
  - Pass variable-length character strings by REFERENCE.
- The passing mechanism of the return value of external functions depends on the data type of the return value:
  - Numeric data types are returned by VALUE.
  - On OpenVMS, fixed-length character strings are returned by DESCRIPTOR or LENGTH.
  - On Digital UNIX, fixed-length character strings of eight or fewer characters are returned by VALUE; more than eight characters by REFERENCE.
  - Variable-length character strings are returned by REFERENCE.
- If you are using DEC Ada V3.0 or earlier, the parameter mechanism is determined by the compiler. You can find out the parameter mechanism chosen by the compiler by using the option 'warnings=compilation_ notes' during compilation. The parameter mechanism specification in the CREATE Routine statement should match the mechanism chosen by the Ada compiler.
- In addition to the mechanism choices listed, you can use the MECHANISM pragma to specify the passing mechanism for exported procedures in the Ada compiler.
- In addition to Ada functions, Ada procedures whose first argument has an OUT mode can be treated as Ada functions. These procedures must be exported as valued procedures.

14.12.2 Using External Routines with C

Keep in mind the following guidelines when you use external routines with the C language:

- SQL data types that have direct C equivalents are: TINYINT, SMALLINT, INTEGER, BIGINT (OpenVMS Alpha and Digital UNIX only), REAL, DOUBLE PRECISION, CHAR(n), and VARCHAR(n).
- The passing mechanism of parameters depends on the data type of the parameter:
  - Pass numeric data types by VALUE or REFERENCE.
  - Pass character string data types by REFERENCE.
- The passing mechanism of the return value of external functions depends on the data type of the return value:
  - Numeric data types are returned by VALUE or REFERENCE.
  - Character string data types are returned by REFERENCE.
- When you pass CHAR(n) and VARCHAR(n) strings by REFERENCE, SQL converts them to null-terminated text for input parameters and converts them from null-terminated text to CHAR(n) or VARCHAR(n) for output parameters.
- The following SQL statement creates an external function definition that points to an external function written in the C language:

```
CREATE FUNCTION CONCAT_STRING (IN CHAR(20), IN CHAR(20)) RETURNS CHAR(40);
       EXTERNAL LOCATION 'concat_location.exe' LANGUAGE C
       GENERAL PARAMETER STYLE;
```

  The C function is coded with the return value as the first argument:

```
void concat_string (char *result_str, char *first_input, char *second_input);
```

OpenVMS VAX

  Some compilers, such as VAX C (V3.2-044), do not place the return value as the first parameter, as it can return the address into R0. You must rewrite the C routine in the following example to correct this condition. For example:

```
char *concat_string (char *first_input, char *second_input);
```

♦

- You can pass fixed-length and variable-length character strings by DESCRIPTOR (on OpenVMS) or LENGTH, but the resulting data type does not represent a C language data type.

14.12.3 Using External Routines with COBOL

Keep in mind the following guidelines when you use external routines with the COBOL language:

- SQL data types that have direct COBOL equivalents are: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, CHAR(n), and VARCHAR(n).
- COBOL numeric equivalents are COMP (TINYINT, SMALLINT, INTEGER, BIGINT), and COMP-1 (REAL). COMP-2 (D-Float double precision) is not supported.
- You must pass all parameters, regardless of data type, by REFERENCE.
- The passing mechanism of the return value of external functions depends on the data type of the return value:
  - Numeric data types are returned by VALUE.
  - Character string data types are returned by REFERENCE.

14.12.4 Using External Routines with FORTRAN

Keep in mind the following guidelines when you use external routines with the FORTRAN language:

- SQL data types that have direct FORTRAN equivalents are: TINYINT, SMALLINT, INTEGER, BIGINT (OpenVMS Alpha and Digital UNIX only), REAL, DOUBLE PRECISION, CHAR(n), and VARCHAR(n).
- The passing mechanism of parameters depends on the data type of the parameter:
  - Pass numeric data types by REFERENCE.
  - Pass character strings by LENGTH or, on OpenVMS, DESCRIPTOR.
- The passing mechanism of the return value of external functions depends on the data type of the return value:
  - Numeric data types are returned by VALUE.
  - Character string data types are returned by LENGTH or, on OpenVMS, DESCRIPTOR.
- You can pass fixed-length and variable-length character strings by REFERENCE, but the resulting data type does not represent a FORTRAN language data type.

14.12.5 Using External Routines with Pascal

Keep in mind the following guidelines when you use external routines with the Pascal language:

- SQL data types that have direct Pascal equivalents are: SMALLINT, INTEGER, REAL, DOUBLE PRECISION, CHAR(n), and VARCHAR(n).
- You can pass all parameters, regardless of data type, by REFERENCE.
- To pass a character string by REFERENCE, declare it as:

  ```
  VAR name : PACKED ARRAY [1..n] OF CHAR
  ```

- You can pass fixed-length character strings by LENGTH, or on OpenVMS, by DESCRIPTOR. To do so, you must declare the resulting argument as shown in the following example:

  ```
  [CLASS_S] PACKED ARRAY [LOW..HIGH :INTEGER] OF CHAR
  ```

- The passing mechanism of the return value of external functions depends on the data type of the return value:
  - Numeric data types are returned by VALUE.
  - Character string data types are returned by REFERENCE.

The following example shows an external function definition called CONCAT_STRING:

```
CREATE FUNCTION CONCAT_STRING (IN CHAR(20) BY REFERENCE,
                               IN CHAR(20) BY REFERENCE)
       RETURNS CHAR(40);
       EXTERNAL LOCATION 'CONCAT_LOCATION' LANGUAGE Pascal
       GENERAL PARAMETER STYLE;
```

Specify an external function in Pascal as follows:

```
function concat_string (var input1_str : packed array[1..20] of char;
                        var input2_str : packed array[1..20] of char)
        : packed array [1..40];
```

14.13 Using Notify Routines

You can use a notify routine for initialization and cleanup operations, such as initializing variables or attaching or detaching from a database. You can also use it to share information about database-related events with the body of the external routine. For example, you can extend the caller's transaction scope to the functions of the body of the external routine by managing a journal of changes made to non-database files by the body of the external routine.

The notify routine must be contained in the shared image or object that contains the external routine.

When you use the NOTIFY clause, you can specify the following events:

- BIND—The notify routine is called when the external routine is activated and deactivated.
- CONNECT—The notify routine is called when a connection is made to the database that contains the external routine and when that database is disconnected.
- TRANSACTION—The notify routine is called when a transaction starts and when the transaction terminates.

When you use the NOTIFY clause, Oracle Rdb calls the notification entry point at the start of a selected event (queued until the next call to execute the body routine), and at the end of the event (immediately prior to event completion, assuming that notification of the start event has occurred.)

You pass a notify routine four 32-bit arguments by reference. The first argument indicates the type of notification, as shown in the following table, and the last three arguments are reserved for future use.

Value	Meaning	Constant
1	Routine activation	RDB$K_RTX_NOTIFY_ACTV_START
2	Routine deactivation	RDB$K_RTX_NOTIFY_ACTV_END
3	Database attachment	RDB$K_RTX_NOTIFY_CONN_START
4	Database disconnect	RDB$K_RTX_NOTIFY_CONN_END
5	Transaction start	RDB$K_RTX_NOTIFY_TRAN_START
6	Transaction commit	RDB$K_RTX_NOTIFY_TRAN_COMMIT
7	Transaction rollback	RDB$K_RTX_NOTIFY_TRAN_ROLLBACK

The notify events have a specific scope hierarchy. The BIND events (activation and deactivation) are the topmost scope, and TRANSACTION events the bottom-most scope. If you select all three events for notification and you invoke the external routine at least once while a transaction is active, the notify routine is called at least six times for the events of: routine bind, database connect, transaction start, transaction end, database disconnect, and routine unbind.

The bind scope you specify in the BIND SCOPE clause can affect which notify events are received. For example, if the bind scope is TRANSACTION, you cannot receive a notification for a database disconnect.

Note that the notification events and use of the notify routine is not linked in any way to the database recovery process.

14.14 Handling Exceptions in External Routines

Exceptions produced by an external routine, whether by the routine identified by the external routine name, or by the notify routine, are returned to the routine caller.

If an exception occurs in a notify routine that is called for an end of scope event, the exception causes the associated SQL statement to fail. However, the notify event is considered complete and trying the SQL statement again does not produce the notify routine exception a second time.

Oracle Rdb can detect the following exception conditions when you invoke external routines:

- A dynamic activation failure of a shareable image or object, which can occur when:
  - An incorrect shareable image or object location is specified.
  - A compilation warning is found in a shareable image or object.
  - A routine entry point cannot be located.
- An external routine execution failure, which can occur when:
  - One of the arguments to the external routine is NULL.

    Oracle Rdb aborts the request and displays a message.
  - An illegal instruction is executed, for example, division by zero.

    Oracle Rdb aborts the request and displays a message.
  - An external routine goes into an infinite loop.

    Oracle Rdb cannot and does not abort a request of this type.

14.15 Understanding the Limitations of External Routines

Oracle Rdb does not allow or support the following:

- Any external routine action that inhibits the functions of the executor manager. Those actions include system calls that would affect the executor process such as SYS$EXIT( ) or exit( ), SYS$DELPRC( ) or kill( ).

OpenVMS Alpha

- Direct calls to OpenVMS Alpha routines that are translated images.

  For such cases, create a jacket routine to invoke the routine in the translated image. Then, compile the jacket routine with the /TIE qualifier, include the jacket routine in a shareable image loaded with the /NONATIVE_ONLY qualifier, and reference the jacket routine and this shareable image in the routine definition. ♦

When you create an external function that calls into the database, keep in mind the following guidelines:

- You cannot use data definition statements.
- You cannot access remote databases.
- You cannot use distributed transactions.
- You cannot pass database or transaction handles to the routine for use in SQL statements. This either generates an exception or causes an indefinite wait situation in the database server.
- The query optimizer always tries to move SQL statistical functions and some predicates, such as EXISTS and IN, to outer levels of a query where they are executed the fewest number of times. This optimization occurs regardless of the presence of an external function declared as VARIANT within the statistical function or predicate.

14.16 Troubleshooting External Routines

This section summarizes common problems you may encounter when executing external routines and suggests the causes and solutions to those problems.

You may encounter the following problems:

- Unexpected results

  These are usually caused by bad parameter data. To correct the error, make sure that:

  - All input parameters have a defined value before calling the routine.
  - All output parameters and any function result are given a defined value by the routine.
  - Arguments are used in a manner that reflects the declared parameter data type and passing mechanism. Be sure to consider any special passing mechanism semantics imposed for the declared language.

- Access violations

These are usually caused by using arguments in a manner that does not reflect the declared parameter data type and passing mechanism.

Test the routine outside the database environment. To detect data type and passing mechanism problems, write a test program in a separate source module to pass expected data instances to the external routine. Using a separate source module can pinpoint mechanism problems which might be obscured by compilers that globally optimize intrasource routine calls.

- Exceptions RTNSBC_INITERR or RTNSBC_TASKERR, reasons 1 through 9

  Generally caused by a quota problem associated with a routine that has a server-site binding.

  Make sure that the user has the required quotas and privileges to allow creating the routine executor process, creating shared memory, and, on OpenVMS, establishing locks and creating mailboxes.

- Exception RTNSBC_TASKERR, reasons 10 through 19

  Generally caused by insufficient shared memory allocated for parameters of routines declared with server-site binding. In addition, if the routine performs database operations, there may be insufficient shared memory allocated for the parameters needed for the database operations.

  Increase the shared memory page count defined by the logical name RDMS$RTX_SHRMEM_PAGE_CNT or the configuration parameter RDB_RTX_SHRMEM_PAGE_CNT.

  When changing the shared memory size, allow 2 pages for the execution control region, 2 pages for each level of call from the server to the executor (callout), and a sufficient number of pages to contain the callout and any database operation (callback) argument list, descriptors, and argument data for all the active callouts and callbacks.

- Exception INVRTNUSE, Image not activated

  Generally, a more common problem for a routine with server-site binding, in which case the file specification or the location clause may refer to OpenVMS logical names which are not defined, or may assume a current directory other than the user's login directory.

  Examine the secondary exceptions for the specific image file specification that could not be found.

- Lock conflicts or deadlocks reported either by an exception or by a bugcheck dump

External routines that attach to databases that are already in use by the user application or other external routines will encounter lock conflicts. In many cases, these lock conflicts are resolved internally. However, certain scenarios can produce unresolvable conflicts or deadlocks that return fatal errors, and a few scenarios can produce severe lock conflict errors that appear to be internal errors and cause a bugcheck dump to be produced.

Refer to the *Oracle Rdb7 Guide to Database Performance and Tuning* to determine how you should rework the application, external routine, or both to avoid lock conflicts.

14.17 Improving Portability and Efficiency of External Routines

Oracle Rdb recommends that you avoid the following when you write external routines:

- Platform-dependent code for routines that are used on multiple operating systems

  For example, avoid logical names, configuration parameters, environment variables, and interactive I/O.
- System service calls and other asynchronous event services
- Exit handlers that perform operations on databases connected by external routines
- Registering the same routine multiple times under different names
- Registering the same routine in multiple databases that are used concurrently by any user
- Recursion involving external routines

Because the definition of the routine in the database contains information about the location of the shareable image or shared object and parameter-passing mechanisms and because that information is often platform-specific, routine definitions are usually not portable.

Part V

Your Program's Context

This part describes the environment in which your program operates:

- Database context
- Transaction context
- SQL connections

15
Attaching to Databases

Before executing an SQL statement, Oracle Rdb must have information about the database context in which the statement executes. This chapter explains how to declare the database context properly for the SQL statements you write. This chapter explains how to:

- Specify and attach to a database
- Specify and attach to a remote database
- Attach to a database in a distributed transaction
- Attach to multiple databases by using aliases
- Detach from a database

15.1 Specifying and Attaching to a Database

Database context identifies the database that you want to access. Your program must attach to a database before it can execute SQL statements.

Most of the work of attaching to a database consists of loading the database (the set of definitions for the database) into memory for use by your process. When you attach to a database, you specify how SQL should access database definitions. You can access database definitions through the repository or directly from a database file.

15.1.1 Specifying File or Repository Access for Database Attachment

You can identify databases with which you want to work in two ways: by file name or by repository path name.

- FILENAME file-specification

  The *file-specification* specifies the source of your database definitions. You type the complete or relative file specification of the Oracle Rdb database.

- PATHNAME path-name

The *path-name* is a character string that represents a node in the repository where database definitions are stored. In programs, you should supply this value indirectly using a logical name or program parameter.

You can always access a database by file name, but you can access a database by repository path name only if database definitions are stored in the repository as well as in a database file.

15.1.2 Specifying the Database Name

To tell SQL that you want to work with a given database, you usually use the ATTACH statement in interactive SQL or the DECLARE ALIAS statement in SQL module language and precompiled SQL. You can use one of the following methods:

- Enter an ATTACH or DECLARE ALIAS statement with a file specification for the database or a path name for the repository.
- Enter an ATTACH or DECLARE ALIAS statement and use a logical name or configuration parameter to refer to either the database file name or the repository path name.

  Logical names and configuration parameters make it easier to maintain your program if you need to move the database or the repository to different disks or nodes. You specify the configuration parameter in the `.dbsrc` file on Digital UNIX.
- Pass a variable containing either a file name or a repository path name.
- Define the logical name SQL$DATABASE or configuration parameter SQL_DATABASE to specify the file specification for the database.

  The SQL precompiler, SQL module processor, and the database run-time system translate SQL$DATABASE or SQL_DATABASE if you do not explicitly declare an alias or attach to a database in a program or in an SQL context file.

  You cannot specify repository access using SQL$DATABASE or SQL_DATABASE. Therefore, if you intend to execute CREATE, ALTER, or DROP statements that include the DICTIONARY IS REQUIRED clause, you should not attach to a database using SQL$DATABASE or SQL_DATABASE.

In host language programs, you should supply the file specification indirectly by using a logical name or configuration parameter. In programs that will be run on various nodes, you may decide to prompt the user for a file specification value and supply it to SQL through a program parameter.

The *Oracle Rdb7 Introduction to SQL* describes in more detail how to attach to databases in interactive SQL.

Although the information you provide when you use the ATTACH or DECLARE ALIAS statements tells SQL to which database you want to attach and whether or not you want repository access, Oracle Rdb may or may not attach to the database or access the repository at the time you provide this information. When Oracle Rdb attaches to a database depends on the environment in which you are working and the way you provide database information. In SQL programs:

- Oracle Rdb usually does not attach to the database until it processes the first executable SQL statement.
- Oracle Rdb *does* immediately attach to the database if you use an ATTACH or CONNECT TO statement.
- If your program declares more than one database, Oracle Rdb attaches concurrently to all declared databases (at the first executable SQL statement). This is true regardless of how many of the databases are accessed by the first transaction started by the program.

You must handle database attachment failures at the first occurrence of the statement that causes SQL to attach to the database. In SQL programs, database attachment occurs when the transaction starts. Depending on what operation your program first performs, the statement that causes databases to be attached could be an ALTER, CREATE, DELETE, DROP, GRANT, INSERT, OPEN, REVOKE, SELECT, SET TRANSACTION, or UPDATE statement.

15.1.3 Specifying Different Databases for Compile Time and Run Time

This section describes why you may choose to attach to your database by path name or file name at different points in your program development cycle.

When you write a program to access a database, you can specify one database definition to be used at compile time and one to be used at run time. The following example declares the database from two different sources:

```
EXEC SQL
       DECLARE LOCAL personnel ALIAS
               COMPILETIME PATHNAME PERS_CDD
               RUNTIME FILENAME PERS;
```

At compile time, the precompiler uses the database definitions in the repository. The PATHNAME clause of the DECLARE ALIAS statement uses the logical name PERS_CDD to point to the repository.

At run time, SQL uses the database definitions in the database file. The FILENAME clause of the DECLARE ALIAS statement uses the logical name or configuration parameter PERS to point to the personnel database.

If you do not specify a run-time option, SQL uses the file name extracted from the repository at compile time, or the one specified in the COMPILETIME FILENAME clause.

The specified source can be either a file specification or a repository path name. However, if you specify access by repository path name at compile time and you do not specify the RUNTIME clause, SQL uses the file name extracted from the repository at compile time.

If your program manipulates or updates only data and not database definitions, file name access is appropriate. But if database definitions are stored in the repository and your program changes data definitions, always specify the PATHNAME qualifier for run time. When you access a database by PATHNAME, any changes you make to database definitions are entered in the repository as well as the database file. This is especially important if other users will include these repository definitions in programs as parameter declarations.

You might want to specify a different database for compile time and run time for the following reasons:

- To permit the user to specify the database at run time

  If you want to accept the file name or path name value into a program parameter at run time, you need some way to obtain this value other than by specifying it explicitly in your program. At compile time, both SQL processors can process a literal, logical name, or configuration parameter in the COMPILETIME clause to find the definitions they need in the repository or a database file. However, SQL does not evaluate parameters in the RUNTIME clause until the program is running. Then, SQL can evaluate the program parameter specified by the RUNTIME clause and attach to the correct database.

- To improve the performance of the SQL precompiler or SQL module processor when dealing with remote databases

  Section 15.2 discusses attaching to remote databases.

15.2 Specifying a Database on a Remote Node

To access an Oracle Rdb database on another system, both systems must use the same communication protocol. That is, both must use TCP/IP or both must use DECnet. In addition, the accounts you use as remote server accounts must be set up correctly.

The *Oracle Rdb7 Installation and Configuration Guide* explains the network protocols and how to set up the remote server accounts.

You specify a database on a remote system by including a node specification in the FILENAME clause. For example, to attach to a database on a remote OpenVMS system, use the following SQL statement:

```
SQL> ATTACH 'FILENAME speedy::disk3:[dept3]personnel';
```

As with local databases, you can specify logical names or configuration parameters for all or part of the file specification.

When you access a remote Oracle Rdb database, you use the Oracle Rdb remote server account. The **remote server account** is the account on the remote node that Oracle Rdb logs into to run Oracle Rdb on the remote node.

Also, you must provide, either implicitly or explicitly, a valid user name and password for an account on the remote system. This **remote user authentication** is the user information on the remote node that Oracle Rdb uses to determine the user's database access privileges.

To provide the user name and password for user authentication on the remote system, use any of the following methods:

- Explicitly, in one of the following ways:
  - In the USER and USING clauses of SQL statements, such as ATTACH or DECLARE ALIAS. See Section 15.2.1 for more information.
  - In qualifiers for the module language or precompiler command line. You use these qualifiers in combination with the USER DEFAULT and USING DEFAULT clauses of the DECLARE ALIAS statement. See Section 15.2.2 for more information.
  - In your client configuration file on the local system

    On Digital UNIX, the configuration file is the `.dbsrc` file; on OpenVMS, it is the RDB$CLIENT_DEFAULTS.DAT file.

    See Section 15.2.3 for more information.
- Implicitly

In certain cases, such as when you attach to a database on a Digital UNIX system from a Digital UNIX system, you do not need to explicitly specify the user name and password *if* the user name and password are the same on both systems. Oracle Rdb implicitly authenticates the user whenever the user attaches to a database. Table 15–1 shows when Oracle Rdb implicitly authenticates the user.

To access the remote server account, use one of the following methods:

OpenVMS VAX OpenVMS Alpha

- Through a proxy account set up for you on the remote node (the recommended approach on OpenVMS). See Section 15.2.4. ♦

OpenVMS VAX OpenVMS Alpha

- Through the RDB$REMOTE default account on OpenVMS for remote access. See Section 15.2.5. ♦

Digital UNIX

- Through the dbsmgr account on Digital UNIX. Oracle Rdb for Digital UNIX sets up the dbsmgr account automatically during installation. You do not modify the account during or after installation, nor do you specify the account name during remote access. ♦

The options you can use when attaching to remote databases depend on the operating systems and the network protocols, as Table 15–1 shows.

Table 15–1 Options for Remote Access

	OpenVMS to OpenVMS	OpenVMS to UNIX	UNIX to UNIX	UNIX to OpenVMS
Remote User Authentication				
Implicit	T[1], D[1]	N/A	T[1]	N/A
USER/USING clauses	T, D	T, D	T, D	T, D
SQL command line qualifiers	T, D	T, D	T, D	T, D
Configuration file	T, D	T, D	T, D	T, D
Remote Server Account				
Proxy account	D	N/A	N/A	D

[1]If the user name and password are the same on both nodes

Key to Network Protocols

T—TCP/IP
D—DECnet
N/A–Not Applicable

(continued on next page)

Table 15–1 (Cont.) Options for Remote Access

	OpenVMS to OpenVMS	OpenVMS to UNIX	UNIX to UNIX	UNIX to OpenVMS
Remote Server Account				
RDB$REMOTE[nn] account	T, D	N/A	N/A	T, D
dbsmgr account	N/A	T, D	T, D	N/A
Key to Network Protocols				
T—TCP/IP D—DECnet N/A–Not Applicable				

The following sections explain each option.

Reference Reading

For more information about setting up remote accounts, refer to the *Oracle Rdb7 Installation and Configuration Guide*. You might also need to refer to the installation and configuration guide for other database access product you use.

15.2.1 Using the USER and USING Clauses for Remote User Authentication

To access databases on remote nodes, you can explicitly provide user name and password information in SQL statements that attach to the database.

You use the USER and USING clauses to embed the user name and password in the ATTACH or DECLARE ALIAS statements or any other SQL statements that attach to a database (such as ALTER DATABASE or CONNECT).

For example, to attach to the mf_personnel database on a remote Digital UNIX node, you can use the USER and USING clauses in the ATTACH statement:

```
SQL> ATTACH 'FILENAME osfrem::/usr/users/heleng/mf_personnel
cont>        USER ''heleng'' USING ''MYpassword''';
```

You must enclose the user name and password in single quotes ('), but because the literal in this example is within the quoted attach-string, you must surround the user name and password with two sets of single quotes.

To avoid placing the user name and password in a program's source code, you can use the DEFAULT keyword for both clauses. Then, use command line qualifiers for the SQL module processor and the SQL precompiler to pass the user name and password.

If you do not specify the USER and USING clause in SQL statements, Oracle Rdb uses the information in the configuration file.

15.2.2 Using Command Line Qualifiers for Remote User Authentication

You can use the following command line qualifiers to specify the user name and password:

Digital UNIX

- On Digital UNIX:
  - For the SQL module processor, the –user and –pass options
  - For the SQL precompiler, the –s -user and –s -pass options ♦

OpenVMS VAX OpenVMS Alpha

- On OpenVMS:
  - For the SQL module processor, the USER_DEFAULT and PASSWORD_DEFAULT qualifiers
  - For the SQL precompiler, the USER_DEFAULT and PASSWORD_ DEFAULT options to the SQL_OPTIONS qualifier ♦

Use these qualifiers in conjunction with the USER DEFAULT and PASSWORD DEFAULT clauses of the DECLARE alias statement. They pass the compile-time user's name and password to the program.

The following example shows how to process an SQL module on Digital UNIX using these qualifiers:

```
$ sqlmod myprog -user heleng -pass MYpassword
```

15.2.3 Using Configuration Files for Remote User Authentication

You can explicitly provide the user name and password for the remote system in the local configuration file, using the SQL_USERNAME and SQL_PASSWORD configuration parameters.

Digital UNIX

On Digital UNIX, provide the information in the local .dbsrc configuration file. ♦

OpenVMS VAX OpenVMS Alpha

On OpenVMS, provide the information in the local configuration file, RDB$CLIENT_DEFAULTS.DAT. ♦

For example, if your user name and password on the remote Digital UNIX node is "heleng" and "MYpassword", add them to your configuration file on the local node, as the following example shows:

```
SQL_USERNAME          heleng
SQL_PASSWORD          MYpassword
```

15.2.4 Using a Proxy Account As the Remote Server Account

OpenVMS VAX OpenVMS Alpha

If you are accessing a remote OpenVMS database, Oracle Rdb recommends that you use a **proxy account**.

A proxy account gives access privileges on a remote node to selected users who do not have a private account on that node. With a proxy account, these selected users gain network access, but they do not have to include an access control string to provide the user name and password for their network login request. Therefore, proxy accounts have several security benefits:

- Passwords are not echoed on the terminal where the request originates.
- Passwords are not passed between systems where they might be intercepted in unencrypted form.
- Users are less tempted to store passwords in command files that would perform the remote access steps.

To maintain the security of your system, a proxy account should not be a privileged account. When a proxy account is set up, it inherits the privileges of the local account.

To attach to a remote database through a proxy account, include a user name in the ATTACH statement. For example, to use the JONES account on NODEB as your remote server account, include the following information:

```
SQL> ATTACH 'FILENAME NODEB"JONES"::USER2:[JONES]PERSONNEL';
```

The proxy account must be assigned process quota values and privileges sufficient for database access. The *Oracle Rdb7 Installation and Configuration Guide* recommends minimal quota values when values normally assigned to OpenVMS users are likely to be inadequate for database operations. In addition, the person who administers the remote database usually must change database protection to allow the proxy account database access.

If the proxy account you are using has inadequate process quotas, your process may not be able to activate required database images on the remote system. In this case, you will probably receive a message that says Oracle Rdb is unavailable to you. If protection for the remote database has not been modified to allow the proxy account access, you will probably receive a message that tells you that you do not have database privileges for the operation you want to perform.

To access a version of Oracle Rdb other than the one specified by the RDB$REMOTE default account, you must define the RDBSERVER and RDMS$VERSION_VARIANT logical names in the proxy account's LOGIN.COM. For example, if Oracle Rdb V6.1 is the default version and

you want to access V7.0, you must add the following to the proxy account's LOGIN.COM:

```
$ DEFINE RDBSERVER SYS$SYSTEM:RDBSERVER70.EXE
$ DEFINE RDMS$VERSION_VARIANT 70                                    ♦
```

15.2.5 Using the RDB$REMOTE Account As the Remote Server Account

OpenVMS VAX OpenVMS Alpha

To attach to a remote database through the RDB$REMOTE default account, the RDB$REMOTE account must be set up correctly on the remote nodes. The *Oracle Rdb7 Installation and Configuration Guide* explains how to set up the account. Then, you specify the node name where the database resides and the database's directory and file specification, as shown in the following example:

```
SQL> ATTACH 'FILENAME speedy::disk3:[dept3]personnel';
```

However, to use a version of Oracle Rdb other than the one specified by the RDB$REMOTE default account, you must specify the name and password of the corresponding account on the remote system. For example, if the RDB$REMOTE account on the remote node is set up for V6.1 of Oracle Rdb, but you want to use V7.0, you must specify the RDB$REMOTE70 account and user name in the access control string:

```
SQL> ATTACH 'FILENAME SPEEDY"RDB$REMOTE70 password"::DISK3:[DEPT3]PERSONNEL';
```

Note

You cannot use an access control string with the TCP/IP network protocol, only with DECnet. For information about accessing an account other than the RDB$REMOTE default account when using TCP/IP, see Section 15.2.6.

To avoid using a password in the SQL statement, set up proxy accounts on the remote node for selected users. Map the proxy account to the version-specific RDB$REMOTE account. For example, to set up the user HELENG to access a V7.0 database on the remote node SPEEDY from the node FASTR, define the proxy account on SPEEDY as shown in the following example:

```
UAF> ADD/PROXY FASTR::HELENG   RDB$REMOTE70/DEFAULT                  ♦
```

15.2.6 Using an Alternate UCX or Internet Service

When you use the TCP/IP network protocol, you can use an alternate service to access a different version of Oracle Rdb on remote OpenVMS systems or to use a different set of configuration defaults on remote Digital UNIX systems. An alternate service is particularly useful in accessing a different version of a Oracle Rdb database on an OpenVMS system from a Digital UNIX system.

If you use the TCP/IP network protocol for remote database access, by default, you use the service RdbServer.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the service uses the user name RDB$REMOTE by default. You can create a new UCX service by using a different user name. You specify that Oracle Rdb use the new service by adding the SQL_ALTERNATE_SERVICE_NAME configuration parameter to your client configuration file. Suppose you create a new service called myservice and specify the user name of the service as RDB$REMOTE61. Add the following line to your client configuration file to access databases using Oracle Rdb V6.1:

```
SQL_ALTERNATE_SERVICE_NAME   myservice                    ♦
```

Digital UNIX

On Digital UNIX, the service uses the user name dbsmgr by default. You cannot change the username, but you can create an alternate service. You specify that Oracle Rdb use the new service by adding the configuration parameter SQL_ALTERNATE_SERVICE_NAME to your client configuration file. Suppose you create a new service called myservice and specify the user name of the service as dbsmgr (the only acceptable value.) Add the following line to your client configuration file:

```
SQL_ALTERNATE_SERVICE_NAME   myservice                    ♦
```

For more information about setting up alternate services and about configuration files, see the *Oracle Rdb7 Installation and Configuration Guide.*

15.3 Avoiding Asynchronous System Traps

You should not use asynchronous system trap (AST) service routines in an application that accesses Oracle Rdb databases. If you do, Oracle Rdb cannot guarantee the behavior of the application.

Because several Oracle Rdb components use AST service routines and because an AST cannot be delivered while an AST service routine is currently executing at the same mode or more privileged mode, using an AST service routine in an application can prevent the delivery of an AST in one of the components of Oracle Rdb.

15.4 Attaching to Databases in a Distributed Transaction

When you use distributed transactions to access databases on remote systems, you may encounter problems unique to distributed transactions. The following sections discuss how to avoid those problems.

15.4.1 Avoiding Undetected Deadlock with Distributed Transactions

When you use distributed transactions to access databases on remote systems, undetected deadlocks may result. **Deadlock** occurs when two users are locking resources that they both need, and neither user can continue until the other user ends a transaction. When deadlock occurs on the same node or the same cluster, the lock manager detects the deadlock and issues the deadlock error condition to one user. However, when a transaction accesses databases on remote systems, the lock manager cannot detect the deadlock.

To help avoid distributed deadlock, Oracle Rdb provides several methods to set the amount of time a transaction waits for locks to be released. Section 16.2.5 explains how to use those options.

See the *Oracle Rdb7 Guide to Distributed Transactions* for more information on avoiding deadlock in distributed transactions.

15.4.2 Avoiding Privilege Errors on Distributed Transactions

When you start a distributed transaction that attaches to a database on a remote node, Oracle Rdb checks that the account on the remote node has the DISTRIBTRAN privilege. For example, if you use a proxy account on the remote node, the proxy account must have the DISTRIBTRAN privilege on that database.

If you do not have the DISTRIBTRAN privilege and you try to start a distributed transaction, Oracle Rdb returns an error and does not start the transaction. This is especially important to remember when you use SQL. SQL starts a distributed transaction by default when you start a transaction that attaches to more than one database.

For more information about database privileges, see the *Oracle Rdb7 Guide to Database Design and Definition* and the *Oracle Rdb7 SQL Reference Manual*.

15.5 Using Aliases for Multiple Attaches

An **alias** is a name you supply to identify a particular attachment to a database. Unlike a file name or repository path name, an alias may differ from one declaration of a database to another. In other words, an alias is a session-specific or program-specific name you give to a database. Once you attach to a database (ATTACH) or declare an alias (DECLARE ALIAS), you refer to the database by the alias in all subsequent statements.

In the following database attach, SP indicates the alias:

```
SQL> ATTACH 'ALIAS SP FILENAME personnel';
```

You can attach to more than one database and thus work with more than one database at the same time by giving each database a different alias. You can also have multiple attachments to the same database if you specify multiple ATTACH statements and these contain different aliases for the same database.

In precompiled SQL, once you identify a database by an alias, you must use the alias name to qualify all references to that database. The SQL precompiler assumes that any unqualified references identify elements from the default database identified by RDB$DBHANDLE. If RDB$DBHANDLE is undefined, you receive an error. If RDB$DBHANDLE is defined, you will be attempting to access a different database than you intended.

When you work with aliases in programs supported by the SQL module processor, specify the alias for the default database in the ALIAS clause of an SQL module header. In the module procedures, you do not have to qualify references to database entities when an alias is declared in the module header.

Generally, you do not have to explicitly specify an alias for a database when you work with one database at a time. If you do not explicitly enter an alias, the SQL precompiler supplies the default alias RDB$DBHANDLE. In the SQL module language, the default is the alias specified in the module header. If an alias is not specified there, SQL assigns the authorization identifier as the alias. If no authorization identifier is assigned, SQL supplies RDB$DBHANDLE as the default.

If you specify your own alias, SQL does not consider the alias a default, and you must always qualify table references with the alias, as Example 15–1 shows.

Example 15–1 Qualifying Table References with an Alias

```
SQL> ATTACH 'ALIAS SP FILENAME personnel';
SQL> SELECT JOB_CODE, JOB_TITLE, MINIMUM_SALARY, MAXIMUM_SALARY
cont>       FROM JOBS LIMIT TO 5 ROWS;
%SQL-F-NODEFDB, There is no default database
SQL> --
SQL> SELECT JOB_CODE, JOB_TITLE, MINIMUM_SALARY, MAXIMUM_SALARY
           FROM SP.JOBS LIMIT TO 5 ROWS;;
JOB_CODE   JOB_TITLE             MINIMUM_SALARY   MAXIMUM_SALARY
APGM       Associate Programmer      $15,000.00       $24,000.00
CLRK       Clerk                     $12,000.00       $20,000.00
ASCK       Assistant Clerk            $7,000.00       $15,000.00
DMGR       Department Manager        $50,000.00      $100,000.00
DSUP       Dept. Supervisor          $36,000.00       $60,000.00
5 rows selected
SQL>
```

A default alias is easy to work with when you are attaching to one database. However, in a multidatabase environment, you should explicitly declare all aliases and provide your own names for them. If you plan to work with more than one database, do not rely on automatic declaration of a default alias for the database with the logical name SQL$DATABASE or configuration parameter SQL_DATABASE.

Example 15–2 shows an excerpt of a SQL precompiled program that declares an alias for one database but uses the default RDB$DBHANDLE for another database.

Example 15–2 Using Aliases

```
  .
  .
  .
 EXEC SQL DECLARE ALIAS FILENAME personnel;
 EXEC SQL DECLARE alias_1 ALIAS FILENAME mf_personnel;
 EXEC SQL SELECT EMPLOYEE_ID, LAST_NAME
                 INTO :employee_id1, :last_name1
                 FROM EMPLOYEES WHERE EMPLOYEE_ID = '00301';
```

```
 EXEC SQL INSERT INTO alias_1.EMPLOYEES
                      (EMPLOYEE_ID, LAST_NAME)
                  VALUES (:employee_id1, :last_name1 );
   .
   .
   .
```

To use two databases at the same time, you must attach to at least one of them using an alias. (The default alias RDB$DBHANDLE can identify only one database.) You must qualify all table references with the alias you specify in a database attach.

Example 15–3 shows part of a program that attaches to two databases at the same time and declares an alias for each.

Example 15–3 Working with More Than One Database

```
/*****************************************************************************
* This routine transfers an employee from the EAST database to the WEST
* database.  It expects the distributed context structure as a parameter.
* The employee record is selected first, deleted from the EAST database
* and inserted into the WEST database.
*****************************************************************************/
   .
   .
   .
EXEC SQL DECLARE east ALIAS FOR FILENAME 2pceast;
EXEC SQL DECLARE west ALIAS FOR FILENAME 2pcwest;
   .
   .
   .
    EXEC SQL USING CONTEXT :local_context
         SELECT EMPLOYEE_ID,LAST_NAME,FIRST_NAME,MIDDLE_INITIAL,
                ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE, SEX, BIRTHDAY
        INTO
        :local_employee.emp_id, :local_employee.emp_last_name,
        :local_employee.emp_first_name, :local_employee.emp_middle_initial,
        :local_employee.emp_address1, :local_employee.emp_city,
        :local_employee.emp_state, :local_employee.emp_zip,
        :local_employee.emp_sex, :local_employee.emp_birth
        FROM east.EMPLOYEES
        WHERE east.EMPLOYEES.EMPLOYEE_ID = :emp_id;
 printf("Deleting the record from the EAST database \n");
```

(continued on next page)

Example 15–3 (Cont.) Working with More Than One Database

```
    EXEC SQL USING CONTEXT :local_context
         DELETE FROM east.EMPLOYEES E
                WHERE E.EMPLOYEE_ID = :emp_id;

        printf("\nInserting the employee record into WEST \n");

    EXEC SQL USING CONTEXT :local_context
         INSERT INTO west.EMPLOYEES
                (EMPLOYEE_ID,LAST_NAME,FIRST_NAME,MIDDLE_INITIAL,
                 ADDRESS_DATA_1,CITY,STATE, POSTAL_CODE, SEX, BIRTHDAY)
         VALUES
                (:local_employee.emp_id, :local_employee.emp_last_name,
                 :local_employee.emp_first_name,
                 :local_employee.emp_middle_initial,
                 :local_employee.emp_address1, :local_employee.emp_city,
                 :local_employee.emp_state, :local_employee.emp_zip,
                 :local_employee.emp_sex, :local_employee.emp_birth);
.
.
.
```

As Example 15–3 shows, you cannot combine data from two different databases in the same data manipulation operation. However, you can work with two databases in the same transaction as long as you are performing different data manipulation operations on each database. In this case, you specify aliases for the databases in a DECLARE TRANSACTION or SET TRANSACTION statement. Section 16.2 discusses these statements in more detail.

15.6 Detaching from a Database

In a program, Oracle Rdb detaches from a database when:

- You enter a DISCONNECT statement

  The DISCONNECT statement performs the following functions:

  - Rolls back any active transactions.
  - Detaches from all databases whose declarations apply to the module in which the DISCONNECT statement is issued. Any databases that are declared with the GLOBAL argument are disconnected also.
  - If the module uses dynamic SQL, the DISCONNECT statement releases any dynamically executed SQL statements in the module, detaches from any dynamically declared aliases in the module, and cancels the characteristics specified in a dynamically executed DECLARE TRANSACTION statement.

- The program stops execution

  The same actions are performed for program termination as for a DISCONNECT statement.

If your program issues a new ATTACH statement specifying an alias that is already in use, you do not cause the previous attachment to end; rather, you receive an error.

16

Managing Transaction Context

This chapter describes transactions and explains how to specify the transaction characteristics so that your SQL programs run correctly and efficiently. The sections that follow explain:

- The concept of a transaction and transaction characteristics
- How to decide what characteristics your transaction needs
- The scope of a transaction
- When to use distributed transactions
- How to control locking of database resources
- How to design transactions so that they do not span input or output
- Constraint evaluation and how to control the timing of the evaluation
- Commit or roll back a transaction

16.1 Understanding Transactions

All access to data in Oracle Rdb databases takes place in the context of a **transaction**. Oracle Rdb treats all SQL statements within the transaction as a single unit; all the statements in the unit complete, or none of them complete. (Statements inside an ATOMIC compound statement may fail separately, however.) Thus, the transaction does not leave partial changes in the database.

For example, suppose that an employee transfers to a new job with a higher rate of pay. In the personnel sample database, this would mean changing and adding rows to the JOB_HISTORY and SALARY_HISTORY tables. If you update each table separately, and an error or hardware failure occurs before all operations in the transaction complete, the database might show that the employee belongs to two departments, has two salaries, or has the old salary with the new job; thus the database would no longer be consistent. To avoid such inconsistencies, include all update tasks in a single transaction. Then,

you can monitor errors and roll back the entire set of operations explicitly when an error occurs.

Your program determines when the transaction starts and when it ends. You specify what access you want other users to have to the database while the transaction is running, what characteristics you want the transaction to have, and whether Oracle Rdb should make the changes permanent (commit them) or remove them from the database (roll them back) when you are done.

16.1.1 Understanding Transaction Characteristics

You can specify many transactions characteristics, including the following:

- The type of access to the database or databases

  You can request read-only, read/write, or batch-update access to the database. The default is read/write access to all attached databases.

  A read-only transaction reads data but cannot change it. It cannot delete or update old data or insert new rows.

  A read/write transaction can perform update as well as retrieval operations. Many transaction characteristics in this list make sense only in the context of a read/write transaction.

  A batch-update transaction is a special-purpose transaction that is *not* for general use. Batch-update transactions cannot be rolled back because they do not write to a recovery-unit journal file. Therefore, a batch-update transaction that encounters an error or ends abnormally corrupts the database. Section 16.2.3 describes the advantages and hazards of using a batch-update transaction.

- The databases you plan to access

  A transaction is associated with one or more databases. If you plan to access only one database, you do not need to specify it. Likewise, if you intend to access every database in the same way, you do not need to specify those databases. In this case, SQL automatically applies the same transaction characteristics to each database.

  If you are working with one or more databases using different transaction characteristics (for example, read-only for some and read/write for others), you must define aliases in the DECLARE ALIAS or ATTACH statements for those databases. Then, you must specify the aliases when you start the transaction. Section 16.2.8 describes how to start a transaction using aliases. Section 16.4 describes what to do if some of those databases are on different systems.

- The tables you plan to use

You can explicitly name the tables you want to work with in a RESERVING clause. If you list tables in a RESERVING clause, you cannot access tables you did not list.

If you do not explicitly name the tables, Oracle Rdb automatically reserves tables as required at the time the transaction performs an operation.

- A lock type

  The lock type tells Oracle Rdb how you plan to use each table. Read access allows you to retrieve information but not change, delete, or insert it. Write access allows you to read, change, add, or delete data. Data definition access allows you to create or drop indexes in the tables you specify, but not to perform any other operations.

  By default, Oracle Rdb locks rows as required when the transaction performs an operation.

- A share mode

  The share mode determines how much access other users may have to each table during your transaction. You can allow other transactions shared access to the tables you are using, restrict them to reading but not writing, or exclude them altogether.

- A wait option

  The wait option specifies whether you want to wait for locks on tables or rows until other users release them (WAIT), or want to receive an immediate lock conflict message (NOWAIT). The default is WAIT.

- An isolation level

  The isolation level specifies how much data written by other transactions you want your transaction to see.

  A serializable isolation level means that if you retrieve a particular row more than once during the course of a transaction, you see the same version of that row each time you retrieve it (assuming that you have not updated or deleted that row between the times that you look at it). Less stringent isolation levels allow you to see changes made by other users, even if those changes are inconsistent with what your transaction previously saw. The default is a serializable isolation level.

  See Section 16.2.6 for more information about choosing an isolation level.

- The time when constraints are to be evaluated.

  See Section 16.7.

Before you write a program to access the database, you should determine the tasks you want to accomplish. Some of these tasks might be:

- Data retrievals from tables in one or more databases
- Changes to existing rows in tables
- Changes to database definitions

If these tasks depend on each other, you should include them in the same transaction. If the tasks are not related, you should include them in separate transactions.

16.1.2 Deciding When to Modify Transaction Characteristics

The defaults for transaction characteristics are reasonable for many transactions, particularly for transactions started by users who update the database and who must share access to a database with other users. The defaults are also reasonable for many report applications that require data that is completely up-to-date and, therefore, should not process previous versions of rows that have been written to a snapshot file.

Change the defaults in any of the following situations, assuming the database must support simultaneous access by multiple users:

- When you enter database queries or generate reports that can include data that is not absolutely up-to-date

  Specify READ ONLY in the transaction statement. Assuming that a snapshot file was enabled for the database, this allows other database users to access the tables. You may also want to reset your isolation level from serializable to repeatable read or read committed. Reducing a transaction's isolation level can improve application performance.

- When your application should take priority over other users; therefore, you want to prevent interference by others after your transaction starts

  Explicitly reserve tables and upgrade table share modes to protected or exclusive. (Table share modes are explained in Section 16.2.4 and Section 16.5.)

- When you modify database definitions (using the CREATE, DELETE, ALTER, GRANT, or REVOKE statement)

  The SHARED DATA DEFINITION option allows others to access the database while you make changes to indexes. To protect database integrity, Oracle Rdb does not let you change most data definitions until no other users are accessing the affected tables in the database. Oracle Rdb behaves as though you explicitly specified restricted access to these tables by other transactions, even though you may not have done so explicitly.

Despite the automatic protection provided by Oracle Rdb, you should explicitly request the most restrictive access to tables affected by a definition change you plan to make. If you do, you know as soon as you attempt to start a transaction whether or not you have the access you need to complete the changes you intend to make.

If you are changing only index definitions, you can reserve the affected tables for SHARED or EXCLUSIVE DATA DEFINITION. No other transaction can query the table if you specify SHARED DATA DEFINITION, but they can concurrently define indexes. If you request a table for EXCLUSIVE DATA DEFINITION, no one else can access the table. No one, including you, can access the table for other purposes.

- When you use a database product that does not support one or more of the default transaction characteristics

  Explicitly specify the transaction characteristics the other product needs.

Section 16.2 describes how and when to specify transaction characteristics.

16.2 Specifying Transaction Characteristics in SQL Programs

Understanding the relationship between transaction characteristics and locking is very important when a database is simultaneously accessed by multiple users. For example, if you use transaction characteristics that are more restrictive to other users than those you need, you unnecessarily lock out other users from the data. In addition, your transaction may be difficult to start if other users are already accessing tables you reserve.

Conversely, if you explicitly specify transaction characteristics that are not restrictive enough for a certain kind of task, one of the following two events occurs:

- Oracle Rdb overrides your specifications if doing so is necessary to protect the database.
- Your transaction encounters lock conflicts or unacceptable delays produced by the interference of other users.

To explicitly control transaction characteristics, use either the DECLARE TRANSACTION or the SET TRANSACTION statement.

The DECLARE TRANSACTION statement is not executable and therefore does not start a transaction. When you use the DECLARE TRANSACTION statement, SQL starts a transaction with the first SQL statement that executes following either the declaration or a COMMIT or ROLLBACK statement. In the latter case (following a COMMIT or ROLLBACK statement), SQL applies the transaction characteristics you declare to the next transaction you start.

The DECLARE TRANSACTION statement offers the following advantages:

- It can establish transaction defaults for an interactive SQL session or program.
- You can include it an SQL context file.

  To manage database and transaction contexts, you must provide information specific to Oracle Rdb that is not included in the standard SQL language. For this reason, the SQL interface to Oracle Rdb lets you choose between including such information in a program source file or including it some other way. One option available for SQL module language and embedded SQL programs is to include database and transaction information in an SQL context file. Refer to Section 5.5 for information about using context files with the SQL module processor and Section 6.6.2 for information about using context files with the SQL precompiler.

When you use the DECLARE TRANSACTION statement, all transactions have the same characteristics, unless they were started by explicit SET TRANSACTION statements.

In programs, you are limited to one DECLARE TRANSACTION statement per SQL module or precompiled source file. To establish different default transaction characteristics in programs that use a DECLARE TRANSACTION statement, distribute the SQL statements in the program among separate files and apply different DECLARE TRANSACTION statements to each file. When you want to change transaction characteristics within one source file, use a SET TRANSACTION statement for each transaction you start.

The SET TRANSACTION statement is an executable statement that both specifies and starts a transaction. You can include multiple SET TRANSACTION statements in a precompiled source file or in an SQL module. The SET TRANSACTION statement offers the following advantages:

- It lets you explicitly control when transactions start.
- It provides flexibility for changing transaction characteristics in a program source file.

Both statements allow you to explicitly specify transaction characteristics. The following sections describe those options in detail.

Section 16.3 explains how to use SET TRANSACTION and DECLARE TRANSACTION statements together.

16.2.1 Using Read-Only Transactions

Specifying a read-only transaction limits your transaction to data retrieval; however, it permits more users to access the database concurrently than would be possible if you specify a read/write transaction.

Specify READ ONLY in a DECLARE TRANSACTION or SET TRANSACTION statement when:

- You do not intend to add new rows or change existing values in the database.
- You do not intend to create, alter, or drop data definitions or to grant or revoke database privileges.
- You do not require absolute, up-to-the-minute accuracy. A read-only transaction sees the database as it was at the moment the transaction began—in effect, a "snapshot" of the database. Unless your application requires an absolutely current picture of the database, you should be able to use a read-only transaction.
- You are accessing a read-only storage area.

  Read-only storage areas do not have snapshot files because you cannot update data in read-only storage areas. (Any discussion in this chapter about snapshot files and locking does not apply to read-only storage areas.)
- You are using Oracle Rdb with a database product that requires read-only access to data.

  Note that the other product might use entirely different locking rules and strategies than Oracle Rdb.

When you start a read-only transaction, your transaction reads current versions of rows from the database file and previous versions of rows from the snapshot file. Because many transactions can share the versions of rows in the snapshot file, your transaction does not conflict with others.

If you start a read-only transaction without specifying a RESERVING clause, your transaction requests access to any table in the database. Your database operations are restricted to retrieval only. Your transaction does not use any row locks in this transaction mode and other transactions can access the same data. If other transactions make changes, your transaction still retrieves the old record from the snapshot file. Because your transaction reads data from the snapshot file, other transactions are free to update rows in any table without waiting or experiencing access conflicts.

You can specify a RESERVING clause in combination with a read-only transaction, but you include the clause only to specify (limit) the tables you can work with during the transaction. The shared read mode is the only combination of share mode and lock type keywords you can specify for a read-only transaction.

If a snapshot file is not enabled for your database (and you are not accessing a read-only storage area), a read-only transaction has the same effect as starting a read/write transaction that reserves tables using a shared read reserving option. If snapshots are enabled deferred, the read-only transaction waits until the snapshot file is updated, then proceeds as normal.

In most cases, the previous versions of rows stored in a snapshot file are adequate; they will likely be outdated only by a matter of seconds or minutes. However, if your retrieval transaction requires an absolutely current picture of the database, start a read/write transaction so that you do not use the snapshot file.

If your retrieval transaction generates a complex report that your site considers high priority, you may also want to reserve the tables that you need to read in protected share mode. Interference from others may make your transaction more difficult to start in a high-contention environment, but once your transaction is underway, it encounters minimal interference from others and completes more quickly.

16.2.2 Using Read/Write Transactions

Specify a read/write transaction when you want to perform retrieval tasks that require an absolutely current picture of the database. In addition, you must use a read/write transaction to modify schema definitions with the INSERT, DELETE, UPDATE, CREATE, ALTER, DROP, GRANT, or REVOKE statements.

The following statement lets you perform operations on any table or view that is not reserved in a conflicting mode by another transaction:

```
SET TRANSACTION READ WRITE;
```

Oracle Rdb reserves the tables as you name them in SQL statements and, depending on the type of operations your transaction performs, places read locks on selected rows to complete a retrieval task and write locks on selected rows to complete an update task.

For example, the first data manipulation statements in your transaction might retrieve data from the EMPLOYEES table. Oracle Rdb locks tables and rows in the same way as specified by a shared read reserving option. Later in the transaction, you might modify values in selected rows in the EMPLOYEES

table. Oracle Rdb, using only the necessary locks to complete the transaction, promotes the shared read locks to shared write.

Data definition statements lock system tables, which contain definitions for the database. Depending on the change you make, the result is that data in one, several, or all tables in the database is inaccessible to other users until you end your transaction. (In other words, the effect is the same as specifying EXCLUSIVE WRITE access to those tables.)

Some database operations require a higher level of protection than your statement specifies. In such cases, Oracle Rdb automatically promotes the mode to protected read or protected write to complete the task. Oracle Rdb always attempts to secure the highest share mode and lock type necessary for the protection of your transactions and the database. Although the share mode may be higher than what you explicitly specify, it is never lower than that level once you start processing data.

16.2.3 Using Batch-Update Transactions

Batch-update transactions provide a performance advantage over read/write transactions, at the cost of recovery and concurrent access. Batch-update transactions work faster than read/write transactions that reserve tables using the exclusive write reserving option, because the batch-update transaction does not write to snapshot or recovery-unit journal files. As a result, disk I/O operations are kept to an absolute minimum. However, that means batch-update transactions must be terminated by a COMMIT statement because the information necessary to roll back a transaction is not available. If a batch-update transaction fails, the database will be corrupt. In this case, your only options are to rebuild the database entirely or restore it from a backup file.

No other users can access a database during a batch-update transaction.

You can use a batch-update transaction to load a database quickly. For example, assume you are refreshing all the data in an extracted database. (An **extracted database** is a copy of some or all data in another database, and is usually created to support impromptu queries or reports.) You have decided that the performance advantage offered by a batch-update transaction is a worthwhile trade-off against possible corruption of the second database that would occur if an abnormal event such as a system failure occurs before the transaction is committed. *You should make a backup copy of the second database immediately prior to starting this transaction*, so that you can restore the database if it becomes corrupted. The following example shows how to start a batch-update transaction on one database and a read-only transaction on another database:

```
SET TRANSACTION
     ON FIRST_DB USING (READ ONLY)
 AND ON SECOND_DB USING (BATCH UPDATE);
```

The preceding transaction starts only if no other users are accessing the database specified as SECOND_DB.

Without the support of a recovery-unit journal, transactions cannot be rolled back. Therefore, you must always terminate batch-update transactions with a COMMIT statement. If an error condition is not handled with a COMMIT statement or if a system or other kind of failure prevents a commit operation from being performed, the database being accessed by the batch-update transaction is marked as corrupt and it cannot be accessed again by anyone.

Specify BATCH UPDATE only from an explicit SET TRANSACTION statement. Use it only when increasing the performance of an update transaction justifies the risk of possible database corruption in the event of abnormal termination of an interactive session or program. This risk may be justified when:

- You are loading a database initially.

  You should have a program or SQL command procedure available to create database definitions again if the load operation fails. If the batch-update transaction fails before it can be committed, you can delete the corrupted database file, redefine the database, and start the load operation again.

- You are performing extensive update operations to most (if not all) the tables in a database.

  You should use the RMU Backup command to back up the database immediately prior to beginning the batch-update transaction. Should the batch-update transaction fail before it can be committed, you can delete the corrupted database file or files, use the RMU Restore command to restore the database from the backup copy, and start the updates again.

16.2.4 Using the RESERVING Clause

If you do not enter a transaction statement, SQL assumes you want to start a transaction using all the attached or declared databases. It behaves as though you have entered the following statement:

```
DECLARE TRANSACTION READ WRITE WAIT;
```

SQL reserves tables as you refer to them in statements. SQL also puts read or write locks on rows as appropriate when you access the rows.

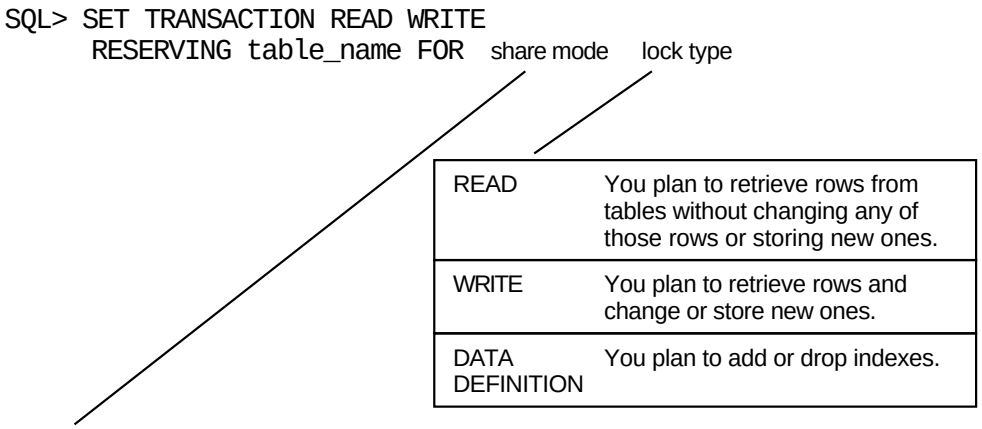
If you know when you start the transaction which tables you will access, you can explicitly reserve them. Oracle Rdb places an intent lock on the entire table. Your program may have to wait for other transactions to release incompatible locks on the table and you may cause other transactions to wait for your transaction to complete.

In read/write transactions, you can specify the following combinations of share mode and lock type in RESERVING clauses:

- SHARED READ
- SHARED WRITE
- SHARED DATA DEFINITION
- PROTECTED READ
- PROTECTED WRITE
- EXCLUSIVE READ
- EXCLUSIVE WRITE
- EXCLUSIVE DATA DEFINITION

Figure 16–1 shows the SET TRANSACTION RESERVING syntax and summarizes the share modes and lock types that apply to read/write transactions.

Figure 16–1 Share Mode and Lock Type Options for Read/Write Transactions



NU−2111A−RA

You can use the RESERVING clause with a read-only transaction. For example:

```
SET TRANSACTION READ ONLY
    RESERVING EMPLOYEES FOR SHARED READ;
```

In this case, you cannot access tables you do not specify in the RESERVING clause. Moreover, Oracle Rdb does not evaluate protected or exclusive share modes or the write lock type when you combine READ ONLY and a RESERVING clause.

To reserve more than one table for your transaction, specify each table explicitly. The following example reserves three tables for read-only access:

```
SET TRANSACTION READ ONLY RESERVING
        EMPLOYEES FOR SHARED READ,
        JOBS FOR SHARED READ,
        DEPARTMENTS FOR SHARED READ;
```

The transaction started by this statement cannot read from tables that you do not specify.

The following SET TRANSACTION statement names different tables for different database tasks (EMPLOYEES for retrieval and COLLEGES and DEGREES for update):

```
SET TRANSACTION READ WRITE RESERVING
            EMPLOYEES FOR SHARED READ,
            COLLEGES FOR SHARED WRITE,
            DEGREES FOR EXCLUSIVE WRITE;
```

The following statement reserves two tables. Use this statement when you want to read data from one table and store it in the other table and you want to avoid locks on any row. This statement prevents other users from accessing either table:

```
SET TRANSACTION READ WRITE RESERVING
           EMPLOYEES FOR EXCLUSIVE READ,
           EMPLOYEES_TEMP FOR EXCLUSIVE WRITE;
```

You cannot start the preceding transaction until all other transactions accessing either table have completed.

Remember that you may also lock tables that you have not explicitly reserved. This happens when a view, constraint, or trigger in the tables that you have reserved refers to other tables. Note the following:

- When a trigger or constraint first accesses a table, Oracle Rdb automatically reserves the table in shared read mode.
- If a trigger updates a table, Oracle Rdb automatically reserves the table in shared write mode.
- If you explicitly reserve a view, Oracle Rdb automatically reserves, in the same share mode, any table to which the view refers.

If another user is accessing any of the tables you need, you can encounter a lock on that table. To prevent locking problems later, explicitly reserve all the tables that you need to access.

Oracle Rdb does not automatically reserve tables to which a computed-by column refers. You must explicitly reserve those tables.

16.2.5 Choosing Whether to Wait for Locks to Be Obtained

You can specify the WAIT option if you want to wait for locks on tables or rows until other users release them, or specify the NOWAIT option if you want to receive an immediate lock conflict message when other users are locking the table you want to access. The WAIT option is the default. For example, the following transaction waits for the locks it needs:

```
SET TRANSACTION READ ONLY;
```

If you want your program to handle lock conflicts directly, you can specify the NOWAIT option in your transaction statement. When another transaction locks a row and your transaction has the no-wait characteristic, you receive a lock-conflict error to inform you that the row is unavailable. You may try to retrieve the row again by ending the current transaction and again executing the statement that accesses the row. Your statement succeeds only after the transaction holding the lock on the row ends.

You specify the NOWAIT option only when you want to be notified immediately about a lock conflict. For example:

```
SET TRANSACTION READ WRITE RESERVING
        EMPLOYEES FOR PROTECTED WRITE,
        JOB_HISTORY FOR PROTECTED WRITE,
        SALARY_HISTORY FOR SHARED READ NOWAIT;
```

The NOWAIT option lets your program decide how to handle lock conflicts. Your program can allow the user to choose whether or not to wait, or it can retry a certain number of times before informing the user that the record is unavailable. For example, to permit the user to choose whether or not to wait, your program could contain the following logic:

1. Start a transaction specifying NOWAIT.
2. Start the data manipulation operation.
3. If the data manipulation statement encounters a lock conflict, roll back the NOWAIT transaction and display a message to the user: "The record you want is in use. Do you want to wait?"
4. If the user replies Y or y, start a transaction specifying WAIT. If the user replies N or n, handle the condition as appropriate for the application.

Specify a wait interval to set the amount of time a transaction waits for locks to be released

To specify an application-specific wait interval, use the WAIT clause of the SET TRANSACTION or DECLARE TRANSACTION statement. The following example shows a SET TRANSACTION statement with a wait interval of 15 seconds:

```
SET TRANSACTION READ WRITE WAIT 15;
```

Oracle Rdb waits 15 seconds for a lock to be released before returning a lock conflict error. (You express the wait interval in seconds, but the exact time period is approximate.)

Oracle Rdb provides several methods to specify a wait interval; the methods can apply to every transaction in your process, to an entire database, or to an entire system. If more than one option for a wait interval applies to your transaction, Oracle Rdb uses the minimum value—that is, it waits the shortest amount of time specified by any wait-interval option. The *Oracle Rdb7 SQL Reference Manual* explains these options in detail and describes how Oracle Rdb decides which wait interval to use.

Whether you specify the wait interval using the WAIT clause in the SET TRANSACTION statement or using another method, Oracle Rdb returns a lock-conflict error if the resource is still locked after the transaction waits the specified interval.

16.2.6 Choosing an Isolation Level

The **isolation level** of a transaction defines the degree to which the read operations of one transaction can be affected by the update operations of other concurrently executing transactions.

Oracle Rdb provides the following three isolation levels for transactions:

- SERIALIZABLE guarantees that the operations of concurrently executing transactions are not affected by any other transaction. Concurrent execution of serializable transactions must produce the same results as would be produced by the execution of the same transactions in a one-after-the-other order.

  By default, all Oracle Rdb transactions run at ISOLATION LEVEL SERIALIZABLE.

- REPEATABLE READ guarantees that if you execute the same query again, your program receives the same rows it read the first time. However, you may also see rows inserted and committed by other transactions. These rows, called **phantoms**, can lead to data inconsistency if your program performs operations that rely on the aggregate properties of the range, such as COUNT, AVERAGE, and so forth.

- READ COMMITTED allows your transaction to see all data committed by other transactions. Oracle Rdb releases read locks when the cursor advances to the next row or the cursor is closed. Thus, data items your transaction reads can be updated and committed by another transaction before your transaction finishes. Therefore, your application cannot rely on data that it reads to remain unchanged. However, data items cannot be changed while you are displaying them.

You explicitly specify the isolation levels in the SET TRANSACTION and DECLARE TRANSACTION statements.

Isolation levels affect only read/write transactions. Read-only transactions always read from the snapshot file, if it is enabled, and thus never see other transactions' changes. Furthermore, only read operations in a read/write transaction are affected, and then only if your program executes the same query more than once in the same transaction.

Oracle Rdb places long-term locks on various database resources while executing a transaction. Serializable transactions ensure data consistency but limit the ability of multiple users to access database objects simultaneously. Reduced isolation levels sacrifice data consistency under well-defined circumstances but can increase transaction concurrency. If your transaction can run correctly with lower levels of protection, the database system may be able to provide increased concurrency for the database. You need to balance your program's needs for consistency and protection against the overall performance of the database.

Table 16–1 shows the phenomena permitted for the isolation levels.

Table 16–1 Phenomena Permitted at Each Isolation Level

Isolation Level	Nonrepeatable Reads Allowed?	Phantoms Allowed?
READ COMMITTED	Yes	Yes
REPEATABLE READ	No	Yes
SERIALIZABLE	No	No

Note

If you reserve a table with a RESERVING clause, that RESERVING clause may override the behavior the specified isolation level implies. For example, Oracle Rdb always prevents phantoms in a table explicitly reserved for protected retrieval. If you reserve some tables for protected retrieval and others for concurrent retrieval, Oracle Rdb does not attempt to prevent phantoms in the tables reserved for concurrent retrieval.

For all isolation levels, if an UPDATE ONLY cursor reads a row, Oracle Rdb locks the row exclusively and holds the lock until you issue a COMMIT or ROLLBACK statement.

16.2.6.1 Using a Serializable Transaction

When you specify a serializable isolation level, Oracle Rdb guarantees that if you retrieve a row multiple times during a transaction, you will retrieve the same version of the row, assuming your own transaction does not change it. If your own transaction does change the row, you will see the changed values if you request the same row again. Other transactions may commit changes to the row after your transaction starts but before you retrieve the row the first time.

However, after you have retrieved the row the first time, you cannot see changes made by other transactions. Your first retrieval of the row places a read lock on it that prevents other transactions from updating or deleting it until your transaction finishes and releases the lock.

16.2.6.2 Using a Repeatable Read Transaction

When you specify a repeatable read isolation level, Oracle Rdb guarantees that a query repeated multiple times during a single transaction will return (for each execution of the query) the same rows as were returned the first time the query was executed. Although a repeatable read transaction guarantees that you see the same set of rows, it cannot guarantee that you will not see extra rows called phantoms.

The following example shows a repeatable read transaction that finds a phantom row. The repeatable read transaction queries the database to select a set of rows. The query returns six rows in this example:

```
ATTACH 'FILENAME mf_personnel';
SET TRANSACTION READ WRITE
     ISOLATION LEVEL REPEATABLE READ;
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL
       FROM EMPLOYEES WHERE EMPLOYEE_ID < '00170';

 EMPLOYEE_ID   LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
 00164         Toliver          Alvin        A.
 00165         Smith            Terry        D.
 00166         Dietrich         Rick         NULL
 00167         Kilpatrick       Janet        NULL
 00168         Nash             Norman       NULL
 00169         Gray             Susan        O.
6 rows selected
```

While the repeatable read transaction is executing, another transaction starts, inserts a single row into the mf_personnel database, and commits the transaction:

```
ATTACH 'FILENAME mf_personnel';
SET TRANSACTION READ WRITE;
INSERT INTO EMPLOYEES
             (EMPLOYEE_ID, LAST_NAME, FIRST_NAME,MIDDLE_INITIAL)
       VALUES ('00161','Muggs','Fred','J');
1 row inserted
COMMIT;
```

If the first transaction executes the same query again, the query returns the same six rows as before, but it also returns the phantom row (Fred Muggs) inserted by the second transaction, as shown in the following example:

```
SELECT EMPLOYEE_ID, LAST_NAME,FIRST_NAME,MIDDLE_INITIAL
       FROM EMPLOYEES WHERE EMPLOYEE_ID < '00170';
 EMPLOYEE_ID   LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
 00161         Muggs            Fred         J.
 00164         Toliver          Alvin        A.
 00165         Smith            Terry        D.
 00166         Dietrich         Rick         NULL
 00167         Kilpatrick       Janet        NULL
 00168         Nash             Norman       NULL
 00169         Gray             Susan        O.
7 rows selected
```

Oracle Rdb guarantees that reads are repeatable; that is, for each execution of the query, Oracle Rdb returns at least the rows first selected. Oracle Rdb cannot guarantee that other rows (phantoms) will not appear.

A repeatable read transaction holds short-term locks on indexes and holds locks on the table until the end of the transaction.

When using the REPEATABLE READ isolation level, Oracle Rdb occasionally holds long-term read locks on rows that are not really required to prevent the nonrepeatable read phenomenon. The REPEATABLE READ isolation level reduces index contention, not data contention.

16.2.6.3 Using a Read Committed Transaction

When you specify a read committed isolation level, Oracle Rdb cannot guarantee that a query repeated multiple times during a single transaction will return (for each execution of the query) the same rows as were returned the first time the query was executed. In addition, a read committed transaction cannot guarantee that you will not observe extra rows called phantoms. The number of rows and the data in those rows can change in a read committed transaction because a read committed transaction allows the reading of data committed by other concurrently executing transactions.

Many applications return inconsistent results when they permit nonrepeatable read/write transactions, such as when an application executes transactions at the READ COMMITTED isolation level. A well-known example, referred to as the buried update anomaly, illustrates this fact:

- Two transactions read the same row and find two items in stock.
- The first transaction subtracts one item from the stock total and updates and commits the result.
- A second transaction also subtracts one item from the stock total, which its original read indicates is still two, and updates and commits the result.

Although the two transactions have actually reduced the number of items by two, the row on disk still shows one item in stock. The second update transaction wrote over the first update. A subsequent transaction will incorrectly read the row containing the item count and find one item in stock when there should be none. This is called a **buried update anomaly**.

To prevent buried update anomalies, you must run the item stock transactions at REPEATABLE READ or SERIALIZABLE isolation level, not at READ COMMITTED.

The following example shows a read committed transaction that queries the database to select a set of rows. The query returns six rows in this example:

```
SET TRANSACTION READ WRITE
       ISOLATION LEVEL READ COMMITTED;
SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL
       FROM EMPLOYEES WHERE EMPLOYEE_ID < '00170';

 EMPLOYEE_ID   LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
 00164         Toliver          Alvin        A.
 00165         Smith            Terry        D.
 00166         Dietrich         Rick         NULL
 00167         Kilpatrick       Janet        NULL
 00168         Nash             Norman       NULL
 00169         Gray             Susan        O.
6 rows selected
```

Meanwhile, another transaction starts, inserts a single row into the mf_personnel database, updates another row, and commits the transaction:

```
SET TRANSACTION READ WRITE;
INSERT INTO EMPLOYEES
              (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL)
       VALUES  ('00161','Muggs','Fred','J');
1 row inserted
```

```
UPDATE EMPLOYEES
       SET FIRST_NAME = 'David',
           LAST_NAME = 'Garroway',
           MIDDLE_INITIAL = 'E'
        WHERE EMPLOYEE_ID = '00164';
1 row updated
COMMIT;
```

When the first transaction requests the same range of ID numbers as before, the query returns the changes made by the second transaction— the phantom row (Fred Muggs) and the David Garroway row, instead of the Alvin Toliver row:

```
SELECT  EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL
        FROM EMPLOYEES  WHERE EMPLOYEE_ID < '00170';

 EMPLOYEE_ID   LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
 00161         Muggs            Fred         J.
 00164         Garroway         David        E.
 00165         Smith            Terry        D.
 00166         Dietrich         Rick         NULL
 00167         Kilpatrick       Janet        NULL
 00168         Nash             Norman       NULL
 00169         Gray             Susan        O.
7 rows selected
COMMIT;
```

In a repeatable read transaction, Oracle Rdb would have redisplayed the Alvin Toliver row (as it did in the first execution of the query) and the update in the other concurrently executing transaction would have had to wait.

A read committed transaction holds short-term locks on indexes and tables.

When a sequential scan is done under the READ COMMITTED isolation level, the number of lock operations may increase. If adjustable lock granularity is disabled, the number of lock operations (and the total number of locks) might increase to unacceptable levels.

16.2.7 Benefits of Using Various Isolation Levels

Using the SERIALIZABLE isolation level enables all transactions to run correctly in relationship to concurrently executing transactions without special programming or application design. Tracing incorrect inventory figures to specific cases of buried updates and debugging tasks can be time consuming. Discovering the reason for inconsistencies in reports can take considerable time as well. The SERIALIZABLE isolation level relieves you of these time-consuming tasks.

Nonetheless, the benefits of serializable transactions come at a cost. To eliminate harmful phenomena, the database system uses long-term read locks on objects that it touches, such as data records and index nodes. Concurrency drops as a result of holding the long-term locks. A transaction that does not need phantom protection to execute correctly, but is running at the SERIALIZABLE isolation level can block transactions from inserting unrelated rows. This can reduce overall performance for the application. By running this transaction at the REPEATABLE READ isolation level, the database system need not maintain long-term read locks, allowing the overall throughput of the application to increase.

Applications that perform communications during a transaction can make good use of read committed transactions, which do not hold long-term read locks on data records or index nodes. Long-running report transactions can also use read committed transactions because they interfere minimally with update transactions. If you need to run a large number of reports, or the reports require serializable protection, a read-only transaction is a better choice than the READ COMMITTED isolation level in terms of a reducing locks.

Although user-selectable isolation levels enable you to tune individual transactions for optimal concurrency and consistency, the flexibility of using various isolation levels can nonetheless introduce data inconsistencies from application design or programming errors. Some applications can run and produce predictable results at isolation levels below the most stringent and default isolation level SERIALIZABLE. Think carefully about whether or not these applications will behave consistently as expected if nonrepeatable reads or phantoms occur. In general, you can run applications at reduced isolation levels when the application does not include a transaction that performs the same query expression two or more times.

Reduced isolation level applications can be divided into two classes: reporting and update.

Using Reduced Isolation Levels for Reporting Applications

You can produce many useful reports or one-of-a-kind inquiries from applications permitting nonrepeatable reads and phantoms. Anyone looking for estimates, for example, can tolerate inconsistencies that would be unacceptable under other circumstances, such as for end-of-period reports that might demand greater precision.

By their nature, one-time reports or queries are immune to nonrepeatable reads and phantoms. For example, a query that makes one pass over sales data, calculating the percentage of each item sold in the last week, does not need the protections afforded by serializable transactions. Such reports

or queries can suit your purposes when run at the READ COMMITTED or REPEATABLE READ isolation levels.

Using Reduced Isolation Levels for Update Applications

Carefully constructed update transactions can often run correctly despite potential interference from nonrepeatable reads and phantoms; however, you might have to add some code to your applications to accomplish this.

Suppose that a customer orders 100 T-shirts of assorted colors. The transaction that processes the order displays how many of each color are in stock. It then allows you to enter the number of each color to be added to the customer's order. You can run this transaction and allow nonrepeatable reads and phantoms (using the READ COMMITTED isolation level) and still avoid the buried update anomaly by including some logic in your program.

Suppose your application reads the row for blue T-shirts and finds 25 in stock. The customer asks you to send him 10. Because the transaction executes at isolation level READ COMMITTED, you cannot update the row to record that there are 15 blue T-shirts left in stock; another transaction might have altered the row between the time your application read and updated it.

So, you must read the row again to make sure no one, except you, can modify it. To do this, you can open a second cursor declared with the UPDATE ONLY clause to reread the row. The application must then recalculate the remaining inventory by subtracting 10 from the value in the reread row, and then issue the update.

This is an alternative to including the read and update operations in the same transaction, ensuring nobody else can read the record while your transaction is working with it.

Using Isolation Levels with Databases Other Than Oracle Rdb

Databases other than Oracle Rdb databases may require you to specify a particular isolation level, frequently ISOLATION LEVEL READ COMMITTED. If you are using SQL with a database product other than Oracle Rdb specify the isolation level required by that product.

16.2.8 Using Aliases to Access More Than One Database in a Single Transaction

If you want to read data from one table in one Oracle Rdb database and store it in another table in another Oracle Rdb database, you must use aliases. In the following example, FIRST_DB and SECOND_DB are aliases specified in ATTACH or DECLARE ALIAS statements:

```
SET TRANSACTION
    ON FIRST_DB USING (READ ONLY
       RESERVING FIRST_DB.EMPLOYEES FOR SHARED READ)
AND ON SECOND_DB USING (READ WRITE
       RESERVING SECOND_DB.EMPLOYEES FOR EXCLUSIVE WRITE);
```

If you use multiple aliases, you must qualify all database names with the alias that identifies the database where the object appears. Otherwise, Oracle Rdb assumes the object comes from the default database and looks for it there. If you have not established a default database, Oracle Rdb returns an error; if the default database is not the one you wanted, you might receive an error or an unexpected result, depending on whether the default database contains an object with the name you specified.

Chapter 17 discusses aliases in more detail.

16.3 Understanding the Scope of a Transaction

The executable statement that marks the start of a transaction and the statement that commits or rolls back the changes to the database, thus ending the transaction, identify the **scope** of the transaction. Oracle Rdb executes either all of the statements in the scope of the transaction or none of them.

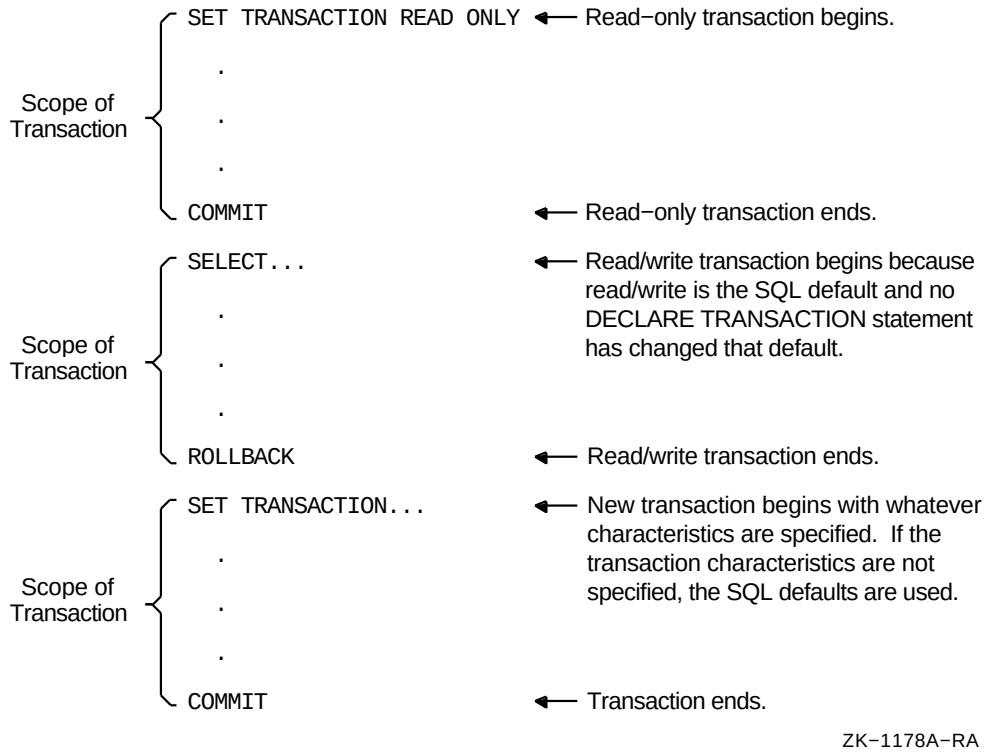
Your program can start a transaction explicitly, using the SET TRANSACTION statement, or implicitly, by executing a statement that requires database access. SQL starts an implicit transaction on the first SQL statement that executes during an interactive SQL session or program and on the first statement that executes following a COMMIT or ROLLBACK statement.

When it starts an implicit transaction, Oracle Rdb uses the current default values for transaction characteristics. These defaults might be defaults set by a DECLARE TRANSACTION statement or, if you don't specify a DECLARE TRANSACTION statement, the Oracle Rdb defaults. The Oracle Rdb default transaction is a read/write transaction that waits for the locks it needs and runs at ISOLATION LEVEL SERIALIZABLE.

Your program can allow Oracle Rdb to implicitly end a transaction or it can explicitly end the transaction by issuing a COMMIT statement to make all your changes permanent or a ROLLBACK statement to remove all changes. Section 16.8 explains your options for ending a transaction.

Figure 16–2 illustrates transaction scope using the SET TRANSACTION statement. The SET TRANSACTION statement works the same way in programs as it does in interactive SQL; that is, it specifies one transaction and starts it.

Figure 16–2 Transaction Scope with a SET TRANSACTION Statement



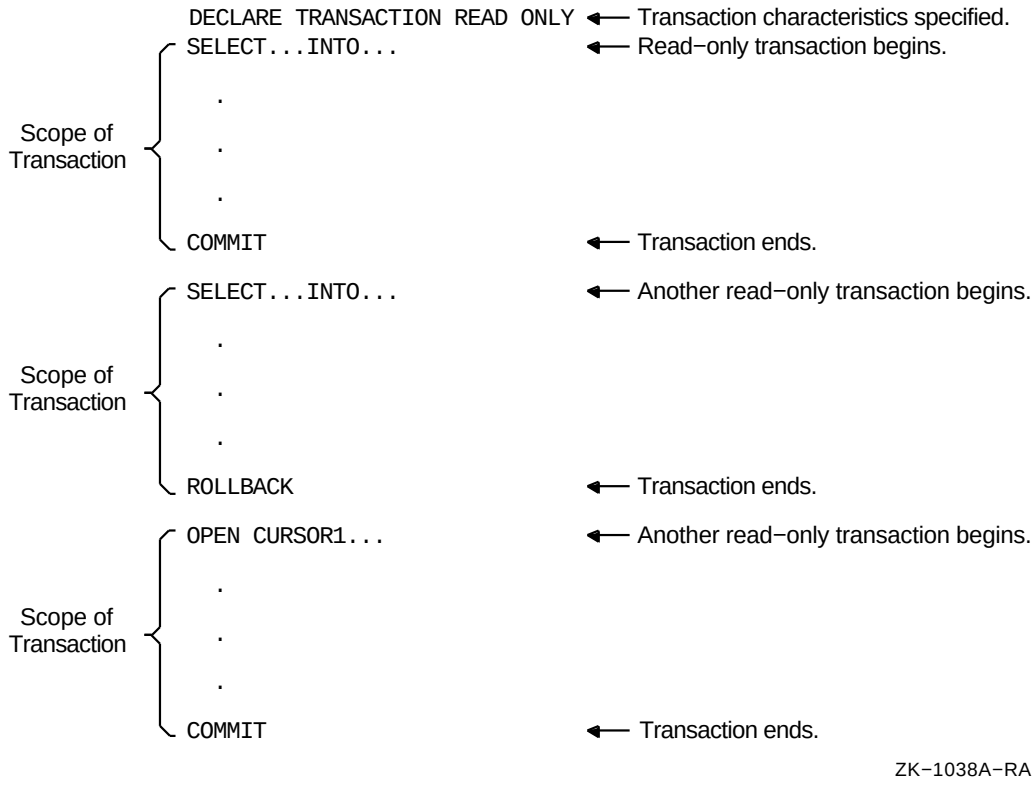
The DECLARE TRANSACTION statement establishes defaults for any transactions started implicitly by statements other than SET TRANSACTION. If, for example, an OPEN statement starts a transaction, the transaction inherits characteristics from the DECLARE TRANSACTION statement. However, if a SET TRANSACTION statement starts a transaction, SQL ignores the DECLARE TRANSACTION statement. Any unspecified transaction characteristics are the Oracle Rdb defaults.

The DECLARE TRANSACTION statement can be useful if you want defaults for your session or program to be different than Oracle Rdb defaults. For example, if your interactive SQL session or program includes queries against a database and all queries should be read-only to avoid locking rows, you can specify DECLARE TRANSACTION READ ONLY just once at the beginning of your session or source file to ensure that your queries avoid locking rows. Conversely, if you specify READ ONLY using a SET TRANSACTION statement, you must enter the statement again after every COMMIT or

ROLLBACK statement to ensure that subsequent queries are processed as read-only transactions.

Figure 16–3 illustrates transaction scope using a DECLARE TRANSACTION statement in a file submitted to the SQL precompiler or SQL module processor.

Figure 16–3 Transaction Scope with a DECLARE TRANSACTION Statement



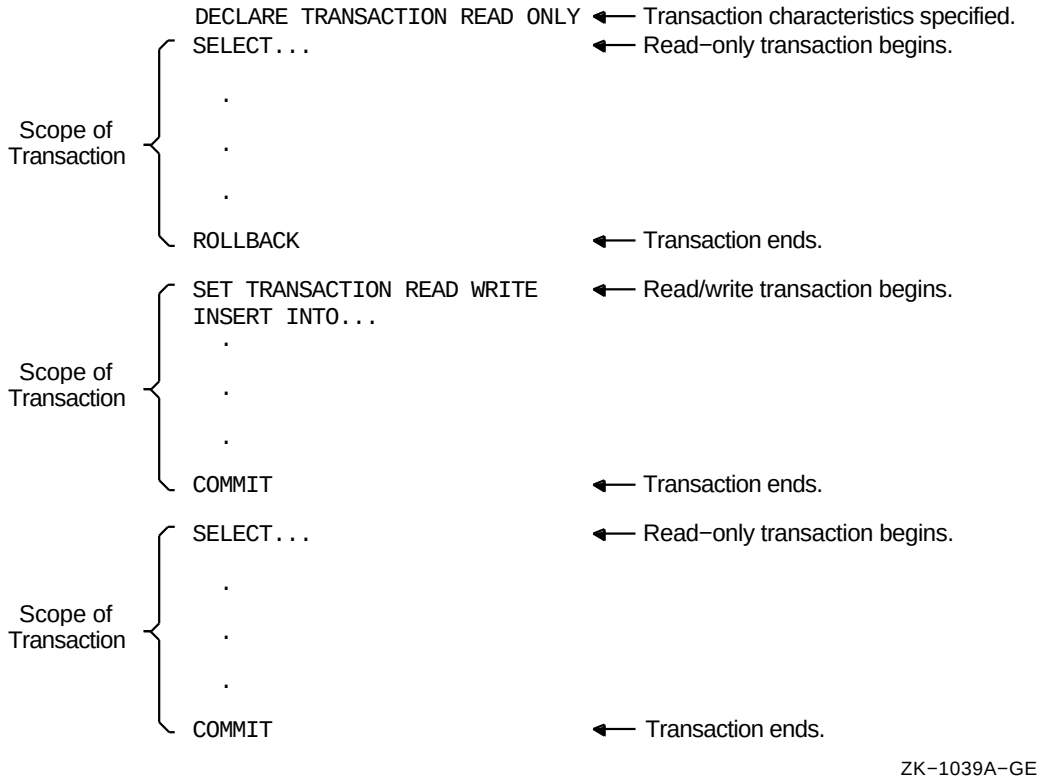
You may also find a use for mixing one DECLARE TRANSACTION and several SET TRANSACTION statements in programs. However, too much variation in the approach to transaction management increases the likelihood of inadvertently starting the wrong kind of transaction at different points in your program. In general, you should explicitly specify the kind of access you want: what tables you will be using and the share mode, lock type, and isolation level you need. The more specific you are about your needs, the more efficient and maintainable your database operations will be.

Reference Reading

The *Oracle Rdb7 SQL Reference Manual* describes the DECLARE TRANSACTION and SET TRANSACTION statements in detail.

Figure 16–4 shows a combination of SET TRANSACTION and DECLARE TRANSACTION statements.

Figure 16–4 Transaction Scope with SET and DECLARE TRANSACTION Statements



16.4 Using Distributed Transactions

A **distributed transaction** groups more than one database or more than one database attachment together into one transaction. The databases can be on the same node or on different nodes. A transaction guarantees that if one operation in a transaction cannot complete, none of the operations complete.

However, in an ordinary multidatabase transaction, it is possible for one database to commit successfully and one to fail.

To ensure that data remains consistent even if your application attaches to more than one database on more than one node, SQL uses the two-phase commit protocol to ensure that every operation in a transaction completes before a transaction is made permanent, even if the transaction is a distributed transaction.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, if a transaction accesses databases on more than one node and DECdtm services are active on your system, Oracle Rdb starts a two-phase commit transaction by default. ♦

A two-phase commit transaction works by first querying each participant in the transaction about whether it is able to commit. If even one participant fails, the whole transaction fails. Only when each and every participant has indicated it can commit does the final commit take place.

For information about the two-phase commit protocol and distributed transactions, see the *Oracle Rdb7 Guide to Distributed Transactions*.

If your application does not use a distributed transaction and a hardware or software problem causes the system to fail, any incomplete transactions are automatically rolled back when you restart the system. Section 16.8 describes in more detail when a transaction is rolled back or committed.

16.5 Locking Database Resources

Locking is the mechanism by which Oracle Rdb controls concurrency and enforces the logical and physical integrity of the database. Locks are used to:

- Synchronize access to the physical database. For instance, pages and logical areas such as tables and indexes are locked as they are used.
- Keep track of events. For instance, the monitor uses a process termination lock to determine when a user is no longer accessing the database.
- Keep your view of the database logically consistent.

Your program does not explicitly control locking. Rather, you specify your needs for protection and access. Oracle Rdb then determines what locking scheme meets your needs while still allowing the most user concurrency and protecting the database's integrity. The actual locks Oracle Rdb acquires might not be what you expect or even what you request.

The following sections describe locking for read/write transactions. A read-only transaction does not lock tables and rows unless snapshots are disabled. If snapshots are disabled, a read-only transaction behaves the same as a read/write transaction. Section 16.5.4 explains read-only transactions and snapshot files.

16.5.1 Locking Strategies

Oracle Rdb uses a simple strategy for locking objects:

- Lock the object.
- Perform the work on the object—for instance, retrieve a physical page from disk or update a row.
- Unlock the object at some later time, most likely at the end of the transaction unless some other condition allows the lock to be freed sooner. Physical page locks, for instance, can be released if another transaction needs to read a different row from the same physical page.

In general, Oracle Rdb tries to lock at the highest level possible. For instance, if you select one row from a 500,000-row table, and no other transaction is using that table, Oracle Rdb might choose to place the lock on the entire table. Thus when you select other rows, no new locks need to be acquired. Only if another transaction requested compatible access to the table would smaller portions of the table—index nodes or individual rows—be locked. This is called **adjustable lock granularity**.

16.5.2 Intent Locks

Whether you start a default transaction or you explicitly reserve tables for a particular access and share mode, Oracle Rdb takes out locks on a table-by-table basis. In a transaction without a RESERVING clause, Oracle Rdb does not lock the tables until an SQL statement accesses them; at that time, Oracle Rdb chooses an appropriate lock based on the operation being performed. When you explicitly reserve tables, Oracle Rdb locks each table at the start of the transaction, placing an **intent lock** on the table.

Table 16–2 shows the intent lock modes, their meaning, and the allowed activities.

Table 16–2 Intent Locks

Lock type	Meaning	Allowed Activities
CR	Concurrent Read	The current transaction explicitly locks every record it reads. Other transactions may read from and write to the same table.
CW	Concurrent Write	The current transaction explicitly locks every record it reads or writes. Other transactions may read from and write to the same table.
PR	Protected Read	The current transaction does not need to lock any records it reads. Other transactions may read from, but not write to, the same table.
PW	Protected Write	The current transaction does not need to lock any records it reads, but it locks every record it modifies or adds. Other transactions may read in CR mode.
EX	Exclusive	The current transaction does not need to lock any record it reads or modifies because no other access to the table is allowed. Other transactions must wait until this transaction finishes before they can have any access to the table.

By default, each transaction holds all locks until the transaction ends. Other transactions cannot change the rows you have locked. In many circumstances, they cannot even read the same rows. Further, depending on how the table is structured and how Oracle Rdb processes your query, it is possible to lock records your transaction does not directly request. For instance, if you request a row that must be located by going through the table one row at a time, you lock the entire table one row at a time. Your transaction may also lock tables to which constraints and triggers refer. Again, these locks are held until your transaction ends. Other transactions cannot use any records you have locked in a manner that is inconsistent with the usage you have requested.

Similarly, if another transaction is already using any of the rows, tables, or indexes that you need, you have to wait for the resource to be released or else handle the lock-conflict error. (Section 16.5.3 explains lock conflicts. Section 10.5 tells how to handle lock conflict and deadlock situations in your program.)

If you cannot gain the access you requested to all the tables you want to reserve, your transaction will not start.

The kind of lock (read or write) placed on a row is determined by the nature of the statement that is processing the row. An OPEN statement, for example, places only read locks on the rows in a cursor, even if you specify the write lock type for the table in a RESERVING clause. You must fetch a row from the opened cursor and then execute an UPDATE or DELETE statement on the row before Oracle Rdb upgrades the read lock to a write lock.

When you reserve tables using the protected or exclusive mode, you minimize the time it takes for your transaction to complete because you reduce the availability of data to other users. The exclusive share mode, for example, does not allow other users to access a table. If a table is not available to other users, you prevent someone else's transaction from locking rows you plan to process.

Deciding if you want to specify share modes and lock types is important only when you start a read/write transaction. The decision is not important for either read-only or batch-update transactions.

16.5.3 Lock Conflicts

When you lock a row, you may or may not conflict with other users, as shown in Figure 16–5. When a conflict occurs, if the second transaction has specified the NOWAIT option, it receives an error immediately. If the second transaction has specified the WAIT option, it waits until one of the following occurs:

- The lock is freed by the first transaction, at which point the second transaction can proceed
- A deadlock is detected, at which point one of the users receives a deadlock error and the other proceeds
- The wait timeout interval expires, at which point the second transaction receives an error.

Figure 16–5 shows when other users wait for row locks to be released and when they encounter a lock-conflict error. Assume that the wait characteristic (the default) applies to other users' transactions. If the no-wait characteristic applies, "a conflict" would appear wherever the chart indicates "a wait." (Note that the table does not indicate when users encounter deadlock errors. Unlike lock-conflict errors, deadlock errors are not easily predicted.)

Remember that other users must wait until you end your transaction for locks to be released. A read/write transaction with shared read access to a table places read locks on all rows that it processes. This may mean that all the rows in a table are locked if you are sequentially searching rows in the table rather than accessing rows by unique index values. Because other users cannot

update rows on which your transaction has read locks, you may prevent users from updating any row in the table until your transaction ends.

If You Access a Row Using Transaction Mode:	Someone Else Using: READ ONLY Has:	READ WRITE SHARED READ Has:	READ WRITE SHARED WRITE Has:	READ WRITE PROTECTED READ Has:	READ WRITE PROTECTED WRITE Has:	READ WRITE EXCLUSIVE READ Has:	READ WRITE EXCLUSIVE WRITE Has:
READ ONLY	No conflict	No conflict	No conflict	No conflict	No conflict	A wait	A wait
READ WRITE SHARED READ	No conflict	No conflict	No conflict to read, a wait to update	No conflict	No conflict to read, a wait to update	A wait	A wait
READ WRITE SHARED WRITE	No conflict	No conflict	No conflict to read a row not updated; otherwise, a wait	A wait	A wait	A wait	A wait
READ WRITE PROTECTED READ	No conflict	No conflict	A wait	No conflict	A wait	A wait	A wait
READ WRITE PROTECTED WRITE	No conflict	No conflict	A wait	A wait	A wait	A wait	A wait
READ WRITE EXCLUSIVE READ	A wait	A wait	A wait	A wait	A wait	A wait	A wait
READ WRITE EXCLUSIVE WRITE	A wait, then a conflict	A wait	A wait	A wait	A wait	A wait	A wait

ZK–1484A–GE

Figure 16–5 Chart of Database Access Conflicts

In all update cases, Oracle Rdb does not allow other read/write transactions to read changed rows until the updating transaction commits or rolls back the transaction. Because Oracle Rdb locks your rows against access by other transactions, you can display the changes you have made to those rows. This row locking assures the consistency and integrity of database rows.

To improve concurrent access to tables, it is particularly important to be as specific as possible when you specify rows in cursor declarations. As soon as it opens a cursor, your transaction places read locks on all rows in the table associated with the cursor and (if you do not access tables by index) other rows as well. In a read/write transaction, if you open a cursor that accesses a large number of rows, you lock out other users who wish to update the database until you commit or roll back the transaction. For example, if you write an application that starts a transaction, opens a cursor, lets a data entry clerk examine each row and update or delete the row, and commits all the changes at once, your application will probably stop operations, even retrieval tasks, of other users who start read/write transactions after you do. As a general rule, cursors opened for an interactive update should be limited to one or a few rows and should be closed as quickly as possible with a COMMIT or ROLLBACK statement. (Executing a CLOSE statement does not release locks.)

Keep in mind that Oracle Rdb may lock a table for exclusive access even if you request shared access, if it is necessary to protect database integrity and consistency. This occurs during certain data definition statements and during other statements as needed.

16.5.4 Read-Only Transactions and the Snapshot File

When snapshots are enabled for a database (the default), read-only transactions do not lock the rows they read. Rather, when an update occurs, Oracle Rdb writes the previous version of rows, the **before-images**, to the snapshot file. The read-only transaction reads the before-images directly from the snapshot file.

If a row is updated multiple times, multiple before-images are written to the snapshot file. All before-images for a row are retained long enough to guarantee that other transactions can read the version of the record that was current when the update transaction started.

Read-only transactions that access an Oracle Rdb database with the snapshot file disabled, however, *do* place read locks on rows processed during the transaction. When there is no snapshot file (and you are not accessing a read-only storage area), read-only transactions are processed as read/write transactions.

Read-only storage areas do not have snapshot files because you cannot update data in read-only storage areas.

16.5.5 Encountering Lock-Conflict Errors with Read-Only Transactions

Transactions that reserve tables in the exclusive share mode update the recovery-unit journal file but not the snapshot file. Eliminating the chore of updating a snapshot file improves the performance of a read/write transaction by reducing the amount of disk I/O operations associated with an update. However, because read-only transactions depend on snapshot file update, a transaction that reserves tables in the exclusive share mode disables all read-only transactions that attempt to access the same tables. If your transaction is a high-priority transaction and you want it to encounter the least interference possible from other read/write transactions, but do not want to interfere with read-only transactions, reserve tables for protected write.

Read-only transactions that access a snapshot file (and normally should not encounter locks at all) sometimes encounter lock-conflict errors. This happens when a read-only transaction tries to access a table that has been reserved by a read/write transaction using the exclusive write option, or when any table in the database is being accessed by a batch-update transaction. (Batch-update transactions reserve all tables in the database using the exclusive write option.) When a read-only transaction encounters either kind of conflict, the read-only transaction is always treated as though it had the no-wait characteristic and it encounters a lock-conflict error.

The following steps illustrate the conflict between read-only transactions and read/write transactions that reserve a table in exclusive-write mode:

1. User A reserves a table for EXCLUSIVE WRITE and fetches a collection of rows.
2. User B starts a read-only WAIT transaction without a RESERVING clause.
3. User B attempts to access the table reserved by User A.
4. User B waits until the earlier exclusive-write transaction commits or rolls back and then receives a lock-conflict error.
5. Even if User A terminates the current transaction, releases all locks, and exits SQL, User B's transaction will encounter the error if it tries to access that table again in the same transaction.

A transaction that specifies EXCLUSIVE WRITE does not write data to the snapshot file. However, all data committed to the database before a read-only transaction starts must be available to that read-only transaction. Oracle Rdb cannot determine whether a transaction in exclusive share mode has written data to the snapshot file; therefore, the read-only transaction's requirements

can never be satisfied. In this case, the WAIT option specified by User B is rejected; thus, User B receives an error rather than waiting forever.

16.5.6 Improving Concurrent Access

Indexes that contain unique values and well-designed indexes that contain duplicate values almost always improve multiuser access to single tables when the following conditions apply:

- The table is large.
- Users access the table concurrently, but by different index values.

To improve concurrent access to a table, use a unique index to search the table, instead of a sequential (nonindexed) search. Nonindexed searches process rows from the beginning of the table to the end and place read locks on every table row. When indexes are unique, a user searching a single table by one index value is unlikely to lock out users who need to use the index to search for other values. When you retrieve rows by unique index value, you are most likely to place read locks only on the table rows you need.

However, indexes are database structures whose entry points (nodes) can be locked by user access just as rows in tables can be locked by user access. Therefore, your transaction may encounter a lock-conflict error or may have to wait for another transaction to end because of a lock on the node for a sorted index. Such a lock may be part of the reason why a user encounters a deadlock error.

Sorted indexes are most likely to cause such conflicts. The unique index values in columns that hold ID numbers, timestamps, order numbers, and so forth frequently increase by 1, and often the most recent rows are the rows most in demand. The likelihood that several needed records will appear in the same index node is quite high. Hashed indexes are less likely to cause contention because they are distributed more evenly through the database. However, hashed indexes can be defined only for multifile databases. Partitioning indexes so that different applications can access a different set of nodes may also help.

Searching tables using sorted indexes that allow duplicates can lock out other users as effectively as a sequential search of the table. The problem is related to the way a sorted index is structured, and arises when there are few index values (for example, only three department code values in an index based on department code). In this case, the structure for the sorted index has so few nodes that it can be completely locked by only one user (who is searching for only one index value).

To avoid this problem, create a sorted index that has fewer duplicates—perhaps basing it on more than one column. For example, if there are few department code values, a sorted index based on both department code and supervisor identification number (ID) increases the number of values in the sorted index and thereby increases the number of nodes for the index. Another solution is to create a hashed index in addition to a sorted index. Keep in mind that Oracle Rdb chooses which index to use depending on whether your query is a range retrieval or an exact match retrieval.

It is important to emphasize that you cannot predict whether a table will be searched sequentially or by index, especially when the query involves a join operation or accesses a view based on multiple tables. For these cases, the Oracle Rdb query optimizer decides which table is searched first and whether the search is done sequentially, by index, or (if both hashed and sorted indexes are available) by a particular index. The query optimizer makes this decision based on a variety of factors, among them the relative size of the tables being joined.

If the query optimizer determines that a join operation in your transaction can be executed most quickly by searching a particular table first, and that it must look at all the rows in that table, your transaction places read locks on all the rows in that table. Other users will not be able to update any rows in that table until your transaction ends. Conversely, your transaction may encounter a lock-conflict error or delays if another transaction has already obtained a conflicting lock on the table.

For example, to evaluate the following query, the query optimizer may read every row in one table to find a starting set of values for which the other tables can be checked. The query optimizer may decide that using the EMPLOYEES table for the starting set of values is fastest because EMPLOYEES is the smallest table. Conversely, the query optimizer may decide that overall query performance is best served by first searching the larger tables to eliminate rows with end-date values that are not null. In any event, you can be reasonably sure that the following query will place read locks on every row of at least one table:

```
SELECT LAST_NAME, FIRST_NAME, JOB_CODE, JOB_START, SALARY_AMOUNT
       FROM EMPLOYEES E, JOB_HISTORY JH, SALARY_HISTORY SH
       WHERE (
             (JH.EMPLOYEE_ID = E.EMPLOYEE_ID)
             AND
             (SH.EMPLOYEE_ID = JH.EMPLOYEE_ID)
             )
             AND (JOB_END IS NULL)
             AND (SALARY_END IS NULL);
```

Thus, your application has only indirect control over what locks will be placed on which objects. Accessing all tables by index values is still the best policy. By joining tables on columns that are indexed for all tables, you allow the query optimizer to choose among access methods rather than forcing it to perform a sequential search of a particular table.

The *Oracle Rdb7 Guide to Database Design and Definition* provides more information about indexes. The *Oracle Rdb7 Guide to Database Performance and Tuning* tells you how to locate and debug locking problems and explains the query optimizer and its potential strategies in detail and offers suggestions for dealing with problems.

16.6 Designing Transactions so They Do Not Span Terminal I/O Operations

In many programs, interactive users provide values used for database retrieval and update. In addition, many interactive users may be concurrently running one program image. In these cases, users should not be using transaction time to read error messages and make corrections to input. You may therefore decide to keep all users' transactions as short as possible rather than ensure that one user's database operation never encounters changes made by others. This tradeoff is the best way to keep any one user from waiting an unacceptably long time or encountering locking errors because of interference by other users. This tradeoff also implies that your program allows interactive users to add, change, or delete only one row in a table per transaction when many users are likely to simultaneously update the data in the same table.

Data update tasks often involve table searches that place read locks on rows and indexes before placing write locks on rows and indexes. When one user causes another user to encounter a long wait or a lock-conflict or deadlock error, the problem is usually caused by one of the following situations:

- One user has write locks on rows or index nodes to which another user needs read access.
- One user has read locks on rows or index nodes to which another user needs to write.

If you allow interactive users to update, delete, or insert only one row in a table per transaction, you reduce both the number and duration of write locks that interfere with data retrieval. For single-row transactions, using verb time for constraint evaluation reduces unnecessary write locks. When you specify constraint evaluation at verb time, a row change that causes a constraint violation is automatically undone without any action on the part of the program.

You can avoid including terminal I/O operation time within a transaction by rolling back a transaction after you retrieve a row for user verification or after a fatal error. As a result, you reduce the length of time a read lock exists on a row or index node and therefore reduce the likelihood that the lock interferes with a write operation.

Keeping terminal I/O operation time out of a transaction can be difficult because often you do not want a user to have to enter all input values for a row again after a rollback operation. You also have to handle the possibility (however remote) that, if you roll back a transaction after a user error, the row on which that user is working may be deleted or updated by another user before the first user enters the corrected input.

This section provides two pseudocode examples to illustrate how you can trade off the risk of a user being adversely affected by operations performed by others against the need to keep transactions short.

Example 16–1 illustrates how to update a row in a table (EMPLOYEES) that may be concurrently accessed by many users. The example logic involves retrieving a row twice, first by index value and then by database key (dbkey) value. (Retrieval by dbkey value, instead of index value, avoids encountering a write lock that occurs because an index node is being updated.) The example assumes that two users will not be updating the same row at the same time. If you cannot make this assumption for the operation your application performs, either do not use a multiple-transaction approach to a user task, or alter it significantly.

Example 16–1 Updating a Row in a Multiuser Environment

```
Set the flag to indicate that EMPLOYEE_ID is invalid and initialize
       other parameters.
Perform VERIFY_EMPLOYEE section until a valid EMPLOYEE_ID is obtained.
Perform UPDATE_EMPLOYEES section.

VERIFY_EMPLOYEE section:

Prompt the user for EMPLOYEE_ID value to determine which employee row
       needs an update.
Verify that input characters represent a valid string for an employee
       identification number; prompt again if necessary.

SET TRANSACTION READ WRITE RESERVING EMPLOYEES FOR SHARED READ

Check the status parameter to ensure that the transaction started
       successfully; if not:
       Handle deadlock and lock conflict with timed retry of the
       SET TRANSACTION statement, or
       Stop the program with appropriate messages for other errors.
```

(continued on next page)

Example 16–1 (Cont.) Updating a Row in a Multiuser Environment

```
Select the row in the EMPLOYEES table that has the input ID number.

Check the status parameter to ensure that the operation was successful;
       if not:
       Handle deadlock and lock conflict with the ROLLBACK statement and
       the timed retry of the SET TRANSACTION statement and row retrieval,
       or display messages, roll back, and stop the program for
       unexpected errors.

If no row is found:
       Roll back.
       Tell the user that the entered employee number is not assigned
       to any employee and to enter a new value.
If row is found:
       Set a predefined flag to indicate the EMPLOYEE_ID exists.
       Store the dbkey value for the row in a parameter.
       Store the row in the first of three identical sets of program
            parameters.
       Roll back.
       Display the row values stored in the parameters on the terminal.

UPDATE_EMPLOYEES section:

Prompt for change values and store them in the second set of parameters
       for the row.
Verify that valid characters are entered for all entries and prompt
       for corrections if necessary.

SET TRANSACTION READ WRITE RESERVING EMPLOYEES FOR SHARED WRITE

Check the status parameter to ensure that the transaction started
       successfully; if not:
       Handle deadlock or lock conflict with timed retry of the
       SET TRANSACTION statement, or
       Stop the program with appropriate messages for other errors.

Retrieve the row by the dbkey value and store it in the third set
       of parameters for the row.
If no row found:
       Roll back.
       Tell the user that the row was unexpectedly deleted by
           another user.
       Exit the program or loop.
If row is found:
       Compare the values in the first and third sets of parameters
       to determine if another user has changed row.
       If yes:
          Roll back.
          Tell the user that an unexpected change occurred to the row.
          Stop the program.
       If no:
          Update the row, setting column values to those input by the
            user.
```

(continued on next page)

Example 16–1 (Cont.) Updating a Row in a Multiuser Environment

```
Check the status parameter to ensure that the operation was
       successful; if not:
       Handle lock conflict or deadlock with ROLLBACK and
       timed retry of the entire transaction.
       Stop the program for unexpected errors.
COMMIT
```

Example 16–2 updates an employee's job history information by inserting a row with information about the employee's new job into the JOB_HISTORY table and by updating the row that stores information about the employee's last job. This problem is different than the problem in Example 16–1 because of intertable constraints. The example involves numerous but brief read/write transactions when an interactive user makes input errors that violate many constraints. However, no transaction spans terminal I/O operations, and the user is prompted again only for values that remain invalid.

Example 16–2 Updating a Table Containing Constraints

```
Perform INITIALIZE_FLAGS section.
Perform VERIFY_EMPLOYEE section until a valid EMPLOYEE_ID is obtained.
Perform VERIFY_ROW_VALUES section until valid SUPERVISOR_ID, JOB_CODE,
        and DEPARTMENT_CODE values are obtained.
Perform UPDATE_JOB_HISTORY section.

INITIALIZE_FLAGS section:

Initialize all flags to indicate "invalid".  (Flags are set to "valid"
       when values input by the user have passed program checks.)

VERIFY_EMPLOYEE section:

Prompt the user for the EMPLOYEE_ID value to determine for which
       employee the job history information needs an update.
Verify that input characters represent a valid string for an
       employee identification number; prompt again if necessary.

SET TRANSACTION READ WRITE RESERVING EMPLOYEES FOR SHARED READ

Check the status parameter to ensure that the transaction started
       successfully; if not:
            Handle deadlock and lock conflict with timed retry.
            Stop the program for unexpected errors.

Select the row in the EMPLOYEES table that has the input ID number.
```

(continued on next page)

Example 16–2 (Cont.) Updating a Table Containing Constraints

```
Check the status parameter to ensure that the operation was successful;
       if not:
            Handle deadlock and lock conflict with ROLLBACK and timed
            retry of the statement.
            Roll back.
            Stop the program for unexpected errors.

If no row is found:
       Roll back.
       Tell the user that the entered employee ID number is not yet
       assigned to any employee in the database; therefore, no update
       to the JOB_HISTORY table using that number is allowed.

If a row is found:
       Set a predefined flag to indicate EMPLOYEE_ID exists.
       Perhaps display the employee ID number and name for the user.
       Roll back.


VERIFY_ROW_VALUES section:

Prompt for the JOB_START date and for any of the following values
       whose associated flag is set to "invalid":
       JOB_CODE, DEPARTMENT_CODE, SUPERVISOR_ID.
Verify that valid characters have been entered for first or revised
       entries.
       Prompt for corrections if necessary.

If the flag for SUPERVISOR_ID is set to "invalid":
       SET TRANSACTION READ WRITE
           RESERVING EMPLOYEES FOR SHARED READ
       Check the status parameter to ensure that the transaction
       started successfully; if not:
            Handle deadlock and lock conflict with timed retry of
            the SET TRANSACTION statement.
            Stop the program for unexpected errors.
            Select the row in the EMPLOYEES table that has the
            input supervisor ID.
       Check the status parameter to ensure that the operation was
       successful; if not:
            Handle deadlock and lock conflict with ROLLBACK and
            timed retry of the statement.
            Stop the program for unexpected errors.
       If a row is found, roll back and set the predefined flag
            to indicate that the SUPERVISOR_ID is valid.
       If no row is found, roll back.

If the flag for DEPARTMENT_CODE is set to "invalid":
       SET TRANSACTION READ WRITE
           RESERVING DEPARTMENTS FOR SHARED READ
```

(continued on next page)

Example 16–2 (Cont.) Updating a Table Containing Constraints

```
       Check the status parameter to ensure that the transaction
       started successfully; if not:
            Handle deadlock and lock conflict with timed retry of
            the SET TRANSACTION statement.
            Stop the program for unexpected errors.
       Select the row in the DEPARTMENTS table that has the input
       department code.
       Check the status parameter to ensure that the operation was
       successful; if not:
            Handle deadlock or lock conflict with ROLLBACK and
            timed retry of the statement.
            Stop the program for unexpected errors.

       If a row is found:
            Roll back.
            Set the predefined flag to indicate that DEPARTMENT_CODE
            is valid.
       If no row is found, roll back.

If the flag for JOB_CODE is set to "invalid":
       SET TRANSACTION READ WRITE RESERVING JOBS FOR SHARED READ
       Check the status parameter to ensure that the transaction
       started successfully; if not:
            Handle deadlock and lock conflict with timed retry of
            the SET TRANSACTION statement.
            Stop the program for unexpected errors.
       Select the row in the JOBS table that has the input job code
       value.
       Check the status parameter to ensure that the operation was
       successful; if not:
            Handle deadlock and lock conflict with ROLLBACK and
            timed retry of the statement.
            Stop the program for unexpected errors.

       If a row is found:
            Roll back.
            Set a predefined flag to indicate that JOB_CODE is
            valid.
       If no row is found, roll back.

If any flag indicates an input code or ID is still "invalid":
            Display messages that specify input values that failed
            verification checks.


UPDATE_JOB_HISTORY section:

SET TRANSACTION READ WRITE
               RESERVING EMPLOYEES FOR SHARED READ,
                         JOBS FOR SHARED READ,
                         DEPARTMENTS FOR SHARED READ,
                         JOB_HISTORY FOR SHARED WRITE
```

(continued on next page)

Example 16–2 (Cont.) Updating a Table Containing Constraints

```
Check the status parameter to ensure that the transaction started
successfullu; if not:
            Handle deadlock and lock conflict with timed retry of
            the SET TRANSACTION statement.
            Stop the program for unexpected errors.

Update the current JOB_HISTORY row where EMPLOYEE_ID is equal to
       the input value for EMPLOYEE_ID and where JOB_END is null,
       setting JOB_END equal to the value input for JOB_START.


Check the status parameter to ensure that the operation was successful
       or a row was not found (an acceptable condition for newly hired
       employees); if not:
            Handle lock conflict and deadlock with ROLLBACK and timed
            retry of the statement.
            Roll back and stop the program for unexpected errors.

Insert into the JOB_HISTORY table a new row using all input values
       and setting JOB_END to null.

Check the status parameter to ensure that the operation was successful;
       if not:
            Handle lock conflict and deadlock with ROLLBACK and timed
            retry of the statement.
            Roll back and stop the program for unexpected errors.
COMMIT
Check the status parameter to ensure that the COMMIT was successful;
       if not:
            Display a message saying an unexpected error occurred.
            Roll back and stop the program.
```

Example 16–2 shows that for the multiple-transaction strategy to work with minimal risk of the interactive user waiting too long for a lock to be released by other users, all transactions that need access to the same set of tables should access those tables in the shared share mode and keep the access time as brief as possible. The strategy also assumes that there is only a remote possibility that anyone would delete values after the program has checked them to make sure they exist. (Something is obviously wrong if a data entry clerk is trying to file new job information for an employee and code values suddenly become obsolete.)

The same strategy may not apply to your application given the kind of information in the database, number of concurrent users, and expected data access patterns at your site. For example, it may be appropriate to force users to start again from the beginning when an intertable constraint violation occurs. Upgrading the table share mode to protected may also be appropriate when a reduction in the number of deadlock encounters is worth the cost of an increase in wait time. Furthermore, program logic cannot solve all

problems. For example, if users who wish to update data are competing for the same tables with the more lengthy read/write transactions of report-writing programs, wait time may be unacceptably long because large amounts of data are not accessible for update. In such cases, your program may need the support of a database operations schedule that ensures interactive users encounter less interference from long read/write transactions.

16.7 Deciding When to Evaluate Constraints

If constraints are defined for tables in a database, all transactions that update those tables must evaluate the constraints. You cannot completely disable constraint evaluation for some update transactions and enable constraint evaluation again for other transactions. However, Oracle Rdb allows users to specify when constraints are evaluated:

- At verb time, which means all applicable constraints are evaluated for each row that is being written to the database file when the write takes place
- Immediately, which means constraints are evaluated at the end of the SQL statement that caused the constraint to be evaluated
- At commit time, which means applicable constraints are checked only once for all write operations applied to the database file during an entire transaction

Section 16.7.1 describes how to specify different constraint evaluation times.

16.7.1 Specifying Constraint Evaluation Time

You can specify when to evaluate constraints in the following ways:

- Define a default evaluation time when you create a table.
- Specify either verb time or commit time evaluation in the EVALUATE clause of the DECLARE TRANSACTION and SET TRANSACTION statements.

In addition, you can specify when commit-time constraints are actually evaluated using the following methods:

- Specify the /SQLOPTIONS=(CONSTRAINT=IMMEDIATE | DEFERRED) or `–s –'cm immediate|deferred'` command line qualifier when using the SQL precompiler

  The SQLOPTIONS=(CONSTRAINT=IMMEDIATE) or or `–s –'cm immediate'` qualifier causes each of the affected constraints to be evaluated immediately, as well as at the end of each statement, until the SET ALL CONSTRAINTS DEFERRED statement is issued or until the transaction completes with a commit or rollback operation.

- Specify the CONSTRAINT=IMMEDIATE | DEFERRED or –cm immediate|deferred command line qualifier when using the SQL module processor

  The CONSTRAINT=IMMEDIATE or –cm immediate qualifier causes each of the affected constraints to be evaluated immediately, as well as at the end of each statement, until the SET ALL CONSTRAINTS DEFERRED statement is issued or until the transaction completes with a commit or rollback operation.

- Specify the SET ALL CONSTRAINTS statement

  The SET ALL CONSTRAINTS IMMEDIATE statement causes all commit-time constraints to be evaluated immediately, as well as at the end of each statement and at commit time, until the SET ALL CONSTRAINTS DEFERRED statement is issued or until the transaction completes.

The following list describes when constraints are evaluated and the behavior of the constraint at evaluation time:

- A NOT DEFERRABLE constraint is *always* evaluated at the end of an SQL statement, regardless of what you specify with SET DEFAULT CONSTRAINT MODE statement or the SET ALL CONSTRAINTS statement. If you try to use the EVALUATING clause of the SET TRANSACTION statement to evaluate a NOT DEFERRABLE constraint at commit time, Oracle Rdb raises an exception.
- A DEFERRABLE constraint can be evaluated at any point up to the COMMIT statement, unless constraint mode is ON or you have specified EVALUATE AT VERB TIME with the SET TRANSACTION statement.
- When a NOT DEFERRABLE constraint or a DEFERRABLE constraint that is being evaluated at verb time is violated during a transaction, Oracle Rdb allows you to issue a COMMIT statement. The COMMIT statement stores the rows that did not violate the constraint. Rows that violated the constraint are not stored. A DEFERRABLE constraint that is evaluated at verb time has the behavior of a NOT DEFERRABLE constraint.
- When a DEFERRABLE constraint that is evaluated at commit time is violated, Oracle Rdb does not allow you to issue a COMMIT statement until you find the records that violated the constraint and specify values that do not violate the constraint. If you cannot find and fix the record that violated the constraint, you must roll back all modifications attempted by the transaction, including any updates that did not violate the constraint.

- A COMMIT statement can produce a constraint violation on execution only when commit-time evaluation of constraints is in effect. If you are designing statement-specific error-handling sections, the error handler for COMMIT does not need to take constraint violation into account. If you pass control to the same error-handling section whether an error is returned on an SQL statement writing to the database or on a COMMIT statement, write a generic error-handling section that monitors for and handles all expected errors.

  If constraints are evaluated at commit time, Oracle Rdb returns an error at the first violation, no matter how many rows it has processed. If you cannot find and fix the row that violated the constraint, you must roll back the entire transaction, including any updates that did not violate the constraint.

If the dialect is set to SQLV40, the default constraint attribute is DEFERRABLE. However, you receive a deprecated feature message if you do not specify a constraint attribute. If the dialect is set to SQL92, the default constraint attribute is NOT DEFERRABLE.

16.7.2 Recommendations for When to Evaluate Constraints

In general, for transactions that perform many update operations, Oracle Rdb recommends that you use constraints that are evaluated at verb time because:

- Any updates that violate a constraint immediately receive an error message, making it easier to find and fix records that violated the constraint.

  In contrast, Oracle Rdb generates one error message when one or more records violate a constraint that is evaluated at commit time. In this case, it is more difficult to find the particular record or records that violated the constraint.

- If only a small percentage of the records being updated violate a constraint, you can still commit all the other records that did not violate any constraints.

  In contrast, you must roll back all the records modified by an update transaction if even one record violates a constraint that is evaluated at commit time (unless you can find and fix the records that violated the constraint).

It is also advantageous to evaluate constraints at verb time when the modifications being performed by a transaction can use the verb-time context (foreign keys, for example) to directly look up records in referenced tables. This context is not available at commit time; therefore, Oracle Rdb may have to scan each row in the referenced table. As a result, even when only a small

number of updates are made, evaluating the constraint at verb time can take considerably less time than evaluating it at commit time.

Oracle Rdb recommends that you evaluate constraints at commit time in the following cases:

- When a constraint may cause every record in two or more tables to be checked. For example, suppose a constraint depends on this relationship:

  ```
  MIN (A.NUMBER) > MAX (B.NUMBER)
  ```

  To calculate the minimum value, Oracle Rdb may need to read every record in table A and table B.

- When you update one or more records several times during a transaction.
- When there are cross-dependencies (A depends on B, and B depends on A) that make it impossible to successfully evaluate the constraint at verb time. In this case, only a commit-time constraint works.

If the database evaluates constraints at commit time, but you want the application to comply with the ANSI/ISO SQL standard without changing the source code, you can use the precompiler or module processor command line qualifiers. With these qualifiers, you can specify that the commit-time constraints be evaluated at the end of each statement.

If you use the precompiler or module processor command line qualifiers to specify that the commit-time constraints be evaluated at the end of each statement, sometimes you may want to defer constraint checking so that you can insert data that may violate a constraint. In this situation, you can use the SET ALL CONSTRAINTS OFF statement.

16.8 Committing or Rolling Back a Transaction

When you end a transaction, you must commit or roll back the changes to the database. A COMMIT statement makes the changes to the database permanent; a ROLLBACK statement undoes the database changes.

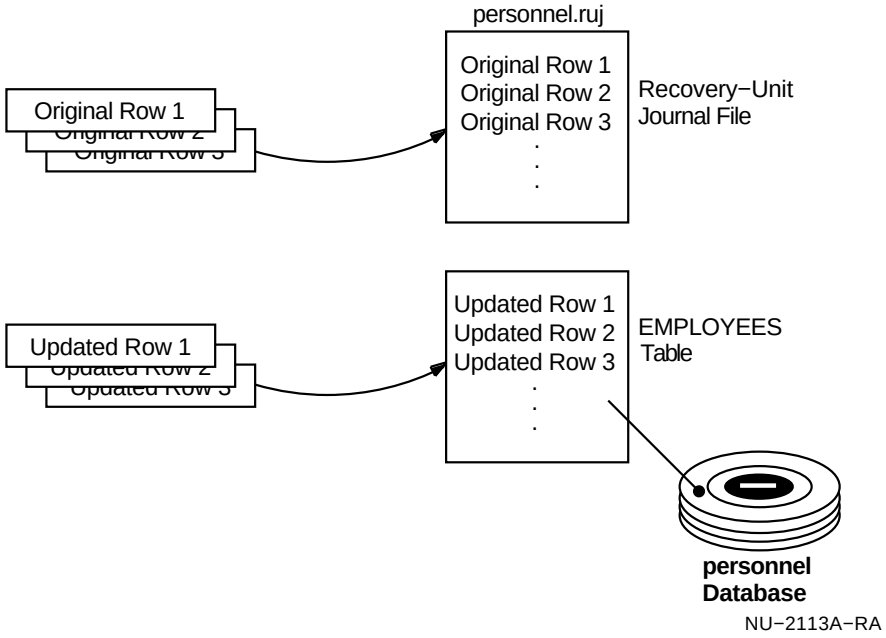
If your transaction involves only Oracle Rdb databases, you end a transaction with a COMMIT or ROLLBACK statement. If you called a transaction manager explicitly to start a distributed transaction, you must call the transaction manager directly to end the transaction. If you do not, the COMMIT or ROLLBACK statements return errors.

Assume that you declare the alias for the sample personnel database and you use a SET TRANSACTION statement to start a read/write transaction reserving the EMPLOYEES table for write access. Within the transaction, you open a cursor and then fetch the rows from a cursor and update the rows.

To support a rollback operation, Oracle Rdb creates a **recovery-unit journal file** (file type .ruj) to which it writes previous versions of rows (or definitions) that are changed or deleted during a transaction. The recovery-unit journal file also notes which rows (or definitions) are added by a transaction. When a transaction is rolled back, Oracle Rdb uses information in the recovery-unit journal file to undo any changes you made to the database during the transaction.

As you update each row, Oracle Rdb writes the original version of the row to the recovery-unit journal file. After all the UPDATE statements execute, the EMPLOYEES table contains modified rows and the recovery-unit journal file contains the rows in their original forms. Figure 16–6 shows the effect of an update on a database.

Figure 16–6 Transaction Recovery-Unit Journal (.ruj) File During an Update Transaction



If you enter a COMMIT statement to make your changes to the EMPLOYEES table permanent, Oracle Rdb empties the recovery-unit journal file and makes it ready for further transactions.

If you enter a ROLLBACK statement to undo the changes you made to the EMPLOYEES table during the transaction, SQL uses the recovery-unit journal file to bring the database back to its original state before the transaction started. SQL replaces the changed rows in the EMPLOYEES table with the original versions stored in the recovery-unit journal file.

Because a transaction groups SQL statements into a unit, it lets you undo (roll back) database changes made by all statements in the unit if, for example, one statement should fail. Thus, a transaction lets you defer the action of making permanent (committing) database changes until you are sure that all statements in a unit have executed successfully and have done what you expect them to do.

In interactive SQL and in programs on OpenVMS, if you do not specify a ROLLBACK statement, SQL (by default) commits database changes when your program exits normally. SQL commits the changes whether or not all update operations in the transaction executed successfully. Therefore, it is especially important to monitor and handle normal errors in programs.

In interactive SQL, the completion status is the status of the last statement prior to the EXIT statement, unless a transaction is active. In that case, SQL attempts to commit the transaction and the completion status is the status of the attempted COMMIT statement.

Digital UNIX

On Digital UNIX, when a program exits from SQL, active transactions are rolled back by default. Active transactions cannot be committed, as they can on OpenVMS, because the termination status is unknown to Oracle Rdb. If you want the transactions committed, you must do so before exiting. ♦

SQL does, however, roll back transactions under abnormal conditions from which your program could not recover. SQL rolls back a transaction, for example:

- On OpenVMS, if a user presses Ctrl/Y and then enters the DCL STOP command to end program execution
- When a hardware failure occurs
- If a program image exits with a failure status

Batch-update transactions must be committed, not rolled back. A batch-update transaction writes changes only:

- To the database storage file or files
- To the after-image journal file (if enabled by an ALTER DATABASE statement)

A batch-update transaction does not write to snapshot or recovery-unit journal files, thus reducing disk I/O operations to an absolute minimum. However, that means the information necessary to roll back a transaction is *not available*. If a batch-update transaction fails, the database will be corrupt. Your only options are to rebuild the database entirely or restore it from a backup.

17

Managing Multiple Connections in Programs

This chapter describes how to use SQL connections interactively for querying, testing, and prototyping programs and how to use them in programs. View connections primarily as programming tools for developing static and dynamic SQL programs. **Connections** allow explicit attaches and detaches from Oracle Rdb databases, concurrent access to one or more databases with one set of SQL procedures, and simultaneous SQL transactions.

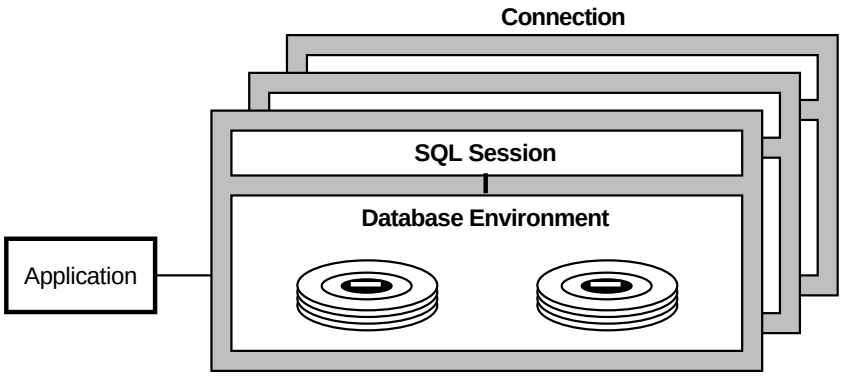
This chapter describes how to:

- Define the components of an SQL connection
- Create, switch between, and end connections
- Use transactions with connections
- Enable and disable connections in programs
- Use connections in applications

17.1 Introducing Connections

An SQL connection consists of two distinct but related pieces, an SQL session and a database environment. A connection associates a session with a database environment to form a matched pair that SQL associates with an application. Applications can have one or multiple connections, as Figure 17–1 shows.

Figure 17–1 Components of an SQL Connection



An SQL **session** is a series of SQL statements and their execution and resource context.

The **database environment** consists of all the databases to which a program (or interactive SQL) is attached. It specifies the set of Oracle Rdb databases that are attached or detached as a unit. Also, a connection is the set of all databases with unique aliases in the current connection.

17.1.1 Defining a Session

In the context of a connection, an SQL session refers both to the execution of a series of consecutive SQL statements and to the state of execution of those statements at any time. A series of SQL statements changes constantly over time. The context within which that change occurs includes:

- The state of all statements, queries, cursors, and dynamically prepared statements
- The resources taken by request handles and dynamically prepared statements

SQL creates a **default session** implicitly when a program executes its first SQL statement. The default session identifies the statement and resource context for all database attaches in the default database environment (see Section 17.1.2). The default session and the default database environment are matching components in the default connection (see Section 17.1.3).

In interactive SQL, a session consists of a series of executed SQL statements. Connections expand the meaning to include SQL statement context. This enhanced view of sessions has particular importance for connections.

Because a single application can include multiple connections and can switch between connections as necessary, connections require SQL to suspend the SQL statement context of a connection before switching to another. Suspending SQL statement execution lets your program switch to another connection and back, allowing it to take up execution where it left off. An analogy helps to clarify this point.

Imagine an application with multiple connections to be like a group of videocassette recorders that you pause and unpause to view video tapes in various stages of completion. Each pause suspends action until you view the tape again. Just as pausing a tape lets you resume viewing where you left off, suspending SQL statement context allows your programs (or interactive SQL) to switch between connections, then return to a session and continue processing in that session at the exact point the SQL statement context paused.

An SQL session can exist in either of two states: dormant (pause) or current (unpause):

- Dormant SQL sessions are the SQL sessions associated with an SQL environment that is not the current SQL environment.
- The current SQL session is the SQL session associated with the current SQL database environment.

SQL automatically handles connections for you. You need to know only that each connection possesses one session and that a session can be either current (currently active) or dormant (currently inactive).

17.1.2 Defining a Database Environment

A valid connection requires that a session have a matching database environment. While a session identifies the SQL statement execution and resource context for a connection, a **database environment** identifies one or more databases that are:

- Associated with a particular session
- Attached or detached as a unit
- Assigned unique aliases in the current connection

In embedded and module language programs, an application possesses at least one **default database environment**. All databases declared at compilation time with the DECLARE ALIAS statement constitute the default environment for an application.

The code fragment in Example 17–1 shows the default database environment created by declaring two aliases using embedded SQL.

Example 17–1 Declaring Databases for the Default Database Environment in Embedded SQL

```
EXEC SQL DECLARE ALIAS FILENAME personnel;
EXEC SQL DECLARE ALIAS_CORP ALIAS FILENAME corporate;
```

The default database environment for this program consists of the two databases, personnel and corporate. SQL permits a program to declare more than one alias but permits only one of those aliases to assume the default. When you omit an alias in the DECLARE ALIAS statement, SQL uses the default alias name RDB$DBHANDLE. The DECLARE ALIAS statement is valid only in precompiled SQL and SQL module language, not in interactive SQL or as a dynamically executable SQL statement.

The DECLARE ALIAS statement specifies the name and the source of the database definitions to be used for module compilation and precompilation and makes the named alias part of the default database environment of the applications. It is a nonexecutable statement that declares the database to the program at compilation. SQL does not attach to the database until it executes the first executable SQL statement in the program or SQL module.

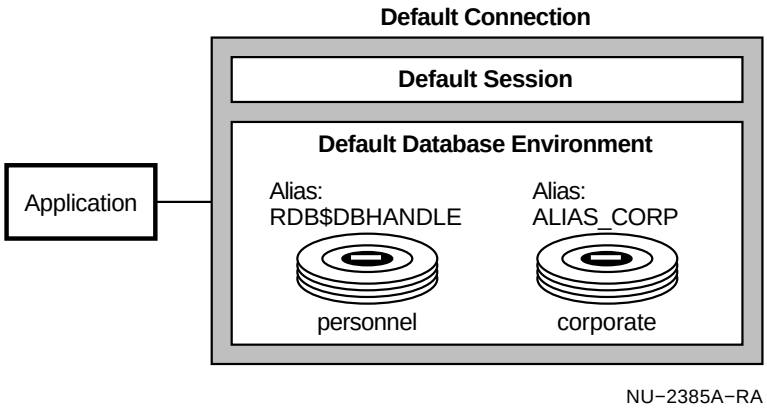
17.1.3 Defining a Connection

A connection associates a session with a database environment and the matched session-environment pair with the application. An application has at least one connection, called the **default connection**, but multiple connections are possible, depending on application need. The default connection contains the:

- Default database environment, which includes all databases declared by a program
- Default session, which identifies all of the SQL statement context and resource context for the default database environment

The default connection for the application in Figure 17–2 contains the default session and two declared databases in the default database environment.

Figure 17–2 Default Connection



The personnel database uses the default RDB$DBHANDLE alias, and the corporate database uses the user-defined alias ALIAS_CORP.

When a program invokes a procedure in a module without explicitly specifying a connection, SQL establishes a default connection and connects it to the default SQL session and default database environment. SQL declares all databases simultaneously and attaches to each of them when the first executable SQL statement in the program or SQL module is executed.

You can create an explicit connection other than the default with the CONNECT statement, as described in Section 17.2.

17.2 Creating, Switching Between, and Ending Connections

The SQL interface provides three statements to help you control connections in your programs. Table 17–1 describes the CONNECT, SET CONNECT, and DISCONNECT statements that let you create, switch between, and end connections, respectively.

Table 17–1 SQL Statements Affecting Connections

SQL Statement	Description
CONNECT	Creates a session and a database environment pair and associates the connection pair with an application. Gives the connection a name that the SET CONNECT statement uses to change connections and the DISCONNECT statement uses to release the named connection.
SET CONNECT	Selects a connection from the available connections, thereby switching the connection. Changes the default database environment for subsequent SQL statements.
DISCONNECT	Detaches from declared databases and releases the aliases that you specified in program declarations.

Section 17.2.1 discusses creating connections. Section 17.2.5 discusses switching connections. Section 17.2.6 discusses ending connections.

17.2.1 Creating Connections

When a program executes a CONNECT statement, SQL creates a connection, attaches to the database environment, and gives the association the name specified in the CONNECT statement. SQL also performs an implicit SET CONNECT statement to establish the connection, which makes execution of an explicit SET CONNECT statement after a CONNECT statement unnecessary.

Section 17.2.2, Section 17.2.3, and Section 17.2.4 show examples of how to use the CONNECT statement in your programs. Section 17.2.5 describes when and how to make an explicit connection.

17.2.2 Duplicating the Default Database Environment

The CONNECT statement lets you duplicate in another connection all aliases in the default connection. You can do this explicitly, with the DEFAULT argument and the AS clause, or with the AS clause only. You can duplicate the default database environment using any of three CONNECT statements.

- Specify explicitly all aliases in the default connection.

  This use of the CONNECT statement to duplicate the default database environment is the most laborious of the three techniques for duplicating the default connection. You can save typing time by using either of the other two techniques; however, explicit specification of the aliases in the CONNECT statement does demonstrate how you might duplicate a portion of the aliases in the default connection by simply including the aliases that you need in the new connection.

A statement that explicitly specifies all aliases might look like this:

```
CONNECT TO 'ALIAS RDB$DBHANDLE, ALIAS ALIAS_CORP' AS 'OTHER';
```

- Use the DEFAULT argument with the AS clause.

  You might prefer to show more clearly (by including the DEFAULT argument) that you want to duplicate the aliases in the default connection, like this:

  ```
  CONNECT TO 'DEFAULT' AS 'OTHER';
  ```

- Use the AS clause only.

  When you use the CONNECT statement with only the AS clause and a name, it might look like this:

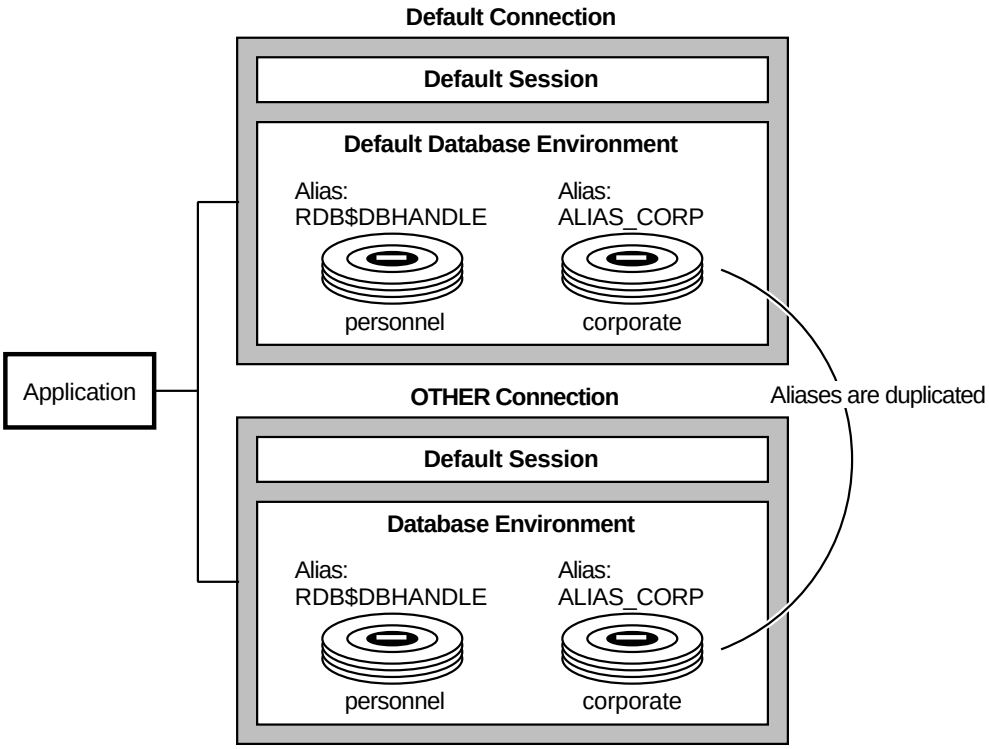
  ```
  CONNECT AS 'OTHER';
  ```

  SQL duplicates the aliases in the default connection with the specified name. This is the easiest way to duplicate the default database environment.

The database environment created by any of these CONNECT statements is shown in Figure 17–3.

New connections (such as OTHER created in the previous examples) can duplicate aliases only from the default connection, not from any previously created connections.

Figure 17–3 Duplicating the Default Connection



17.2.3 Specifying Different Databases for the Same Aliases

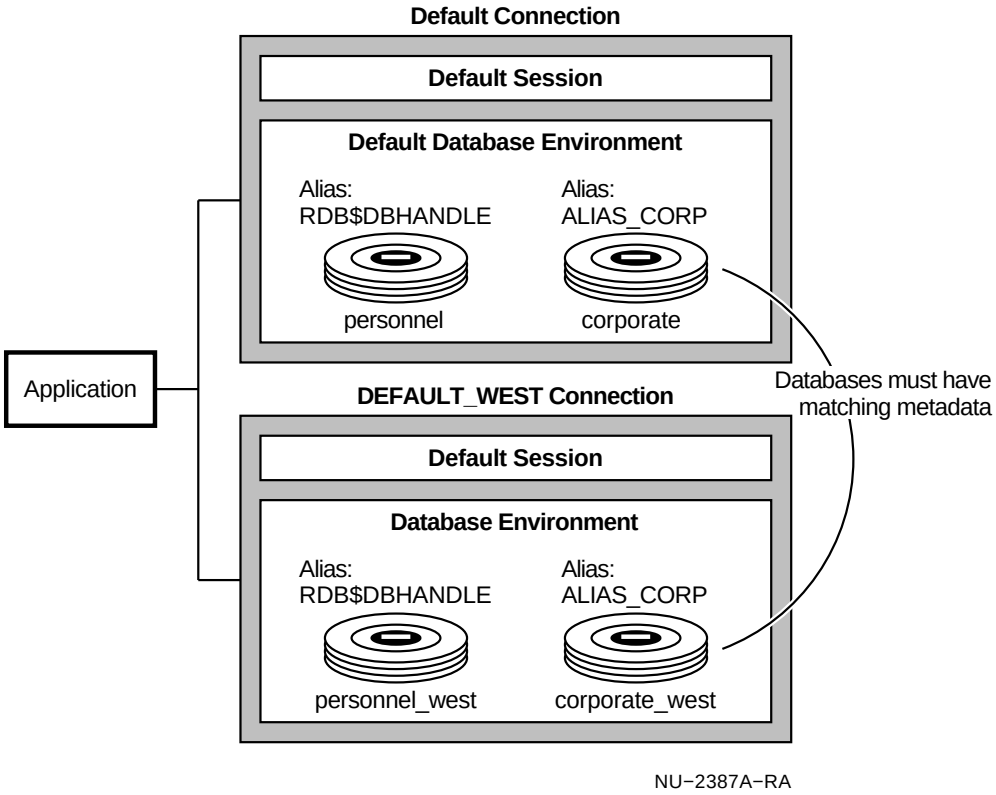
You might find it necessary to have two connections that contain different databases but the same aliases. The following example shows how to do this:

```
CONNECT TO 'FILENAME personnel_west,
           ALIAS ALIAS_CORP FILENAME corporate_west' AS 'DEFAULT_WEST';
```

Databases sharing aliases in different connections must contain the same metadata. Thus, the personnel and personnel_west databases must have matching metadata, as must the corporate and corporate_west databases.

The database environment created by this CONNECT statement is shown in Figure 17–4.

Figure 17–4 Specifying Different Databases for the Same Aliases

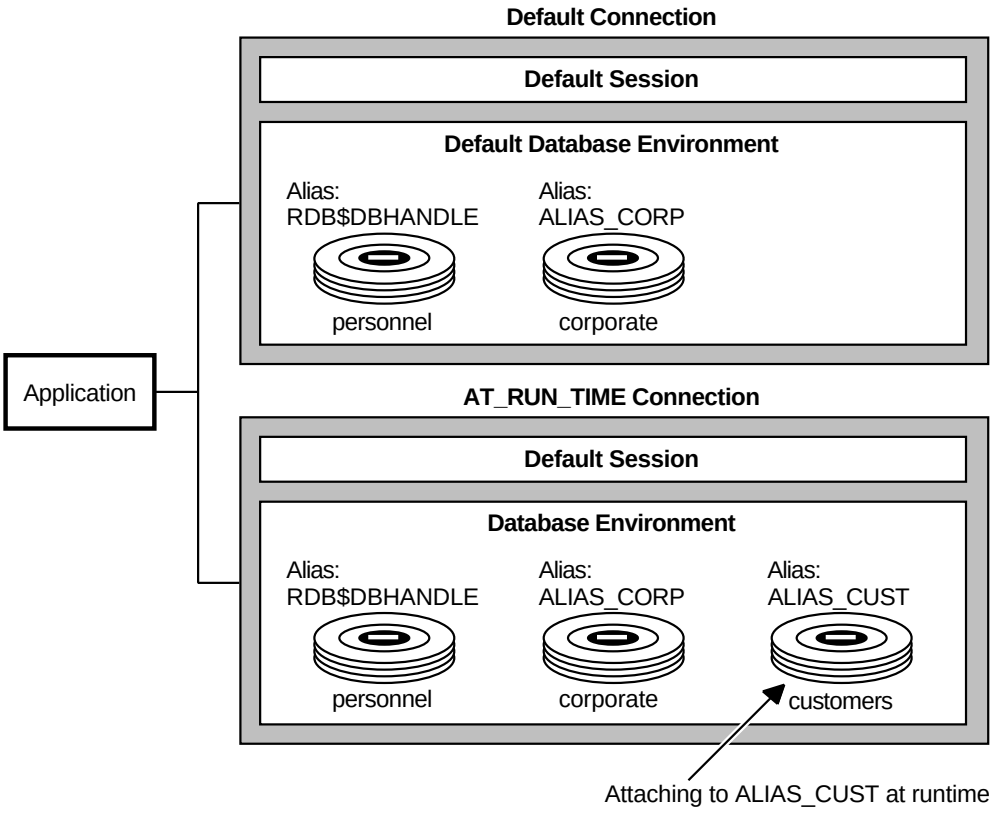


17.2.4 Specifying an Additional Run-Time Attach

Figure 17–5 shows the database environment created by using the CONNECT and ATTACH statements to attach at run time. In this case, you include the ATTACH statement in a CONNECT statement, as shown in the following example:

```
CONNECT TO 'ALIAS RDB$DBHANDLE, ALIAS ALIAS_CORP,
            ATTACH ALIAS ALIAS_CUST FILENAME customers' AS 'AT_RUN_TIME';
```

Figure 17–5 Specifying an Additional Run-Time Attach



NU–2388A–RA

17.2.5 Switching Between Connections

Once you create a connection with the CONNECT statement, your program must have a way to connect to it. In SQL embedded and module language programs, you can use either of two forms of the SET CONNECT statement to accomplish this:

- SET CONNECT connection-name

  Connects your program to the named connection from the connection names available to the program.

- SET CONNECT DEFAULT

Connects your program to the default connection, which includes the default session and the default database environment consisting of all databases declared by the program at compile time.

The SET CONNECT statement suspends the current connection and switches to the connection specified in the statement. After switching connections, your program executes subsequent SQL statements in the context of that connection.

You can issue another SET CONNECT statement to return to the first connection and resume processing.

17.2.6 Ending Connections

When a program no longer needs a connection, issue the DISCONNECT statement. The DISCONNECT statement:

- Deletes one or more connections
- Closes the session associated with each connection
- Frees all the resources associated with each connection
- Releases all the compiled requests and prepared statements
- Detaches all databases
- Releases the memory for session contexts

The DISCONNECT statement provides the flexibility to disconnect some or all the connections, using the following forms:

- DISCONNECT ALL ends all available connections.
- DISCONNECT DEFAULT ends the default connection only.
- DISCONNECT CURRENT ends your program's current connection.
- DISCONNECT connection-name ends the named connection.

When a program exits without an explicit connection disconnect, SQL:

- Commits all transactions in the default connection and executes an implicit DISCONNECT ALL statement
- Rolls back all other transactions available to the same program and executes an implicit DISCONNECT ALL statement

For compliance with the ANSI/ISO SQL standard, use the explicit DISCONNECT ALL statement, which rolls back all transactions. To disconnect the current connection, use COMMIT and the DISCONNECT CURRENT statements.

17.3 Using Transactions with Connections

Every session has its own transaction, which SQL starts when it first enters a connection. You can change the characteristics of a transaction within a connection by using the SET TRANSACTION statement, as the SQL module in Example 17–2 shows. Because a database cannot be attached while a transaction is active, connections allow you to have multiple transactions active simultaneously without having to commit or roll back a transaction.

17.4 Enabling and Disabling Connections in Programs

You must enable SQL connections before using them in SQL modules or in precompiled programs. SQL disables connections by default. This section describes the process of enabling and disabling connections when using the SQL module processor and the SQL precompiler.

17.4.1 Enabling and Disabling Connections for Module Programming

To enable connections for the SQL module processor, compile your program, process the module with the CONNECT or the `-conn` qualifier, link the program and processed module, and then run it.

Connections are disabled by default. However, you can explicitly disable them by using the NOCONNECT or `-noconn` qualifier.

When you enable connections with the SQL module processor, do not combine in a single program the modules you compiled with connections enabled and the modules you compiled with connections disabled. Either all modules must be compiled with connections enabled, or all must be compiled without connections. You cannot mix the two.

17.4.2 Enabling and Disabling Connections for Precompiled Programs

To enable connections for the SQL precompiler, use the qualifier SQLOPTIONS=CONNECT or `-s ' -conn`, link your application, and then run it.

Connections are disabled by default. However, you can explicitly disable them by using the /SQLOPTIONS=NOCONNECT or `-s ' -noconn` qualifier.

17.5 Using Connections in an Application

This section contains a sample SQL module (Example 17–2) and its matching C program (Example 17–3) that show one use for connections. The program creates two connections, each containing separate attaches to the same database. In the first connection, a read-only transaction allows the program to open and increment with FETCH through a cursor. In the second connection, a read/write transaction allows the program to update the database.

Example 17–2 SQL Module Using Connections

```
-- SQL module that uses connections for two concurrent read/write
-- transactions.

MODULE CURSOR_LOOP
LANGUAGE C
AUTHORIZATION JONES
PARAMETER COLONS

DECLARE ALIAS FILENAME test
DECLARE A CURSOR FOR SELECT CITY FROM S
          WHERE CITY IS NOT NULL

-- Connection management routines.

-- Create an explicit connection that duplicates the default database
-- environment.
PROCEDURE CONNECT
     (SQLCODE,
      :CONNECT_2 CHAR(31));

     CONNECT AS :CONNECT_2;

-- Switch to the nondefault connection called CONNECT_2.

PROCEDURE SET_CONNECT
     (SQLCODE,
      :CONNECT_2  CHAR(31));

     SET CONNECT :CONNECT_2;

-- Switch to the default connection.
PROCEDURE SET_CONNECT_DEFAULT
     (SQLCODE);

     SET CONNECT DEFAULT;
```

(continued on next page)

Example 17–2 (Cont.) SQL Module Using Connections

```
-- Disconnect all available connections (CONNECT_2 and DEFAULT).
PROCEDURE DISCONNECT
     (SQLCODE);

     DISCONNECT ALL;

-- Open and fetch row information from cursor A in a read-only transaction.
PROCEDURE  OPEN_A
     (SQLCODE);

     OPEN A;

PROCEDURE FETCH_A
     (SQLCODE,
      :RET_CITY    CHAR(15));

     FETCH A INTO :RET_CITY;

PROCEDURE CLOSE_A
     (SQLCODE);

     CLOSE A;

-- Update rows that meet a certain program criteria.
PROCEDURE UPDATE_P
     (SQLCODE,
      :PCITY       CHAR(15));

     UPDATE MYP SET CITY_CODE = 'XXXXX' WHERE CITY = :PCITY;

-- Start a read-only transaction for fetching rows.
PROCEDURE RO_TXN
     (SQLCODE);

     SET TRANSACTION READ ONLY NOWAIT;

-- Start a read/write transaction for updating rows.
PROCEDURE RW_TXN
     (SQLCODE);

     SET TRANSACTION READ WRITE NOWAIT;
```

```
-- Commit the update transaction.
PROCEDURE COMMIT
     (SQLCODE);

     COMMIT;
```

Example 17–3 shows a program that uses the modules from Example 17–2.

Example 17–3 C Program Using Connections

```
   .
   .
   .
/* Create the read-only transaction in the default connection. SQL
implicitly creates a default connection when a program executes its
first SQL statement.*/
  RO_TXN( &sqlcode );
  if (sqlcode != SQLCODE_SUCCESS)
     goto err;

/* Create a second connection to handle update operations. */
  CONNECT( &sqlcode, updater );
  if (sqlcode != SQLCODE_SUCCESS)
    goto err;

/* Return to the default connection to read rows. */
  SET_CONNECT_DEFAULT( &sqlcode );
  if (sqlcode != SQLCODE_SUCCESS)
      goto err;

/* Open a cursor and read the first row. */

  OPEN_A( &sqlcode );
  if (sqlcode != SQLCODE_SUCCESS)
      goto err;

  FETCH_A( &sqlcode, city );
    while (sqlcode == SQLCODE_SUCCESS) {

/* Test whether or not values in the row meet program criteria.  If criteria
is not met, the program loops back to read another row;  otherwise it
continues. */

    if (strcmp( city, "London         ") == 0) {

          /* Switch to the nondefault connection.  */
          SET_CONNECT( &sqlcode, updater );
          if (sqlcode != SQLCODE_SUCCESS)
             goto err;
```

(continued on next page)

Example 17–3 (Cont.) C Program Using Connections

```
          /* Create the read/write transaction for an update operation. */
          RW_TXN( &sqlcode );
          if (sqlcode != SQLCODE_SUCCESS)
             goto err;

          /* Update the rows that meet the criteria. */
          UPDATE_P( &sqlcode, city );
          if (sqlcode != SQLCODE_SUCCESS)
              goto err;

          /* Commit the change. */
          COMMIT( &sqlcode );
          if (sqlcode != SQLCODE_SUCCESS)
              goto err;

          /* Return to the default connection to read rows. */
          SET_CONNECT_DEFAULT( &sqlcode );
          if (sqlcode != SQLCODE_SUCCESS)
              goto err;
      }
  /* Read another row. */
  FETCH_A( &sqlcode, city );
  }
/* Close cursor A. */
  CLOSE_A( &sqlcode );
  if (sqlcode != SQLCODE_SUCCESS)
      goto err;
  return;

/* Error handling. */
  err:
      sql_signal();
    /* Disconnect all active connections. */
    DISCONNECT(&sqlcode);
      EXIT(0);
  }
```

Connections allow this program to have two active database attaches, two concurrent transactions, and concurrent access to both attaches.

Part VI

Data Manipulation in Programs

This part discusses:

- How to declare and use cursors
- How to load and update data from your SQL programs
- How to manipulate data in multischema databases

18
Using Cursors

This chapter defines SQL cursors and describes the SQL cursor classes and types. The sections that follow explain:

- The concepts related to cursors
- The different categories of cursors
- How to control the opening and closing of cursors
- How to use table cursors
- How to use holdable table cursors
- How to use list cursors
- How to use scrollable list cursors
- How to use dynamic cursors
- How to use extended dynamic cursors

Reference Reading

Chapter 11 shows how to use dynamic and extended dynamic cursors to process SELECT statements at run time. The *Oracle Rdb7 SQL Reference Manual* includes detailed information about how cursors work.

18.1 Introduction to Cursors

In most cases, when you use SQL to retrieve data, SQL retrieves multiple rows as a result of the select expression. However, application programs cannot process multiple rows together but rather must process the rows one row at a time. Thus, unless SQL retrieves only one row (a singleton select), you must use cursors in programs in order to read, insert, or update data in a database.

A **cursor** lets you execute a query and process multiple rows one row at a time. When you use a cursor, you create:

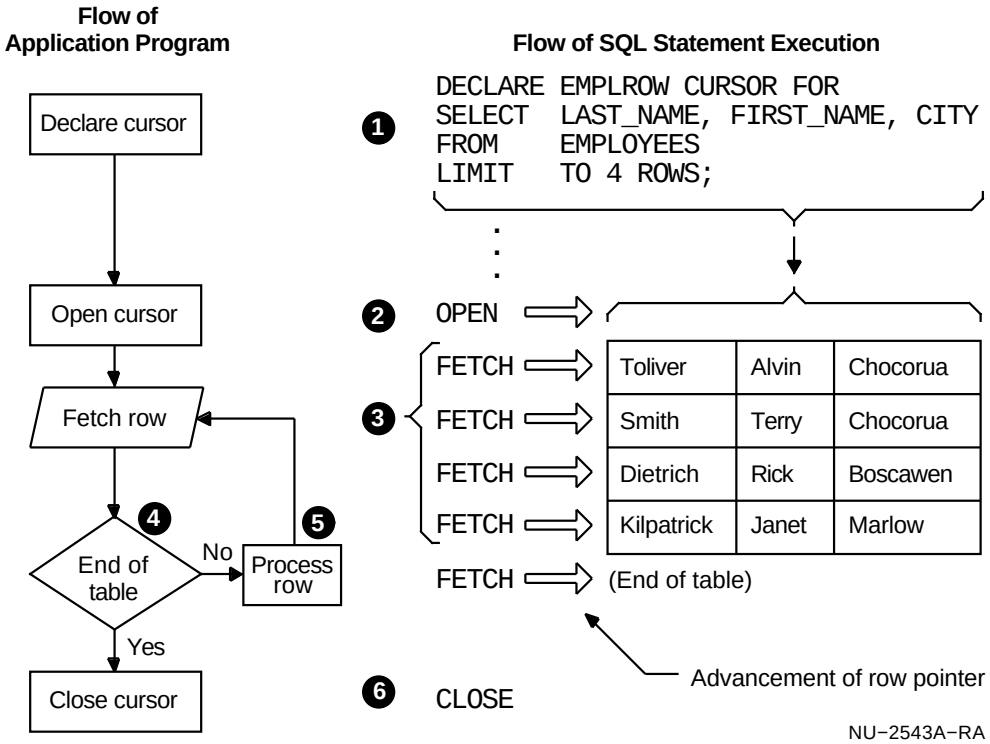
- An ordered **result table**, which is a temporary collection of columns and rows from one or more tables or views.
- A **row pointer**, which determines either the next row to be fetched or the end of the result table.

Cursors are modeled after the general methods used to process records from a data file. To process multiple rows, your program must perform a sequence of basic cursor operations—declaring the cursor, opening the cursor, fetching rows, and closing the cursor.

18.1.1 How Cursors Work

Basic cursor operations include declaring a cursor, opening a cursor, fetching rows, and closing the cursor to delete the result table. Figure 18–1 shows this basic algorithm and the steps that you take to retrieve multiple rows using a table cursor.

Figure 18–1 How Cursors and Related Statements Work Together



To use a cursor, take the following steps, which are keyed to Figure 18–1:

❶ Declare the cursor

You use the DECLARE CURSOR statement to provide a name for the result table and to specify its table sources, rows, and columns. The cursor declaration consists of:

- The name of the cursor, in this case, EMPLROW.
- The select expression that produces the result table, in this case, a query that generates a result table with three columns and four rows.

The following example shows how to declare the cursor EMPLROW:

```
DECLARE EMPLROW CURSOR FOR
    SELECT LAST_NAME, FIRST_NAME, CITY FROM EMPLOYEES
```

❷ Open the cursor

Opening a cursor executes the query, creates the result table associated with the cursor declaration, and sets the row pointer to a position before the first row in the result table.

When you execute the OPEN statement (not when you declare the cursor), the result table becomes available for further processing. The following example shows how to open the cursor:

```
OPEN EMPLROW;
```

The result table exists until you close the cursor.

Opening a cursor after it has been closed positions the next fetch data retrieval operation back to the first row in the result table. If a cursor is already open, SQL returns an error and the next FETCH statement retrieves the next row.

❸ Fetch a row

The FETCH statement retrieves the row pointed to by the row pointer (also called the **current row**), then advances the row pointer to the next row in the result table. If the row pointer is positioned before the first row in the result table, SQL retrieves the first row. If the row pointer is positioned after the last row of the result table, SQL signals that all rows have been fetched.

Therefore, the row pointer is either before the first row, on a certain row, or after the last row. A row pointer may be before the first row or after the last row of a table even if the result table is empty.

The following example shows how to fetch a row of the EMPLROW cursor using interactive SQL:

```
FETCH EMPLROW;
```

When you use SQL module language or precompiled SQL, you use the INTO clause to store the data values in program parameters. The following example shows how to use the FETCH statement to store the values from the EMPLROW cursor in parameters in an SQL module language procedure:

```
FETCH EMPLROW INTO :L_NAME_P, :F_NAME_P, CITY_P;
```

Each time the FETCH statement executes, the pointer moves to a new row in the result table. If the result table contains five rows, five FETCH statements must execute to retrieve all rows. In an SQL program, you typically execute a loop containing a FETCH statement to process all rows.

SQL provides considerable flexibility in fetching rows. You can retrieve rows in the order in which they occur in the result table. In addition, when you use a list cursor, you can retrieve rows at random using a scrollable list cursor. For more information about scrollable list cursors, see Section 18.7.

If you do not declare a scrollable list cursor, you must use the FETCH statement to retrieve rows only in the order in which they occur. You cannot skip rows or go backwards. To retrieve a row that was fetched earlier, you must close the cursor, reopen it, create a *new* result table, and then fetch rows until you get the one you want.

If you attempt to fetch a row after you close the cursor, the fetch operation fails because the result table for the cursor declaration no longer exists.

❹ Test for the condition "No data"

As in data file processing, you need to know when you are at the end of your result table so you can finish processing in a clean manner. Check the SQLCODE or SQLSTATE status parameter to detect that there is no data.

❺ Process the row

If there is data, the currently fetched row is available to your program. You can use the retrieved data in your program, modify the row using the positioned UPDATE statement, or remove the row using the positioned DELETE statement.

When you successfully execute a positioned DELETE statement in a cursor operation, the row pointer is positioned on the next row. If the deleted row is the last row in the result table, then the row pointer is positioned after the last row.

❻ Close the cursor

Closing a cursor makes the result table unavailable for further processing and deletes the result table associated with the cursor declaration. You can open and close cursors many times. Reopening a cursor after it has been closed executes the query again and positions the row pointer before the first row in the *new* result table. The following example shows how to close the cursor:

```
CLOSE EMPLROW;
```

To close all open cursors, you can use the sql_close_cursors( ) routine.

See Section 18.3 for more information about controlling the opening and closing of cursors.

Even after you close a cursor, the cursor declaration remains in effect throughout a connection within a program execution cycle. After you disconnect from the connection (using the DISCONNECT statement or attaching to another database using the same alias for the database), the cursor declaration is no longer in effect.

Because opening a cursor creates a new result table, if you open the same cursor in different transactions, the result table may reflect updates to the database made by other procedures. Refer to Chapter 16 for information on maintaining a consistent state of the data in a transaction.

When an application program opens a cursor, Oracle Rdb checks that the user has the appropriate privileges to open the particular type of cursor (such as an update cursor) and that the user has all privileges necessary to execute the request. That is, if the application uses a cursor to insert, update, or delete rows in the same module as the OPEN statement, the user must have appropriate privileges to perform those actions, even if the application may not perform those actions during runtime.

Reference Reading

The FETCH statement updates row count information in the SQL Communications Area (SQLCA). After successful execution of the FETCH statement, the third element of the SQLERRD array in the SQLCA indicates the ordinal position in the cursor of the row retrieved by the statement. For more information, read the *Oracle Rdb7 SQL Reference Manual* section about the SHOW SQLCA statement and the *Oracle Rdb7 SQL Reference Manual* appendix about the SQLCA.

18.1.2 Comparing Cursors and Views

A cursor is a logical construct similar to a view. Like a view, a cursor creates a temporary collection of columns and rows from one or more tables or views. This collection of columns and rows is called a result table and is created dynamically.

The result table associated with a view is not created until a statement accesses the view. The result table associated with a cursor is not created until you open the cursor.

Unlike when you refer to views in a SELECT, INSERT, or UPDATE statement, you can vary the length of time that a cursor's result table exists. The table created for a view does not exist longer than it takes to execute the statement that refers to the view; however, the table created for a cursor exists until you close the cursor implicitly or explicitly.

A cursor is a logical entity, not a specific result table. However, to simplify text, the term "cursor" is often used to refer to a result table created by an OPEN statement. Keep this fact in mind when you read other SQL manuals and other chapters in this book.

18.1.3 Deciding When a Cursor Is Needed

In a program, you must use a cursor when the result table contains multiple rows. There are a few cases where this is not necessary. For example:

- You use a value in a program variable to determine a condition for deleting multiple rows, and then execute a DELETE statement that erases all those rows.
- You use values in program variables to determine a condition for selecting rows for update, and to change all occurrences of specified columns to the same value.

You do not have to use cursors when you want to perform multiple operations on a single row. For single-row processing, it is simpler to use a **singleton select**, a SELECT statement that includes an INTO clause to retrieve the row from the database and store its values in program variables. In fact, you can use the INTO clause in SELECT statements *only* for program retrieval of single rows. The following are singleton select statements in an SQL module:

```
SELECT EMPLOYEE_ID INTO :ID_NUMBER:ID_NUMBER_IND
         FROM DBH1.EMPLOYEES
         WHERE EMPLOYEE_ID = :INPUT_ID

SELECT LAST_NAME, FIRST_NAME INTO :FULL_NAME
         FROM DBH1.EMPLOYEES
         WHERE EMPLOYEE_ID = :INPUT_ID
```

If your program retrieves or stores data in columns that might contain null values, you must use indicator parameters, as shown in the first of the preceding two examples. Refer to Chapter 8 for a discussion on when and how to specify indicator parameters.

In most cases, however, you need to work with multiple rows one at a time, by retrieving each row's values into program parameters. Doing this requires a cursor declaration and a statement to create a result table for the cursor.

To see how cursors work in a complete program, refer to the precompiled program sql_report in the samples directory.

18.2 Understanding the Different Categories of Cursors

SQL provides different types of cursors, different classes of cursors, and different modes of cursors.

With SQL, you can create two different types of cursors:

- Table cursors

  Table cursors provide a method to access individual rows of a result table. Many of the examples in this chapter, including Figure 18–1, illustrate the use of table cursors. Section 18.4 explains how to work with table cursors.

- List cursors

  List cursors provide a method for accessing individual elements in a list. A **list** is an ordered collection of elements, also called segments. A list is a LIST OF BYTE VARYING data type and each segment of the list is a BYTE VARYING data type. List cursors enable you to manipulate objects that are too large to be stored in CHAR or VARCHAR fields. (Such fields often contain structured binary data.) Section 18.6 describes how to use list cursors.

The classes of cursors differentiate what information is specified at compile time. The three classes of cursors are:

- Static cursors, which this manual usually refers to using the general term of "cursor."

  If you explicitly specify both the cursor name and the SELECT statement in a DECLARE CURSOR statement, the cursor is a static cursor. In other words, the cursor name and the SELECT statement are known at compile time.

  Use static cursors in interactive SQL and in programs. Most of the examples in this chapter demonstrate the use of static cursors.

- Dynamic cursors

  If you supply the name of a prepared statement instead of a SELECT statement in the DECLARE CURSOR statement, the cursor is a dynamic cursor.

  Use dynamic cursors in dynamic SQL programs. You explicitly specify the cursor name at compile time; however, you do not explicitly specify the SELECT statement. In other words, the cursor name is known at compile time, but the SELECT statement is not known until run time. Section 18.8 describes how to use dynamic cursors.

- Extended dynamic cursors

If you supply parameters for both the cursor name and the SELECT statement in the DECLARE CURSOR statement, the cursor is an extended dynamic cursor. Neither the cursor name nor the SELECT statement is known until run time. Extended dynamic cursors let you use one set of cursor-related statements to process any number of dynamically generated statements. Section 18.9 explains how to use extended dynamic cursors.

SQL further divides cursors into the modes of operation that each type of cursor can perform:

- You can specify the following modes for table cursors:
  - Update

    You must use an update cursor if you want to modify a row in the result table. Update cursors are the default table cursors. With update cursors, Oracle Rdb first reads the rows and uses SHARED (or PROTECTED) READ access. When you update a row, Oracle Rdb uses EXCLUSIVE access.
  - Update-only

    When you use update-only cursors, Oracle Rdb uses a more aggressive locking model, locking the rows for EXCLUSIVE access when it first reads the rows. As a result, update-only cursors may avoid deadlocks normally encountered during lock promotion. Use update-only cursors when you are updating all (or a high percentage of) fetched rows.
  - Read-only

    Use a read-only cursor when you do not want to update information in a result table.
  - Insert-only

    Use insert-only cursors when you want to position the cursor on a row that has just been inserted so that you can load lists into that row.
- You can specify the following modes for list cursors:
  - Read-only

    Use read-only cursors to read existing lists.
  - Insert-only

    Use insert-only cursors to insert data into a list.

  You cannot update data in a list.

18.3 Controlling the Opening and Closing of Cursors

Section 18.1.1 describes using the OPEN statement to open an cursor and the CLOSE statement to close a cursor. This section describes in more detail how you can control the opening and closing of cursors.

By default, the result table of a table or list cursor exists from the time an OPEN statement executes until you close the cursor. You can close a cursor in the following ways:

- Explicitly, by using the CLOSE statement
- Implicitly, using a COMMIT or ROLLBACK statement
- Implicitly, when a program stops execution
- Implicitly, when you exit from interactive SQL

When you close the cursor, SQL deletes the result table. If you reopen the cursor, you read a new result table containing any new or changed data, unless you use the REPEATABLE READ isolation level.

When you disconnect from a database, not only does SQL close the cursor, but it no longer recognizes the cursor declaration.

For table cursors, SQL provides additional control with the WITH HOLD clause of the DECLARE CURSOR statement, letting you keep cursors open across transactions. These cursors, called **holdable cursors**, remain open after the transaction ends and hold the position of the row pointer when a new transaction begins.

See Section 18.5 for a description of holdable cursors.

18.4 Using Table Cursors

Table cursors let you access individual rows of a result table, retrieving, updating, or deleting them one row at a time.

The statements you execute to process the rows typically include a combination of SQL and programming language statements. If the program is processing multiple rows in the result table, these statements execute in a loop. The first time the loop executes, the FETCH statement points to the first row in the result table and stores its values into one or more parameters specified by the program. Subsequent statements manipulate the parameter values, for example, to:

- Include them in calculations along with values supplied by the program
- Write them to a report

- Display them on a terminal screen

The next time the loop executes, the FETCH statement stores values of the next row into the program parameters and the program operations are repeated using the new set of values.

A program ends a row-processing loop when it detects that a FETCH statement caused a "no data" completion condition. The "no data" completion condition is returned by the database system as RDB$_STREAM_EOF. In interactive SQL, this error appears as "%RDB-E-STREAM_EOF, attempt to fetch past end of record stream." In a program, this error is returned as value 100 in the SQLCODE status parameter and value 02000 in the SQLSTATE status parameter. Chapter 10 describes how you handle completion exception conditions and errors in a program.

If a cursor is opened and its result table contains no rows, the *first* FETCH statement that executes (first iteration of the loop) causes the "no data" completion condition. When the "no data" completion condition occurs, there is no current row for the cursor and no values are transferred from columns to the program variables specified in the INTO clause of the FETCH statement.

Example 18–1 shows how to declare a table cursor that contains current job history data associated with the department code MBMF and how to open the cursor, fetch rows, and close the cursor.

Example 18–1 Using Table Cursors

```
SQL> DECLARE CURRENT_MBMF TABLE CURSOR FOR
cont>    SELECT JOB_CODE, EMPLOYEE_ID, JOB_START FROM JOB_HISTORY
cont>       WHERE (DEPARTMENT_CODE = 'MBMF') AND
cont>             (JOB_END IS NULL)
cont>       ORDER BY JOB_CODE, JOB_START;
SQL> --
SQL> -- The OPEN statement creates the result table.
SQL> --
SQL> OPEN CURRENT_MBMF;
SQL> --
SQL> -- The first FETCH statement retrieves the first row.
SQL> --
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 APGM       00174         22-Sep-1981
SQL> --
```

(continued on next page)

Example 18–1 (Cont.) Using Table Cursors

```
SQL> -- Each subsequent FETCH statement retrieves the
SQL> -- row positioned next in the result table.
SQL> --
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 ASCK       00165          8-Mar-1981
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 DMGR       00227         25-Nov-1981
SQL> --
SQL> -- The preceding row was the last row in the result table.
SQL> -- Another FETCH statement reveals this fact.
SQL> --
SQL> FETCH CURRENT_MBMF;
%RDB-E-STREAM_EOF, attempt to fetch past end of record stream
SQL> --
SQL> -- The CLOSE statement deletes the result table.
SQL> --
SQL> CLOSE CURRENT_MBMF;
SQL> --
SQL> -- However, the cursor declaration remains in effect during
SQL> -- an entire interactive session or program execution cycle
SQL> -- unless you disconnect from the session using the DISCONNNECT
SQL> -- statement or attach to another database using the same alias.
SQL> --
SQL> -- Another OPEN statement creates the result table again.
SQL> --
SQL> OPEN CURRENT_MBMF;
SQL> --
SQL> -- After the cursor is opened again, the FETCH statement starts
SQL> -- data retrieval at the first row in the result table.
SQL> --
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 APGM       00174         22-Sep-1981
SQL> --
SQL> -- If a cursor is already open, an OPEN statement generates an
SQL> -- error message.
SQL> --
SQL> OPEN CURRENT_MBMF;
%SQL-F-CURALROPE, Cursor CURRENT_MBMF was already open
SQL> --
SQL> -- The next FETCH statement retrieves the next row.
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 ASCK       00165          8-Mar-1981
SQL> --
```

(continued on next page)

Example 18–1 (Cont.) Using Table Cursors

```
SQL> -- A COMMIT or ROLLBACK statement implicitly executes a CLOSE
SQL> -- statement, which deletes the result table.  An attempt to
SQL> -- fetch a row after ending the transaction fails because no
SQL> -- result table exists for the cursor declaration.
SQL> --
SQL> COMMIT;
SQL> FETCH CURRENT_MBMF;
%SQL-F-CURNOTOPE, Cursor CURRENT_MBMF is not opened
SQL>
```

You cannot override cursor declarations. If you open the cursor and find that you have selected far too many rows (you have to enter many FETCH statements to get to the rows you want to change), you have two choices:

- Close the cursor and then declare and open another cursor (using a different name) that is more specific in selecting rows.

  If you start a read/write transaction and are accessing rows that other users may want to update, end your transaction to release any locks you may have placed on rows that you do not plan to use. (Closing a cursor does not release any locks you place on rows; specifying a COMMIT or ROLLBACK statement does.) When you start read/write transactions in a multiuser database environment, try to avoid creating cursors that contain many more rows than you need, and keep transactions as short as possible.

- Issue a DISCONNECT statement and attach to the database again. You can then declare the cursor again using the same name but a different select expression, or you can attach to a database with the same alias as the database for the cursor. See Section 15.6 for more information about the DISCONNECT statement.

Reference Reading

The *Oracle Rdb7 Introduction to SQL* also includes more information about the DISCONNECT statement.

18.5 Using Holdable Cursors

Holdable cursors provide more control in using and positioning table cursors by letting cursors remain open across transactions. When you declare a holdable cursor, it remains open and the row pointer maintains its position in the result table even when a transaction ends.

Consider using a holdable cursor when your application needs to query a user after it fetches each row, but before it modifies it. Holdable cursors let you release locks by committing or rolling back the transaction, but still preserve the positioning of the cursor.

You can use the following options with holdable cursors:

- Use the WITH HOLD PRESERVE ON COMMIT clause to specify that the cursor remain open when you commit a transaction. If you rollback the transaction, SQL implicitly closes the cursor.

  Because the PRESERVE clause is an extension to the SQL standard and PRESERVE ON COMMIT is the default, Oracle Rdb recommends that you use only the WITH HOLD clause.
- Use the WITH HOLD PRESERVE ON ROLLBACK clause to specify that the cursor remain open when you roll back a transaction. If you commit the transaction, SQL implicitly closes the cursor.
- Use the WITH HOLD PRESERVE ALL clause to specify that the cursor remain open when you commit *or* roll back a transaction.
- Use the WITH HOLD PRESERVE NONE clause to override the SET HOLD CURSOR clause.

Example 18–2 shows how a holdable cursor works. Contrast it to the basic table cursor in Example 18–1.

Example 18–2 Using Holdable Table Cursors

```
SQL> DECLARE CURRENT_MBMF TABLE CURSOR WITH HOLD PRESERVE ON COMMIT FOR
cont>    SELECT JOB_CODE, EMPLOYEE_ID, JOB_START FROM JOB_HISTORY
cont>       WHERE (DEPARTMENT_CODE = 'MBMF') AND
cont>             (JOB_END IS NULL)
cont>       ORDER BY JOB_CODE, JOB_START;
SQL> --
SQL> OPEN CURRENT_MBMF;
SQL> --
```

(continued on next page)

Example 18–2 (Cont.) Using Holdable Table Cursors

```
SQL> -- The first FETCH statement retrieves the first row.
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 APGM       00174         22-Sep-1981
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 ASCK       00165          8-Mar-1981
SQL> --
SQL> -- Commit the transaction.
SQL> COMMIT;
SQL> --
SQL> -- Because this is a holdable cursor using the WITH HOLD
SQL> -- PRESERVE ON COMMIT clause, the cursor remains open and
SQL> -- the row pointer retains its position.  The next FETCH
SQL> -- statement reads the next record.
SQL> FETCH CURRENT_MBMF;
 JOB_CODE   EMPLOYEE_ID   JOB_START
 DMGR       00227         25-Nov-1981
```

You can use the SET HOLD CURSOR statement to specify a default hold attribute for all cursors that do not specify a WITH HOLD clause. The SET HOLD CURSORS statement controls the hold attributes of all cursors that you declare *after* you specify the SET HOLD CURSORS statement. For example, to force cursors that do not specify the WITH HOLD clause to remain open on commit, use the following statement:

```
SQL> SET HOLD CURSORS 'ON COMMIT';
```

18.6 Using List Cursors

List cursors let you read from and insert large data structures (lists) into a row. A list is an ordered collection of elements or segments of the data type LIST OF BYTE VARYING. Lists are frequently used to store multimedia objects, such as video displays, long pieces of text, audio recordings, and so forth.

Because lists exist as a set of elements with a row of a table and you use a table cursor to position on a specific row in the table, a list cursor must refer to a table cursor.

You can use list cursors to read or insert lists; however, you cannot use list cursors to update a list.

To read or insert a list, you must first declare a table cursor and then specify both the table column and list column names in a select list. The table cursor provides the row context for the list cursor. In other words, before you can read or insert a list in row 1, you must position the table cursor on row 1.

After you declare the table cursor, you declare the list cursor and specify the table cursor in the WHERE CURRENT OF clause of the DECLARE CURSOR statement. The WHERE CURRENT OF clause associates the list cursor with the table cursor.

Remember that before you open a list cursor, you must open the table cursor and position it on a row by using the FETCH statement. Example 18–3 shows how to declare a list cursor and read lists. It reads the employee ID and the resume in the RESUMES table. The RESUME column is defined as a data type of LIST OF BYTE VARYING.

Example 18–3 Using List Cursors

```
SQL> -- Declare the table cursor.
SQL> DECLARE RESUME_CURS TABLE CURSOR FOR
cont>   SELECT EMPLOYEE_ID, RESUME FROM RESUMES;
SQL> --
SQL> -- Open the table cursor.
SQL> OPEN RESUME_CURS;
SQL> --
SQL> -- Fetch the first row.  The table cursor returns the employee ID
SQL> -- and segmented string identifier for the RESUME column.
SQL> --
SQL> FETCH RESUME_CURS;
    EMPLOYEE_ID   RESUME
 00164                         86:2:4
SQL> --
SQL> -- To read the contents of the RESUME column, you must declare
SQL> -- a list cursor and position it on the row.
SQL> --
SQL> DECLARE LIST_CURS LIST CURSOR FOR
cont>   SELECT RESUME WHERE CURRENT OF RESUME_CURS;
SQL> --
SQL> -- Open the list cursor.
SQL> OPEN LIST_CURS;
SQL> --
```

(continued on next page)

Example 18–3 (Cont.) Using List Cursors

```
SQL> -- Fetch the first segment of the list.   Remember that
SQL> -- the table cursor is already opened and positioned on
SQL> -- the row containing employee ID 00164.
SQL> FETCH LIST_CURS;
    RESUME
    This is the resume for Alvin Toliver
SQL> --
SQL> -- Fetch the remaining segments of the list.
SQL> FETCH LIST_CURS;
    RESUME
    Boston, MA
SQL> FETCH LIST_CURS;
    RESUME
 Oracle Corporation
SQL> FETCH LIST_CURS;
    RESUME
%RDB-E-STREAM_EOF, attempt to fetch past end of record stream
SQL> --
SQL> -- To move to the next row, first close the list cursor, then
SQL> -- fetch the next table row.
SQL> --
SQL> CLOSE LIST_CURS;
SQL> FETCH RESUME_CURS;
    EMPLOYEE_ID   RESUME
 00165                         86:2:8
SQL> --
SQL> -- Open the list cursor.
SQL> OPEN LIST_CURS;
SQL> -- Fetch the first segment of the list for employee 00165.
SQL> FETCH LIST_CURS;
    RESUME
    This is the resume for Terry Smith
SQL> --
SQL> -- Close the cursors.
SQL> CLOSE LIST_CURS;
SQL> CLOSE RESUME_CURS;
```

18.7 Using Scrollable List Cursors

A scrollable list cursor differs from a one-direction (nonscrollable) list cursor in that it permits applications to scan randomly through a result table. An application can fetch up or down, fetch the first or last row directly, or fetch any single row randomly. The following example shows how to declare a scrollable list cursor:

```
DECLARE BCURSOR SCROLL LIST CURSOR
        FOR SELECT RESUME WHERE CURRENT OF ACURSOR;
```

The FETCH statement allows the following options on a SCROLL list cursor:

- FETCH NEXT
- FETCH PRIOR
- FETCH FIRST
- FETCH LAST
- FETCH RELATIVE simple-value-expression
- FETCH ABSOLUTE simple-value-expression

The simple-value-expression must be either a positive or negative integer, or a numeric module language or host language parameter. The *Oracle Rdb7 SQL Reference Manual* includes detailed information about these options.

Example 18–4 shows an excerpt of a C program with embedded SQL statements that use scrollable list cursors.

Example 18–4 Using Scrollable List Cursors

```
   .
   .
   .
/* Declare a table cursor. */
EXEC SQL DECLARE ONE TABLE CURSOR
                FOR SELECT EMPLOYEE_ID, RESUME FROM RESUMES
                WHERE EMPLOYEE_ID = :sel_emp_id ;

EXEC SQL DECLARE TWO READ ONLY SCROLL LIST CURSOR
                FOR SELECT RESUME
                WHERE CURRENT OF ONE;

EXEC SQL OPEN ONE;
     dump_error();

EXEC SQL FETCH ONE INTO :emp_id, blob;
     dump_error();

EXEC SQL OPEN TWO;
     dump_error();

EXEC SQL FETCH LAST FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();
```

(continued on next page)

Example 18–4 (Cont.) Using Scrollable List Cursors

```
EXEC SQL FETCH FIRST FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH NEXT FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH FIRST FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH LAST FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH PRIOR FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

EXEC SQL FETCH ABSOLUTE 1 FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

two_s = 2;

EXEC SQL FETCH ABSOLUTE :two_s FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();

two_l = 2;

EXEC SQL FETCH ABSOLUTE :two_l FROM TWO INTO :seg2;
     printf("Segment returned: %s\n", seg2 );
     dump_error();
   .
   .
   .
```

18.8 Using Dynamic Cursors

You use dynamic cursors in dynamic SQL programs. Dynamic cursors are different than regular (or static) cursors because you do not explicitly specify the SELECT statement in the source code. Instead, you specify the name of a prepared statement. The cursor name is known at compile time, but the SELECT statement is not known until run time. You can use dynamic cursors when you do not know what query a user will input.

To use dynamic cursors, first declare the cursor using the dynamic DECLARE CURSOR statement. Then, prepare the SELECT statement for dynamic execution using the PREPARE statement.

Example 18–5 shows an SQL module that uses dynamic cursors to fetch rows from a table.

Example 18–5 Using Dynamic Cursors

```
--  This module uses dynamic cursors to fetch rows one at a time.
--
MODULE          C_MOD_DYN_CURS
LANGUAGE        GENERAL
AUTHORIZATION   RDB$DBHANDLE
PARAMETER COLONS

DECLARE ALIAS FILENAME personnel

-- Declare the dynamic cursor. Use a statement name to identify
-- a prepared SELECT statement.
DECLARE CURSOR1 CURSOR FOR STMT_NAME

-- Prepare the statement from a statement entered at run time.
-- Specify that SQL write information about the number and
-- data type of select list items to the SQLDA.
PROCEDURE PREP_STMT
    (SQLCODE,
     :COMMAND_STRING  CHAR (256));

      PREPARE STMT_NAME  FROM :COMMAND_STRING;

PROCEDURE DESCRIBE
    (SQLCODE,
     SQLDA);

      DESCRIBE STMT_NAME SELECT LIST INTO SQLDA;

PROCEDURE OPEN_CURSOR
    (SQLCODE);

    OPEN CURSOR1;
```

(continued on next page)

Example 18–5 (Cont.) Using Dynamic Cursors

```
PROCEDURE FETCH_CURSOR
    (SQLCODE,
     SQLDA);

    FETCH CURSOR1 USING DESCRIPTOR SQLDA;

PROCEDURE CLOSE_CURSOR
    (SQLCODE);

    CLOSE CURSOR1;

PROCEDURE ROLLBACK
    (SQLCODE);

    ROLLBACK;
```

For more information about dynamic SQL and dynamic cursors, see Chapter 11.

18.9 Using Extended Dynamic Cursors

You use extended dynamic cursors in dynamic SQL programs. When you use extended dynamic cursors, you supply parameters for both the cursor name and the SELECT statement. The cursor name and SELECT statement are not known until run time. Extended dynamic cursors let you use one set of cursor-related statements to process any number of dynamically generated statements.

To use extended dynamic cursors, first declare the cursor using the extended dynamic DECLARE CURSOR statement. Then, prepare the SELECT statement for dynamic execution using the PREPARE statement.

Example 18–6 shows an SQL module that uses extended dynamic cursors to fetch rows from a table.

Example 18–6 Using Extended Dynamic Cursors

```
--
--  This module uses extended dynamic cursors to fetch rows one at a time.
--
MODULE          MOD_C_EXTDYN_CURS
LANGUAGE        C
AUTHORIZATION   RDB$DBHANDLE
PARAMETER COLONS
```

(continued on next page)

Example 18–6 (Cont.) Using Extended Dynamic Cursors

```
DECLARE ALIAS FILENAME personnel

-- Prepare the statement from a statement entered at run time.

PROCEDURE PREP_STMT
    (SQLCODE,
     :STMTID  INTEGER,
     :COMMAND_STRING  CHAR (256),
     SQLDA);

      PREPARE :STMTID FROM :COMMAND_STRING;

-- Specify that SQL write information about the number and
-- data type of select list items to the SQLDA.
PROCEDURE DESCRIBE
    (SQLCODE,
     :STMTID INTEGER,
     SQLDA);

      DESCRIBE :STMTID SELECT LIST INTO SQLDA;

-- Declare an extended dynamic cursor.

PROCEDURE DEC_CUR_FROM_STMT
    (SQLCODE,
     :CURSOR_NAME CHAR(32),
     :STMTID INTEGER );

    DECLARE :CURSOR_NAME CURSOR FOR :STMTID;

PROCEDURE OPEN_CURSOR_NAME
    (SQLCODE,
     :CURSOR_NAME CHAR(32));

    OPEN :CURSOR_NAME;

PROCEDURE FETCH_CURSOR_NAME
    (SQLCODE,
     :CURSOR_NAME CHAR(32),
     SQLDA);

    FETCH :CURSOR_NAME USING DESCRIPTOR SQLDA;
```

(continued on next page)

Example 18–6 (Cont.) Using Extended Dynamic Cursors

```
PROCEDURE CLOSE_CURSOR_NAME
    (SQLCODE,
     :CURSOR_NAME CHAR(32));

    CLOSE :CURSOR_NAME;

PROCEDURE ROLLBACK
    (SQLCODE);

    ROLLBACK;
```

For more information about dynamic SQL and extended dynamic cursors, see Chapter 11.

19

Inserting, Updating, and Deleting Data

This chapter explains how to use programs to insert, update, and delete table rows. It also shows how to use list cursors to insert large data structures into a database.

The sections that follow explain how to:

- Load a database
- Insert rows
- Use list cursors to insert large data structures
- Update rows
- Delete rows
- Use triggers with insert, update, and delete operations

19.1 Loading a Database

To load data into tables with SQL, you use the INSERT statement, which is discussed in Section 19.2.

If you are loading large amounts of data (from a file or another table, for example, refer to the chapter about loading data in the *Oracle Rdb7 Guide to Database Design and Definition*. That chapter discusses different ways to load data and describes how to make the load operation more efficient.

Several online programs, all beginning with "sql_load", demonstrate how to stores row into tables of the sample databases. The programs are available in several host languages in the samples directory.

19.2 Inserting Rows

To store rows, use the INSERT statement, which has two general forms:

- INSERT . . . VALUES
- INSERT . . . SELECT

Both forms identify a table or view and the columns in which to store values. However, in the INSERT . . . VALUES form, you explicitly specify values for only one row; in the INSERT . . . SELECT form, you specify a select expression to supply values for one or more rows.

19.2.1 Using the INSERT . . . VALUES Statement

The first form of the INSERT statement, INSERT . . . VALUES, inserts data that is listed in the VALUES clause. The data can be explicitly listed in the VALUES clause or parameters can represent the data to be loaded by a user or from a data file.

Example 19–1, an excerpt of the program sql_load_employees.sc, loads the EMPLOYEES table from a data file.

Example 19–1 Loading a Table from a Data File in a Precompiled C Program

```
   .
   .
   .
/*
In the INSERT statement, the list of names in the VALUES clause corresponds
to the host variables containing the values.  The list of names that follows
the INSERT clause names the columns in the table that are to be inserted.
*/
EXEC SQL
   -- Insert the row.
   INSERT INTO EMPLOYEES
              (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
              ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE, SEX, BIRTHDAY,
              STATUS_CODE)
          VALUES
              (:emp_id, :last, :first, :middle:middle_ind,
               :address, :city, :state, :zip, :sex,
               :date_out, :status);
   .
   .
   .
```

When you create a program to insert rows, you may need to store either an actual or a null value for a particular column; you cannot predict for all rows being stored which of the two types of values a column will have. For example, in Example 19–1 a null value may be loaded into the MIDDLE_INITIAL column. To handle the null value, an indicator parameter, middle_ind, is passed to SQL.

For more information about working with stored and null values in programs, see Section 8.10.

19.2.2 Using the INSERT . . . SELECT Statement

The second form of the INSERT statement, INSERT . . . SELECT, stores rows from another source into a table. This form of the statement is useful when you extract data for a replicate database. It is also useful when you restructure a table in which data is already stored.

You can use this form of INSERT statement in interactive SQL or in a program. Note that when rows being inserted in a table are transferred from other tables or views, programs do not have to include variable declarations to store column values. When a select expression specifies the data, SQL can transfer the data without intermediate processing by the program.

Use the INSERT statement to:

- Update the data in a table that performs an informational function (such as a view or complex query) and that actually stores data. Give users read-only privileges for this table.

  A view can access many tables in complex ways. If many users and programs frequently request information through the view, database performance may suffer. If some users who retrieve data through the view can be satisfied by information that is not completely current, you can create an extra table in which to store that data. In this case, the INSERT statement that updates the data in this extra table reads the view only occasionally. This method reduces the number of times the database system must perform complex operations, and database performance may improve.

- Store data from more than one query into only one table.

  If you combine the values of columns in one table with the values of columns in another table using the UNION operator, you can store the results in a table.

- Copy data that resides in a remote database to your own system.

For example, if you plan queries or reports that frequently use remote Oracle Rdb databases or data from databases other than Oracle Rdb databases, you can periodically copy the remote data into a local database. Then you can generate queries and reports against the local database. Response time for queries against a local database is better than response time for queries against a remote database.

- Store data into one or several tables that have been created only temporarily to be the basis for complex or spontaneous reports or "what if" calculations.

  If you know that you will be working on complex calculations or queries, particularly if these access many tables, you can create tables to use temporarily for early stages of your work. (This is the time you enter the same sort of query repeatedly until it is refined.) These tables could contain a representative subset of the data in the original tables. While using the tables, you obtain results faster and do not interfere with other users' access to permanent tables. When your query or calculation is refined, you delete the tables you created and revise the query to specify permanent tables.

You can also use an INSERT statement that includes a select expression as part of an operation to restructure a table.

If you plan to delete and create a table again in a database on which your site depends, follow these guidelines:

- Take necessary safeguards to protect data in its original form (back up the database).
- Prevent users from writing to or reading from any of the tables you are working on until you finish. (To do this, reserve tables for EXCLUSIVE WRITE in a SET TRANSACTION statement.)

  You should be the only user accessing the database when you are changing a pivotal table. Otherwise, your operations can disrupt the activities of other users, and the activities of other users can interfere with what you are doing.

19.3 Using List Cursors to Insert Large Data Structures

If a table contains a list, you must use a list cursor to insert data into the list. A list is an ordered collection of elements of the data type LIST OF BYTE VARYING. A list is sometimes called a segmented string. List cursors enable you to scan through very large data structures from within a language that does not provide support for objects of such size.

First, declare a table cursor and then specify both the table column and list column names in the FROM clause of the DECLARE CURSOR statement. After you declare the table cursor, declare the list cursor and specify the table cursor in the WHERE CURRENT OF clause of the DECLARE CURSOR statement. The WHERE CURRENT OF clause associates the list cursor with the table cursor. Section 18.6 describes list cursors in more detail.

In order to insert lists into an existing row, the list cursor must use insert-only mode. Before opening the list cursor, open the table cursor and use the FETCH statement to position the cursor on the desired row.

If you want to insert a new row that will contain a list, both the table cursor and the list cursor must use insert-only mode. Be careful not to add data to an existing list, because SQL replaces the previous list with the data you just inserted.

The sql_resumes program in the samples directory demonstrates how to insert lists into rows using list cursors.

19.4 Updating Rows

To modify values in existing rows, use the UPDATE statement. Unlike inserting rows, updating rows is a more complex operation that involves selecting the row (or set of rows) whose values you need to modify. There are several methods for using the UPDATE statement:

- You can include data selection conditions in the UPDATE statement.
- You can use the UPDATE statement with a cursor.
- You can use an UPDATE . . . RETURNING statement.

Reference Reading

For more information about updating rows, see the *Oracle Rdb7 SQL Reference Manual*. The section on the UPDATE statement provides details about the update operation itself. The sections on the DECLARE CURSOR statement, CREATE VIEW statement (view update examples), and FETCH statement (INTO clause) provide additional guidance on operations that affect data modification.

19.4.1 Selecting Data in the UPDATE Statement

When you want to change values in only one row, use a simple UPDATE statement with a SET clause. This form of the UPDATE statement has the following format:

```
UPDATE table-or-view-name SET list-of-column-assignments
       WHERE condition(s)-identifying-row(s)-to-be-modified ;
```

You can also update multiple rows this way, but only if you are setting all column occurrences to the same value expression. For example, you can specify multiple JOB_HISTORY rows in the WHERE clause of the UPDATE statement to store the same value for supervisor ID in all the rows.

The following extracts from a precompiled SQL program show this kind of update statement:

```
EXEC SQL UPDATE JOB_HISTORY SET JOB_END = :BINTIMOUT
                WHERE EMPLOYEE_ID = :ID-NUMBER
                      AND JOB_END IS NULL;
   .
   .
   .
EXEC SQL UPDATE SALARY_HISTORY SET SALARY_END = :BINTIMOUT
                WHERE EMPLOYEE_ID = :ID-NUMBER
                      AND SALARY_END IS NULL;
   .
   .
   .
```

19.4.2 Using the UPDATE Statement with a Cursor

To change values in more than one row at a time, use the UPDATE statement with a cursor. This form of UPDATE statement has the following format:

```
DECLARE cursor-name CURSOR FOR
        select-expression
        FOR UPDATE OF list-of-columns-to-be-modified ;
OPEN cursor-name;
                                    -+
FETCH cursor name;                   | Repeat FETCH and
UPDATE table-or-view-name            | UPDATE statements
       SET list-of-column-assignments | to change each
       WHERE CURRENT OF cursor-name ; | row in the cursor.
                                    -+
```

In this case, you work with each row individually and can specify values for updated columns that are different for each row.

You can specify a FOR UPDATE clause in a cursor declaration to identify the columns you want to update. You can update rows from a cursor whether or not you include a FOR UPDATE clause in your cursor declaration. However, if you include the FOR UPDATE clause and update columns other than the columns named in the clause, SQL returns a warning message because you are changing values you may not have intended to change.

When you declare a cursor for the purpose of updating rows, your DECLARE CURSOR statement must include SELECT statement syntax that allows rows to be updated. The rules for determining whether a SELECT statement will allow rows to be updated are defined differently by the ANSI/ISO SQL standard and by Oracle Rdb. If you refer to one table in your DECLARE CURSOR statement and optional WHERE clause, it is safe to assume that the cursor will allow rows to be updated.

Example 19–2 illustrates an UPDATE statement with a cursor in a precompiled SQL program. The example is an extract from the source code for the precompiled C program sql_terminate.sc. The complete source file for the program (along with sources in other languages) is available on line, in the samples directory.

Example 19–2 Updating Rows in a Precompiled C Program

```
/*  ABSTRACT:
*
*       This program demonstrates the use of a cursor to fetch and update
*       rows in the database using the C precompiler.
*
*       This program prompts the operator for the employee ID and termination
*       date of an employee, until the operator asks to exit.  It uses
*       the employee ID to check for the employee by opening a cursor
*       and fetching the employee row.  If the employee is found in the
*       database, it updates three tables with the employee's status and
*       termination date.  Then, it commits the transaction and prompts
*       the operator for the next request.
*/
   .
   .
   .
/* Declare the cursor to be used for fetching employee records. */
    EXEC SQL DECLARE EMPLROW CURSOR FOR
                SELECT EMPLOYEE_ID, STATUS_CODE FROM EMPLOYEES
                WHERE EMPLOYEE_ID = :employee_id;
```

(continued on next page)

Example 19–2 (Cont.) Updating Rows in a Precompiled C Program

```
   .
   .
   .
/* Open the cursor that had been previously declared. */
    EXEC SQL OPEN EMPLROW;
   .
   .
   .
/* Fetch a row from the opened cursor. */
    EXEC SQL FETCH EMPLROW INTO :e_id, :status_code;
   .
   .
   .
/* Update the EMPLOYEES table. */
    EXEC SQL UPDATE EMPLOYEES
                SET STATUS_CODE = '0'
                WHERE CURRENT OF EMPLROW;
   .
   .
   .
```

19.4.3 Using the UPDATE . . . RETURNING Statement

The UPDATE . . . RETURNING statement allows you to request that updated values be returned to your program. It can be used only when you update a single row.

You can request that, after the row is updated, the database system return the value of the dbkey of the row. Subsequent queries can use the dbkey value to access the record directly.

The following example shows how to use an UPDATE statement to return the dbkey:

```
EXEC SQL UPDATE JOB_HISTORY
            SET JOB_END = :BINTIMOUT
            WHERE EMPLOYEE_ID = :ID-NUMBER
                  AND JOB_END IS NULL
     RETURNING DBKEY INTO :TERM_DBKEY;
```

Subsequent queries can use the dbkey value to look up this job history record for tasks such as calculating termination benefits or issuing a report to a state agency.

Instead of the DBKEY keyword, you can use the ROWID keyword.

The RETURNING clause can return more than one value, and the value need not be a dbkey, as shown in this example:

```
EXEC SQL UPDATE ACCOUNT
                SET AMOUNT = AMOUNT + :TXN
                WHERE ACCOUNT_NUMBER = :ACCNUM
                RETURNING AMOUNT INTO :NEW_AMOUNT;
```

19.5 Deleting Rows

The DELETE statement lets you erase rows from tables. You can use the DELETE statement in two ways: alone or associated with a cursor.

When you use the DELETE statement as a standalone statement, specify the table name and the conditions that identify the rows to be deleted:

```
DELETE FROM EMPLOYEES
       WHERE EMPLOYEE_ID = '00164';
```

When you use the DELETE statement with a cursor, specify the table name and use the CURRENT OF clause to specify that the row to be deleted is the row to which the cursor points. The following example shows the cursor declaration, the associated cursor statements, and the DELETE statement:

```
-- Declare the cursor.
DECLARE DEL_CURSOR CURSOR FOR
        SELECT DEPARTMENT_CODE, DEPARTMENT_NAME
        FROM DEPARTMENTS;

OPEN DEL_CURSOR;

-- Fetch the row.
FETCH DEL_CURSOR;

-- Delete the current row.
DELETE FROM DEPARTMENTS WHERE CURRENT OF DEL_CURSOR;
```

Note that you can optionally specify a FOR UPDATE clause in a cursor declaration to identify the columns you want to update.

When you declare a cursor for the purpose of deleting rows, the DECLARE CURSOR statement must include SELECT statement syntax that allows rows to be deleted. The rules for determining whether a SELECT statement will allow rows to be deleted are defined differently by the ANSI/ISO SQL standard and Oracle Rdb. If you refer to one table in your DECLARE CURSOR statement and optional WHERE clause, it is generally safe to assume that the cursor will allow rows to be deleted.

Reference Reading

For more information about the DELETE statement, see the chapter on SQL statements in the *Oracle Rdb7 SQL Reference Manual*.

19.6 Deleting List Data

You cannot delete a column of data type LIST unless you delete the entire row. However, you can set the column to NULL, as shown in Example 19–3.

Example 19–3 Deleting List Data From a Row

```
SQL> -- Set the list column, RESUME, to NULL
SQL> UPDATE RESUMES
cont>       SET RESUME = NULL
cont>       WHERE EMPLOYEE_ID ='00164';
1 row updated
SQL> select * from resumes;
 EMPLOYEE_ID   RESUME
 00164         NULL
 00165                          86:2:8
 00166                         86:2:12
3 rows selected
```

19.7 Using Triggers with Insert, Update, and Delete Operations

When you insert, update, or delete data in a database with a defined trigger, you cause other actions to be automatically performed on the database. A **trigger** causes one or more actions to be performed before or after a particular write operation is performed. For example, you can define a trigger that tracks which users make changes to the database and the date and time of the changes. The *Oracle Rdb7 Guide to Database Design and Definition* explains how to define triggers and shows an example of a trigger definition using the CURRENT_TIMESTAMP function to track changes to the data.

20

Using the Multiple Schema Option

An Oracle Rdb database can contain multiple schemas. The SQL interface to Oracle Rdb lets you use the ANSI/ISO standard naming of multiple schemas (multischemas) within a catalog and multiple catalogs within an Oracle Rdb database. The multischema SQL option does not change how Oracle Rdb stores objects, only how SQL names those objects.

The sections that follow tell you how to:

- Use the terminology and concepts associated with multischema databases
- Name multischema objects in programs written in SQL module language
- Name multischema objects in precompiled SQL programs

Reference Reading

To understand the multischema SQL option completely, see the following manuals:

- The *Oracle Rdb7 SQL Reference Manual* discusses multischema terms, syntax for creating and altering multischema databases, multischema naming conventions, and reference information about how to use the multischema option in the SQL module language and the SQL precompiler.
- The *Oracle Rdb7 Guide to Database Design and Definition* describes how to alter, create, and delete multischema databases and objects and shows the data definitions for the sample multischema corporate_data database used in examples throughout this chapter.

20.1 Understanding Multischema Databases

SQL defines schema and catalog objects in a multischema Oracle Rdb database as follows:

- A **schema** is a group of objects within a catalog. The schema contains objects such as tables, views, constraints, triggers, domains, collating sequence, storage maps, and indexes and privileges for each of these objects. You can create one or more schemas within a catalog in a multischema database.

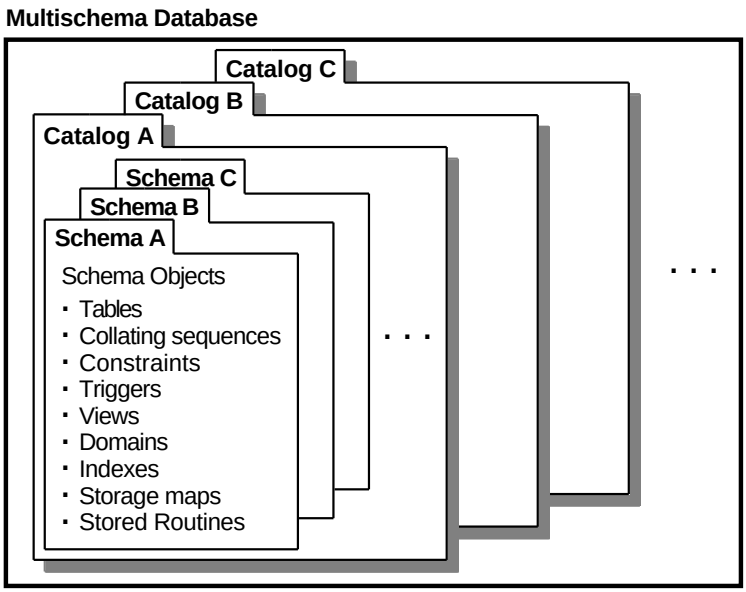
  Each multischema database contains at least one schema, called RDB$SCHEMA, which is contained in the Oracle Rdb default database catalog, RDB$CATALOG.

- A **catalog** is a multischema object that identifies a group of schemas within a multischema database. You can create one or more catalogs in a multischema database.

  Each multischema database contains at least one catalog, called RDB$CATALOG, which contains the database schema, RDB$SCHEMA.

A **multischema database** therefore contains one or more catalogs that can each contain one or more schemas. Figure 20–1 shows a multischema database.

Figure 20–1 Structure of a Multischema SQL Database



You enable the multischema attribute on an Oracle Rdb database by specifying the MULTISCHEMA IS ON clause of the CREATE DATABASE or ALTER DATABASE statement. By default, SQL creates a database with only one schema and does not allow you to create additional schemas in that database.

20.2 Using Multischema Databases with the SQL Module Processor

This section describes the default settings that apply when you process an SQL module. It shows a sample module file that uses multischema names.

20.2.1 Setting Defaults for SQL Modules

You can set the default catalog and default schema in the module header when you use an Oracle Rdb database that has the multischema attribute. If you set a default schema and catalog, you do not need to include a catalog and schema name to qualify every object to which you refer. Objects within the default schema and default catalog can be referenced without schema and catalog qualification. Thus, if you plan to work primarily with one schema and one catalog, set the defaults in the header of the module.

You can also specify the default catalog and schema in a context file.

When you do not qualify a schema object and the SQL context is ambiguous, SQL uses the defaults listed in Table 20–1.

Table 20–1 Module Defaults for Multischema SQL

Compiler Attribute	Implicit Default
Alias	Authorization identifier
Authorization identifier	User name
Catalog	RDB$CATALOG
Schema	Authorization identifier

When you select the multischema option, the default authorization identifier for each schema is the user name of the user compiling the module. You can specify a different authorization identifier by using the AUTHORIZATION clause in the module header.

This authorization identifier defines the default alias and schema. If you do not qualify the name of a schema element, SQL implicitly qualifies it with the current authorization identifier. You can use the ALIAS and SCHEMA clauses to override the default settings. When you do not specify an alias or a schema in the module header, but do explicitly specify an authorization identifier in the module header, SQL uses the authorization identifier defined in the module header as the default alias and default schema. If you do not specify a default schema name in a module header, you must specify an authorization identifier. The RDB$CATALOG catalog name remains the default until overridden by the CATALOG clause.

The authorization identifier is used for privilege checking. If you specify a RIGHTS clause with the RESTRICT option in the module header, SQL bases privilege checking on the default authorization identifier, in compliance with the ANSI/ISO SQL standard, and does not allow execution of the program at run time unless the compile-time authorization identifier matches the run-time authorization identifier. If you specify RIGHTS INVOKER (the default), SQL bases privilege checking on the user name of the person who executes the module.

20.2.2 Using Multischema Naming in an SQL Module File and C Program

Example 20–1 illustrates an SQL module, sql_msdb_mod.sqlmod. You can find a copy of this module and the programs that call it in the samples directory.

Example 20–1 Using Multischema Names in an SQL Module File

```
-------------------------------------------------------------------------
-- This module shows how to use SQL module language when working with a
-- multischema database and using multischema naming conventions. The
-- C program sql_msdb.c with which this module is linked uses the sample
-- multischema database corporate_data.
-------------------------------------------------------------------------
-- Header Information Section
-------------------------------------------------------------------------
MODULE          sql_msdb_mod      -- Module name
LANGUAGE        C                 -- Language of calling program
CATALOG         ADMINISTRATION    -- Set default catalog ❶
SCHEMA          PERSONNEL         -- Set default schema  ❷
PARAMETER       COLONS            -- Require colons before parameter names
-------------------------------------------------------------------------
-- DECLARE Statements Section
-------------------------------------------------------------------------
DECLARE ALIAS FILENAME corporate_data   -- Declare the database.

-- Declare table cursor for the DEPARTMENTS table in the ACCOUNTING schema.
DECLARE DEPT_CURSOR TABLE CURSOR FOR
    SELECT DEPARTMENT_CODE, DEPARTMENT_NAME, MANAGER_ID
        FROM ACCOUNTING.DEPARTMENTS ❸
        ORDER BY DEPARTMENT_CODE

-- Declare table cursor for the EMPLOYEES table in the PERSONNEL schema.
DECLARE EMP_CURSOR TABLE CURSOR FOR
    SELECT E.EMPLOYEE_ID,  E.LAST_NAME, E.FIRST_NAME
        FROM EMPLOYEES E, JOB_HISTORY JH ❹
        WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID
              AND JH.DEPARTMENT_CODE = :DEPT_CODE
              AND JH.JOB_END IS NULL
        ORDER BY E.LAST_NAME
```

(continued on next page)

Example 20–1 (Cont.) Using Multischema Names in an SQL Module File

```
-------------------------------------------------------------------------
-- Procedure Section
-------------------------------------------------------------------------
❺

-- Opens the cursor declared for the DEPARTMENTS table.
PROCEDURE OPEN_DEPT_CURSOR
    (SQLCODE);

    OPEN DEPT_CURSOR;

-- Opens the cursor declared for the EMPLOYEES table.
PROCEDURE OPEN_EMP_CURSOR
    (SQLCODE,
     :DEPT_CODE CHAR(4));

    OPEN EMP_CURSOR;

-- Fetches data from the opened cursor for the DEPARTMENTS table.
PROCEDURE FETCH_DEPT_DATA
    (SQLCODE,
     :DEPT_CODE    CHAR(4),
     :DEPT_NAME    CHAR(20),
     :DEPT_MAN_ID  CHAR(5));

    FETCH DEPT_CURSOR INTO
                :DEPT_CODE, :DEPT_NAME, :DEPT_MAN_ID;

-- Fetches data from the opened cursor for the EMPLOYEES table.

PROCEDURE FETCH_EMP_DATA
    (SQLCODE,
     :EMP_ID          CHAR(5),
     :EMP_LAST_NAME    CHAR(20),
     :EMP_FIRST_NAME   CHAR(20));

    FETCH EMP_CURSOR INTO
                :EMP_ID, :EMP_LAST_NAME, :EMP_FIRST_NAME;

-- Closes the DEPT_CURSOR cursor for the DEPARTMENTS table.
PROCEDURE CLOSE_DEPT_CURSOR
   (SQLCODE);

   CLOSE DEPT_CURSOR;
```

(continued on next page)

Example 20–1 (Cont.) Using Multischema Names in an SQL Module File

```
-- Closes the EMP_CURSOR cursor for the EMPLOYEES table.
PROCEDURE CLOSE_EMP_CURSOR
   (SQLCODE);

   CLOSE EMP_CURSOR;
```

The following descriptions are keyed to the numbered items in Example 20–1:

❶ CATALOG clause

Specifying a catalog name in the module header lets you reference schema objects without qualifying them with a catalog name. The CATALOG clause changes the default catalog from RDB$CATALOG to the ADMINISTRATION catalog, which contains both the ACCOUNTING and PERSONNEL schemas used in the module.

❷ SCHEMA clause

Specifying a schema name in the module header lets you reference schema objects without qualifying them with a schema name. If you do not specify a default schema name, you have to specify an authorization identifier with the AUTHORIZATION clause. Otherwise, SQL issues an error message such as"%SQL-F-SCHNOTDEF, Schema SCHWARTZ is not defined." This message indicates that SQL is looking for a schema with the same name as the user name of the user who compiled the module.

❸ ACCOUNTING schema object qualification

Because the DEPARTMENTS table belongs to the ACCOUNTING schema and not to the default PERSONNEL schema, you must qualify the DEPARTMENTS table with the ACCOUNTING schema name. If you had not qualified the DEPARTMENTS table, SQL would have tried to gather department data from the DEPARTMENTS table in the default PERSONNEL schema; however, it would not have been able to find the MANAGER_ID column and would have generated an error.

❹ PERSONNEL schema object qualification

Because the EMPLOYEES and JOB_HISTORY tables belong to the default PERSONNEL schema, you do not have to qualify schema object names with their schema name.

❺ Procedure section

Processing a multischema file is no different from processing a single-schema file. Only the naming you use is different.

The samples directory includes a C program that calls the module procedure shown in Example 20–1. Once compiled and linked with the SQL module object file, the program displays a list of employees belonging to each department in the corporate_data multischema database. The program uses the same statements and logic as it would to process a single-schema database.

You compile the SQL module and C program, link them to create the executable image, and run the image to display the employees assigned to each corporate department as you would any other SQL module language programs.

To see what SQL syntax in your module does not comply with the ANSI/ISO SQL standard, specify the FLAG_NONSTANDARD or `-std` qualifier when you compile your SQL module file. By default, SQL does not flag non-standard syntax. For other qualifiers that you can use on the SQL module processor command line, refer to the *Oracle Rdb7 SQL Reference Manual*.

20.3 Using Multischema Databases with the SQL Precompiler

This section describes the default settings that apply when you process a program with the SQL precompiler. It shows a precompiled program that uses multischema names. You can find a copy of the sql_msdb_pre.sc program in the samples directory.

20.3.1 Default Settings for the SQL Precompiler

When you use the SQL precompiler, you can change the default settings for alias, authorization identifier, catalog, and schema by using the DECLARE MODULE statement. If you do not use this statement, you must explicitly specify the attributes within the embedded program. Table 20–2 shows the implicit compile-time defaults for SQL compiler attributes.

Table 20–2 SQL Defaults for Compiler Attributes in Precompiled Programs

Compiler Attribute	Implicit Default
Alias	RDB$DBHANDLE
Authorization identifier	User name
Catalog	RDB$CATALOG
Schema	Authorization identifier

The following list explains the default settings for compiler attributes in embedded programs:

- Default alias

  The default alias for precompiled programs is RDB$DBHANDLE. Specifying a default database means that statements that refer to the default database do not need to use an alias. You need to assign an explicit alias when your program attaches to more than one database because only one database can have the RDB$DBHANDLE alias default.

- Default authorization identifier

  The default authorization identifier is the authorization identifier (user name) of the user who compiles an embedded program. SQL uses the authorization identifier for privilege checking. If you specify a RIGHTS clause in a DECLARE MODULE statement or a context file, SQL bases privilege checking on the default authorization identifier, in compliance with the ANSI/ISO SQL standard.

  Specifying RIGHTS RESTRICTED tells SQL to compare the user name of the person who executes an embedded program with the authorization identifier with which the program was compiled. SQL prevents any user other than the one who compiled the program from invoking that program. By default, SQL uses RIGHTS INVOKER and bases privilege checking on the user name of the user who invoked the program.

- Default catalog

  The default catalog, RDB$CATALOG, for precompiled programs is the same as the default in the SQL module language. The RDB$CATALOG, by default, contains the RDB$SCHEMA schema, which contains all Oracle Rdb system tables.

- Default schema

  The default schema for schema objects named in a precompiled program is taken from the authorization identifier (user name) of the user who most recently compiled the program. The RDB$SCHEMA is not the default schema. The authorization identifier of the user compiling the precompiled program is the default schema.

You can also specify the default catalog and schema in a context file.

If you do not specify the SCHEMA and CATALOG clauses in the DECLARE MODULE statement, you must qualify schema object names in precompiled programs with their catalog, schema, and optionally alias names, unless you are using only the default catalog and schema.

20.3.2 Using Multischema Naming in a Precompiled Program

Example 20–2 shows how to name schema objects in a precompiled C program.

Example 20–2 Using Multischema Names in a Precompiled C Program

```
/* ABSTRACT:
  This sample precompiled C program lists the employees assigned to
  each department defined in the sample multischema corporate_data
  database. */

#include <stdio.h>

main()
{
/* Declare return status variable for error handling. */

    int   SQLCODE;

/* Declare module to specify SQL standard dialect, including quoting rules. */

    EXEC SQL DECLARE MODULE SQL_MODULE
        DIALECT SQL92;          ❶

/* Variables for program use. */
    char  emp_id[6],
          emp_last_name[21],
          emp_first_name[21],
          dept_name[21],
          dept_code[5],
          dept_man_id[6];

/* Declare the corporate_data database. */

EXEC SQL DECLARE ALIAS FILENAME corporate_data;    ❷

/* Declare the cursor for the DEPARTMENTS table. */

EXEC SQL DECLARE DEPT_CURSOR TABLE CURSOR
         FOR SELECT
             DEPARTMENT_CODE, DEPARTMENT_NAME, MANAGER_ID FROM
             "RDB$DBHANDLE.ADMINISTRATION".ACCOUNTING.DEPARTMENTS ❸
         ORDER BY DEPARTMENT_CODE;
```

(continued on next page)

Example 20–2 (Cont.) Using Multischema Names in a Precompiled C Program

```
/* Declare the cursor for the EMPLOYEES table. */

EXEC SQL DECLARE EMP_CURSOR TABLE CURSOR
         FOR SELECT
             E.EMPLOYEE_ID, E.LAST_NAME,E.FIRST_NAME FROM
             "RDB$DBHANDLE.ADMINISTRATION".PERSONNEL.EMPLOYEES E,
             "RDB$DBHANDLE.ADMINISTRATION".PERSONNEL.JOB_HISTORY JH
         WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID
             AND JH.DEPARTMENT_CODE = :dept_code
            AND JH.JOB_END IS NULL
         ORDER BY E.LAST_NAME;

/* Print the report title. */

   printf("%20s DEPARTMENT EMPLOYEE LISTING\n\n"," ");

   /* Open the DEPARTMENTS cursor. */

      EXEC SQL OPEN DEPT_CURSOR;

   while (1)
   {

    /* Fetch the department data. */

      EXEC SQL FETCH DEPT_CURSOR INTO
                :dept_code,
                :dept_name,
                :dept_man_id;
   .
   .
   .
```

The following descriptions are keyed to the numbered items in Example 20–2:

❶ DECLARE MODULE statement

The DIALECT clause specifies, among other things, ANSI/ISO quoting rules. Because this program uses delimited identifiers, you must specify that you want to use ANSI/ISO SQL (SQL92) quoting rules. Otherwise, SQL incorrectly interprets the string inside the double quotation marks as a string literal and generates an error indicating that the use of double quotation marks (") for a string literal is a deprecated feature.

The default for the dialect or quoting rules is SQLV40, which interprets quoted strings as string literals.

❷ Declare alias

You do not have to specify the MULTISCHEMA IS ON clause when the database to which you want to attach was created as a multischema database. Use the MULTISCHEMA IS OFF clause in a DECLARE ALIAS statement when you want to view a multischema database in the single-schema mode. With the multischema mode disabled, you must refer to schema objects in SQL statements by their external names.

❸ Multischema object naming

If you do not specify the SCHEMA and CATALOG clauses in a precompiled program, you must qualify schema objects with catalog and schema names. Alias names are optional. In the name "RDB$HANDLE.ADMINISTRATION".ACCOUNTING.DEPARTMENTS, RDB$HANDLE is the alias, ADMINISTRATION is the catalog name, and ACCOUNTING is the name of the schema in which the DEPARTMENTS object resides. To avoid exceeding the three-level limit for naming, the alias and catalog name are together enclosed in quotation marks (") to appear as one level of naming.

To use quotation marks in this situation, you must use the DIALECT SQL92 or QUOTING RULES SQL92 clause in the DECLARE MODULE statement.

Because the example program contains only one attach, qualifying the object name with the RDB$DBHANDLE default alias is unnecessary.

You link the program as you would if it did not use multischema naming.

A

Using SQL International Options

Oracle Rdb provides a number of features that are useful when the data in the database is not in English or when the users' primary language is not English.

This appendix describes:

- The statements that control the format of data for input and display
- The specification of collating sequences to control sorting and comparisons
- The behavior of specific collating sequences
- The collating order used by all Oracle Rdb character sets

A.1 Controlling Input and Display Formats

You can modify the input format (and in many cases, display the format) for the following features:

- Radix point character

  Use the SET RADIX POINT and SHOW RADIX POINT statements.

- Digit separator character

  Use the SET DIGIT SEPARATOR and SHOW DIGIT SEPARATOR statements.

- Currency indicator character

  Use the SET CURRENCY SIGN and SHOW CURRENCY SIGN statements.

- Date and time format for DATE VMS data types

  OpenVMS VAX OpenVMS Alpha

  On OpenVMS, use the SET DATE FORMAT and SHOW DATE FORMAT statements. ♦

  Digital UNIX

  On Digital UNIX, you cannot modify the date and time format. ♦

- Language used for various input and displays, such as day names, month names, and so on

OpenVMS VAX OpenVMS Alpha

On OpenVMS, use the SET LANGUAGE and SHOW LANGUAGE statements. ♦

Digital UNIX

On Digital UNIX, use a locale setting. See Section A.1.1 for more information. ♦

The SET and SHOW statements related to these features are documented in the *Oracle Rdb7 SQL Reference Manual*.

A.1.1 Using Locale Settings on Digital UNIX

Digital UNIX

Locale refers to the international environment of a program. It defines localized behavior of that program at run time. This information can be established from one or more sets of locale databases.

Oracle Rdb complies with the POSIX standard in implementing locale support. It provides message files in subdirectories that are named using the language field and territory field. The following example shows a message subdirectory for the language Japanese (ja) and the territory Japan (JP):

```
/usr/lib/dbs/sql/vnn/lib/ja_JP/
```

The language field name is based on the POSIX iso639lang field; the territory field name is based on the POSIX iso3166terr field.

The default message subdirectory is named C_C. The following example shows the file specification for the SQL message file when no locale has been specified:

```
/usr/lib/dbs/sql/vnn/lib/C_C/sqlmsg.mdf
```

See your system manager to set up locales on Digital UNIX. ♦

A.2 Specifying Collating Sequences

By default, Oracle Rdb uses the ASCII collating sequence for all sorting and Boolean operations; however, you can override this default by specifying one of the following:

- A language-specific collating sequence supplied by the OpenVMS National character set utility (NCS)
- A user-defined collating sequence using NCS

You can specify a collating sequence for an entire database or for a particular domain, not for columns in tables. (If you define a column using a domain, however, the column inherits any collating sequence you specify for the domain.) The collating sequence determines how rows are sorted when the column is used as a sort key. The collating sequence also determines the behavior of Boolean operations that compare two columns or a column with a literal value.

Refer to the *Oracle Rdb7 SQL Reference Manual* for descriptions of all the SQL statements involved with collating sequences.

A.3 Using Collating Sequences

A predicate specifies a condition that SQL evaluates as true, false, or unknown. The following list describes the behavior of specific collating sequences with the CONTAINING, STARTING WITH, and LIKE predicates:

- CONTAINING

  The CONTAINING predicate is not sensitive to diacritical markings used in multinational character sets. Thus *a* matches *A*, *á*, *à*, *ä*, *Á*, *À*, *Â* and so on. (In Norwegian, *ä* is treated as if it is *ae*.)

  In Spanish, *ch* and *ll* are treated as if they are unique single letters. Thus, `CONTAINING 'C'` finds *C*, *c*, *ç*, and *Ç* but not *CH*, *ch*, *Ch* and *cH*.

- STARTING WITH

  Because the STARTING WITH operator is case sensitive, searches for uppercase multinational characters do not include lowercase multinational characters; the reverse is also true. For example, `STARTING WITH 'C'` retrieves a set of rows that is different from those retrieved by `STARTING WITH 'ç'`.

  In Spanish, *ch* and *ll* are treated as if they were individual, unique, single letters. For example, if a domain is defined with the collating sequence SPANISH, then `STARTING WITH 'c'` does not retrieve the word *char* but does retrieves the word *cat*.

- LIKE

  The behavior of the LIKE operator is the same as that of the STARTING WITH operator. Because the LIKE operator is case sensitive, searches for uppercase multinational characters do not include lowercase multinational characters; the reverse is also true. For example, `LIKE Ç` retrieves a different set of records than `LIKE ç`.

  In Spanish, the LIKE operator recognizes the Spanish combinations of *ch* and *ll* each as one character. For example, if a domain is defined with the collating sequence SPANISH, then `LIKE c` does not retrieve the word *char* but does retrieve the word *cat*.

The following list describes multinational character set behavior that applies to all predicates:

- The character *ñ* is always treated as different from the character *n*, in keeping with the practices of the Spanish language. In a similar manner,

the character *ç* is treated the same as the character *c*, in keeping with the practices of the French language.

- The character *ü* is treated the same as the character *u* for many languages, but is sorted between the characters *x* and *z* (with the y's) for Danish, Norwegian, and Finnish languages.

See the chapter on language and syntax elements in the *Oracle Rdb7 SQL Reference Manual* for any restrictions that these predicates might impose.

A.4 Collating Order for Oracle Rdb Character Sets

The standard collating sequences, in which characters are compared octet for octet, are the only collating sequences that are available for use with the character sets in Oracle Rdb.

Note

If a column uses multi-octet character sets, user-defined collating sequences do not work. For example, if you define DEC_KANJI as the database default character set and all the data values are ASCII characters, you can use the DEC Multinational character set (MCS) collating sequence. However, if a Kanji character appears in the data, collating results appear random and are ineffective when using the MCS collating sequence.

Table A–1 details the collating order that is used for the different character sets supported by Oracle Rdb.

Table A–1 Collating Order for Oracle Rdb Character Sets

Type of Character Set	Collating Sequence Definition
Single-octet character sets [1]	The collating sequence is user-defined or is the default MCS collating sequence.
Fixed multi-octet character sets [1]	The collating sequence is based on the numeric values specified by the KANJI, HANZI, HANYU, SICGCC, and KOREAN character sets.

[1] See the chapter on language and syntax elements in the *Oracle Rdb7 SQL Reference Manual* for a list of single-octet and multi-octet (fixed and mixed) character sets.

(continued on next page)

Table A–1 (Cont.) Collating Order for Oracle Rdb Character Sets

Type of Character Set	Collating Sequence Definition
Mixed multi-octet character sets [1]	The collating sequence of these character sets is based on the following (in order from lowest value): 1. ASCII characters The collating sequence for the ASCII character set is based on the numeric values defined in the MCS collating sequence table. 2. User-defined characters The collating sequence for user-defined characters is based on either the numeric value of the individual octets or as the values specified in the user-defined collating sequence table. 3. DEC_HANZI, DEC_KOREAN, DEC_HANYU, and DEC_SICGCC character sets The collating sequences for these character sets are based on the numeric values of the individual octets.

[1] See the chapter on language and syntax elements in the *Oracle Rdb7 SQL Reference Manual* for a list of single-octet and multi-octet (fixed and mixed) character sets.

The DEC_KANJI character set is different from the other mixed multi-octet character sets because it also includes the Hankaku (narrow) Katakana character set. The collating sequence of this character set is the following (in order from lowest value):

1. ASCII character set

   The collating sequence for the ASCII character set is based on the numeric values defined in the MCS collating sequence table.

2. Katakana character set

   The collating sequence for the Katakana character set is based on the numeric values provided by JIS X0201.

3. User-defined characters

   The collating sequence for user-defined characters is based on either the numeric value of the individual octets or the values specified in the user-defined collating sequence table.

4. Kanji character set

The collating sequence for the KANJI character set is based on the numeric values provided by JIS X0208.

Index

A

B

C

D

E

F

M

N

O

P

Q

R

S

T

U

V

W