AT&T 585-350-918
Issue 1
Comcode 407298702

PROGRAMMER'S GUIDE TO THE ORACLE PRECOMPILERS

VERSION 1.5
Part Number 5315-15-1292
December 1992

ORACLE®

Cooperative Server Technology for Transparent Data Sharing

Programmer's Guide to the ORACLE Precompilers
Version 1.5

Part No. 5315-15-1292
December, 1992

Contributing Author: Tom Portfolio

Contributors: Peter Clare, Ziyad Dahbour, Julia Espina, Steve Faris, Radhakrishna Hari, Nancy Ikeda, Ken Jacobs, Terry Olkin, Lee Osborne, Tim Smith, Gael Turk, Rao Vasireddy

PREFACE

This manual is a comprehensive user's guide and on-the-job reference to the ORACLE Precompilers. It explores a full range of topics—from underlying concepts to advanced programming techniques-and uses clear, hands-on examples to teach you all you need to know. It shows you step-by-step how to develop applications that use the powerful database language SQL to access and manipulate ORACLE data.

What This Guide Has to Offer

This guide shows you how the ORACLE Precompilers and embedded SQL can benefit your entire applications development process. It gives you the know-how to design and develop applications that harness the power of ORACLE. And, as quickly as possible, it helps you become proficient in writing embedded SQL programs.

An important feature of this guide is its emphasis on getting the most out of the ORACLE Precompilers and embedded SQL. To help you master these tools, this guide shows you all the “tricks of the trade” including ways to improve program performance. It also includes many program examples to better your understanding and demonstrate the usefulness of embedded SQL.

Note: You will not find installation instructions or system-specific information in this guide. For that kind of information about the ORACLE Precompilers, see the ORACLE installation or user’s guide for your system.

Who Should Read This Guide?

Anyone developing new applications or converting existing applications to run in the ORACLE environment will benefit from reading this guide. Written especially for programmers, this comprehensive treatment of the ORACLE Precompilers will also be of value to systems analysts, project managers, and others interested in embedded SQL applications.

To use this guide effectively, you need a working knowledge of the following subjects:

- applications programming in a high-level language such as C, COBOL, FORTRAN, Pascal, or PL/I
- the SQL database language
- ORACLE concepts and terminology

What's New in Versions 1.4 and 1.5?

Version 1.4 of the ORACLE Precompilers offers an array of improvements and new features. For example, now you can benefit from

- better performance due to
  - separate executable
  - terser, faster generated code
  - improved handling of cursors and host variables
  - bundled database calls
- close ANSI/ISO compliance
- datatype equivalencing
- an enhanced WHENEVER statement
- a new SQLCA debugging aid
- improved precompiled options
- a simpler precompiled command

With Version 1.5 of the ORACLE Precompilers, you can also benefit from

- full ANSI/ISO compliance
- calls to stored PL/SQL procedures
- the use of host arrays within PL/SQL blocks
- full support for ORACLE7 SQL

For more information, see Appendix A.

How This Guide Is Organized

This guide contains eleven chapters and six appendixes. Chapters 1 and 2 give you your bearings, then Chapters 3 to 11 lead you through the essentials of embedded SQL programming. A brief summary of what you will find in each chapter and appendix follows.

Chapter 1: Getting Acquainted
This chapter introduces you to the ORACLE Precompilers. You look at their role in developing application programs that manipulate ORACLE data and find out what they allow your applications to do.

Chapter 2: Learning the Basics
This chapter explains how embedded SQL programs do their work. You examine the special environment in which they operate, the impact of this environment on the design of your applications, the key concepts of embedded SQL programming, and the steps you take in developing an application.

Chapter 3: Meeting Program Requirements
This chapter shows you how to meet embedded SQL program requirements. You learn the embedded SQL coremands that declare variables, declare communications areas, and connect to an ORACLE database. You also learn about the ORACLE datatypes, National Language Support (NLS), data conversion, and how to take advantage of datatype equivalencing. In addition, this chapter shows you how to embed OCI calls in your program and how to develop X/Open applications.

Chapter 4 Using Embedded SQL
This chapter teaches you the essentials of embedded SQL programming. You learn how to use host variables, indicator variables, and the fundamental SQL commands that insert, update, select, and delete ORACLE data.

Chapter 5: Using Embedded PL/SQL
This chapter shows you how to improve performance by embedding PL/SQL transaction processing blocks in your program. You learn how to use PL/SQL with host variables, indicator variables, cursors, stored procedures, host arrays, and dynamic SQL.

Chapter 6: Defining and Controlling Transactions
This chapter describes transaction processing. You learn the basic techniques that safeguard the consistency of your database.

Chapter 7: Handling Runtime Errors
This chapter provides an in-depth discussion of error reporting and recovery. You learn how to detect and handle errors using the SQLCA and the WHENEVER statement. You also learn how to diagnose problems using the ORACA.

Chapter 8: Using Host Arrays
This chapter looks at using arrays to improve program performance. You learn how to manipulate ORACLE data using arrays, how to operate on all the elements of an array with a single SQL statement, and how to limit the number of array elements processed.

Chapter 9 Using Dynamic SQL
This chapter shows you how to take advantage of dynamic SQL. You are taught four methods-from simple to complex—for writing flexible programs that, among other things, let users build SQL statements interactively at run time.

Chapter 10 Writing SQL*Forms User Exits
This chapter focuses on writing user exits for your SQL*Forms applications. You learn how host-language subroutines can do certain jobs more quickly and easily than SQL*Forms.

Chapter 11: Running the ORACLE Precompilers
This chapter details the requirements for running an ORACLE Precompiled. You learn what happens during precompilation, how to issue the precompiled command, how to specify the many useful precompiled options, how to do conditional and separate precompilations, and how to embed OCI calls in your host program.

Appendix A New Features
This appendix highlights the improvements and new features introduced with Versions 1.4 and 1.5 of the ORACLE Precompilers.

Appendix B: Quick Reference to Embedded SQL
This appendix focuses on the differences between embedded and interactive SQL. You are given the purpose of each embedded SQL statement, its syntax diagram, parameter descriptions, usage notes, and one or more programming examples.

Appendix C ORACLE Reserved Words and Keywords
This appendix lists words that have a special meaning to ORACLE.

Appendix D: Error Messages
This appendix lists error messages you might see when running the ORACLE Precompilers. Also listed are error messages that the ORACLE runtime library might return to the SQLCA. For each error, the probable cause and corrective action are given.

Appendix E: Performance Tuning
This appendix shows you some simple, easy-to-apply methods for improving the performance of your applications.

Appendix F: Syntactic and Semantic Checking
This appendix shows you how to use the SQLCHECK option to control the type and extent of syntactic and semantic checking done on embedded SQL statements and PL/SQL blocks.

Conventions Used in This Guide

Important terms being defined for the first time are italicized. In discussions, UPPER CASE is used for database objects and SQL keywords, and *italicized lower case* is used for the names of variables, constants, and parameters.

To emphasize the role that embedded SQL statements play in a program, they are boldfaced in program examples. The program examples are written in pseudocode to avoid language-specific issues. Oriented to professional programmers, the pseudocode is somewhat formal. In the following example, a mail order for books is processed:

```
·· process book order
IF prepaid OR ( credit_rating > 2 ) THEN
    set order_total = 0; -- initialize order_total
    FOR EACH book
        IF catalog_number is valid THEN
            EXEC SQL SELECT QTY INTO : quantity_on_hand
                FROM STOCK
                WHERE CATNO = : catalog_number;
            IF quantity_on_hand > 0 THEN
                set line_price = catalog_price;
                set line_total = line_price * quantity;
                add line_total to order_total;
                subtract quantity from quantity_on_hand;
                create order_ line;
            ELSE
                create backorder;
            ENDIF;
        ELSE
            display 'Invalid catalog number';
        ENDIF;
    ENDFOR;
    create customer_order;
ELSE
    display 'Credit rating too low' ;
ENDIF;
```

This example has the following important features:

- Indenting reveals structure.
- Keywords (in upper case) make constructs and logic clear.
- Statements end with a semicolon (;).
- ANSI-style comments are used; they begin with two consecutive hyphens and extend to the end of a line.
- Parentheses help avoid ambiguity.

Notation

The following notation is used in this guide:

< >	Angle brackets enclose the name of a syntactic element.
[ ]	Square brackets enclose optional items.
{}	Braces enclose items only one of which is required.
|	A vertical bar separates options within brackets or braces.
...	An ellipsis shows that the preceding parameter can be repeated or that statements or clauses irrelevant to the discussion were left out.

ANSI/ISO Compliance

The Version 1.4 ORACLE Precompilers comply closely with the ANSI/ISO SQL standards. Compliance is governed by the MODE option, which is described in Chapter 11.

The Version 1.5 ORACLE Precompilers comply *fully* with the ANSI/ISO SQL standards. Compliance with these standards was certified by the National Institute of Standards and Technology (NIST).

To flag extensions to ANSI/ISO SQL, the Version 1.5 ORACLE Precompilers provide a PIPS Flagger and a new option named PIPS, which is described in Chapter 11. Again, compliance is governed by the MODE option. For more information about ANSI/ISO compliance, see Chapter L

Related Publications

The information in this guide is generic. Language-specific information for C, COBOL, FORTRAN, Pascal, and PL/I is provided in the supplements listed below. (The Pro*Ada Precompiled is fully documented in its own manual.)

The following publications are recommended for use with this guide:

- *ORACLE7 Server Application Developer's Guide,* Part No. 6695-70
- *ORACLE7* Server *Messages and Codes Manual,* Part No. 3605-70
- *ORACLE7* Server *SQL Language Reference Manual,* Part No. 778-70
- *PL/SQL User's Guide* and Reference, Part No. 800-20

- *Programmer's Guide to the ORACLE Call Interfaces,* Part No. 5411-70
- *Programmer's Guide to the Pro*Ada Precompiler,* Part No. 6434-15
- *Pro*C Supplement to the ORACLE Precompilers Guide,* Part No. 5452-15
- *Pro*COBOL Supplement to the ORACLE Precompilers Guide,* Part No. 5451-15
- *Pro*FORTRAN Supplement to the ORACLE Precompilers Guide,* Part No. 5453-15
- *Pro*Pascal Supplement to the ORACLE Precompilers Guide,* Part No. 5455-15
- *Pro*PL/I Supplement to the ORACLE Precompilers Guide,* Part No. 5454-15
- *SQL*Forms Designer's Reference,* Part No. 3304-30
- *SQL*Net User's Guide,* Part No. 3604-11
- *SQL*Plus User's Guide and Reference Manual,* Part No. 5142-31
- *Trusted ORACLE7 Server Administrator's Guide,* Part No. 6610-70

Occasionally, this guide refers you to other Oracle manuals for system-specific information. Typically, these manuals are called installation or user's guides, but their exact names vary by operating system and platform.

For a complete list of the publications that support ORACLE, see the *ORACLE Technical Publications Catalog,* Part No. 3903.

Your Comments Are Welcome

The Oracle Corporation technical staff values your comments. As we write and revise, your opinions are the most important input we receive. Please use the Reader's Comment Form at the back of this manual to tell us what you like and dislike about this Oracle publication. If the form has been used or you want to contact us, please call us at (415) 506-7000 or use the following address or FAX number:

ORACLE Precompilers Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

FAX: (415) 506-7200

CONTENTS

CHAPTER

1

GETTING ACQUAINTED

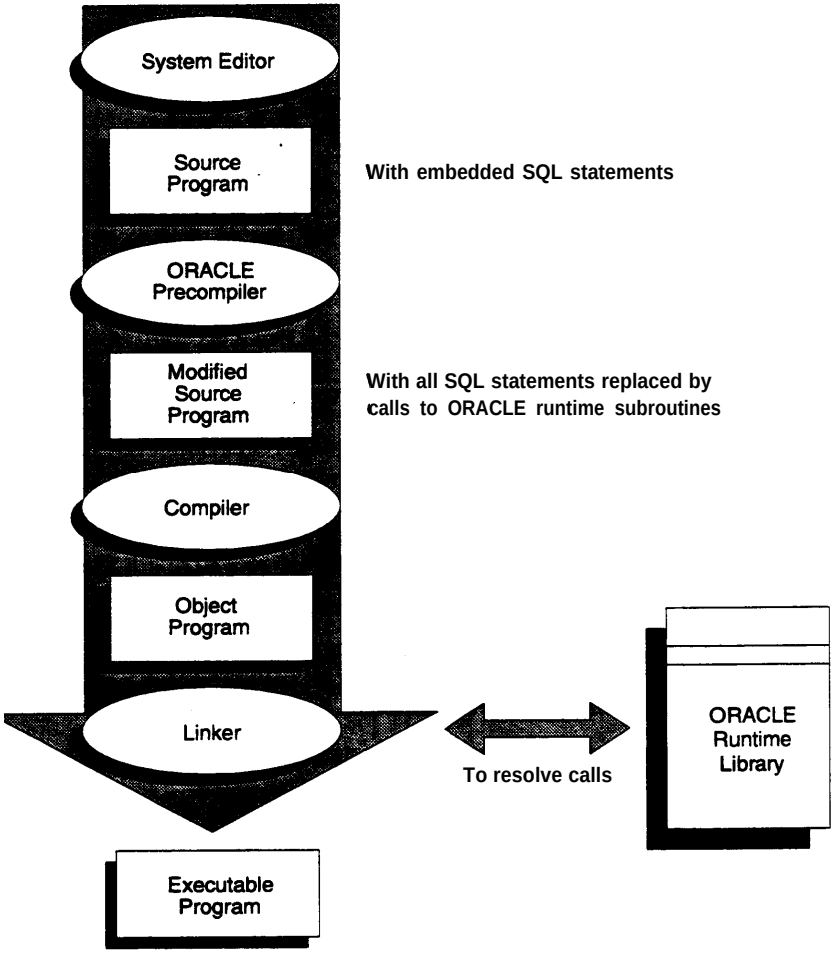
This chapter introduces you to the ORACLE Precompilers. You look at their role in developing application programs that manipulate ORACLE data and find out what they allow your applications to do. The following questions are answered:

- What is an ORACLE Precompiled?
- Why use an ORACLE Precompiled?
- Why use SQL?
- Why use PL/SQL?
- What do the ORACLE Precompilers offer?
- Do the ORACLE Precompilers meet industry standards?

What Is an ORACLE Precompiler?

An ORACLE Precompiler is a programming tool that allows you to embed SQL statements in a high-level source program. As Figure 1-1 shows, the precompiled accepts the source program as input, translates the embedded SQL statements into standard ORACLE runtime library calls, and generates a modified source program that you can compile, link, and execute in the usual way.

Figure 1-1
Embedded SQL
Program Development



Language Alternatives

Six ORACLE Precompilers are available (not on all systems); they support the following high-level languages:

- Ada
- C
- COBOL
- FORTRAN
- Pascal
- PL/I

Meant for different application areas and reflecting different design philosophies, these languages offer abroad range of programming solutions.

Note: This guide is supplemented by companion books devoted to C, COBOL, FORTRAN, Pascal, and PL/I. (See the section "Related Publications" in the Preface.) The Pro*Ada Precompiled is fully documented in the *Programmer's Guide to the Pro*Ada Precompiler*. From here on, references to the ORACLE Precompilers do *not* include the Pro*Ada Precompiler.

Why Use an ORACLE Precompiled?

The ORACLE Precompilers let you pack the power and flexibility of SQL into your application programs. You can use SQL in familiar high-level languages such as C, COBOL, FORTRAN, Pascal, and PL/I. A convenient, easy to use interface lets your application access ORACLE directly.

Unlike many application development tools, the ORACLE Precompilers let you create highly customized applications. For example, you can create user interfaces that incorporate the latest windowing and mouse technology. You can also create applications that run in the background without the need for user interaction.

Furthermore, the ORACLE Precompilers help you fine-tune your applications. They allow close monitoring of resource use, SQL statement execution, and various runtime indicators. With this information, you can tweak program parameters for maximum performance.

Although precompiling adds a step to the application development process, it saves time because the precompiled, not you, translates each embedded SQL statement into several native-language ORACLE calls.

Why Use SQL?

If you want to access and manipulate ORACLE data, you need SQL. Whether you use SQL interactively through SQL*PlusS or embedded in an application program depends on the job at hand. If the job requires the procedural processing power of C, COBOL, FORTRAN, Pascal, or PL/I, or must be done on a regular basis, use embedded SQL.

SQL has become the database language of choice because it is flexible, powerful, and easy to learn. Being non-procedural, it lets you specify what you want done without specifying how to do it. A few English-like statements make it easy to manipulate ORACLE data one row or many rows at a time.

You can execute any SQL (not SQL*Plus) statement from an application program. For example, you can

- CREATE, ALTER, and DROP database tables dynamically
- SELECT, INSERT, UPDATE, and DELETE rows of data
- COMMIT or ROLLBACK transactions

Before embedding SQL statements in an application program, you can test them interactively using SQL*Plus. Usually, only minor changes are required to switch from interactive to embedded SQL.

Why Use PL/SQL?

An extension to SQL, PL/SQL is a transaction processing language that supports procedural constructs, variable declarations, and robust error handling. Within the same PL/SQL block, you can use SQL and all the PL/SQL extensions.

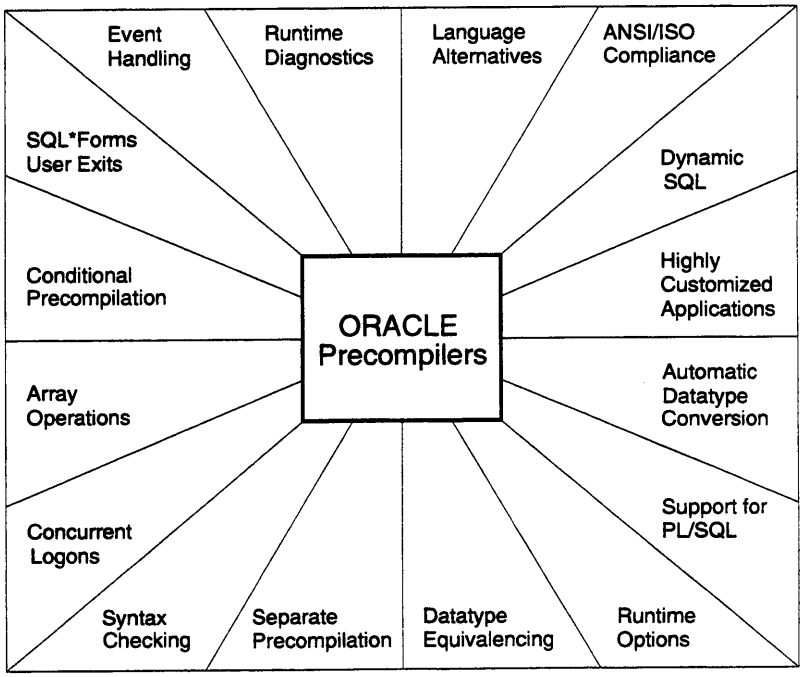
The main advantage of embedded PL/SQL is better performance. Unlike SQL, PL/SQL allows you to group SQL statements logically and send them to ORACLE in a block rather than one by one. This reduces network traffic and processing overhead.

For more information about PL/SQL, including how to embed it in an application program, see Chapter 5, “Using Embedded PL/SQL.”

What Do the ORACLE Precompilers Offer?

As Figure 1-2 shows, the ORACLE Precompilers offer many features and benefits, which help you to develop effective, reliable applications.

Figure 1-2
Features and Benefits



For example, the ORACLE Precompilers allow you to

- write your application in any of six popular high-level programming languages
- follow the ANSI standard for embedding SQL statements in a high-level language
- take advantage of dynamic SQL, an advanced programming technique that lets your program accept or build any valid SQL statement at run time
- design and develop highly customized applications
- automatically convert between ORACLE internal datatypes and high-level language datatypes
- improve performance by embedding PL/SQL transaction processing blocks in your application program
- specify useful precompiled options inline and on the command line and change their values during precompilation

- use datatype equivalencing to control the way ORACLE interprets input data and formats output data
- separately precompiled several program modules, then link them into one executable program
- completely check the syntax and semantics of embedded SQL data manipulation statements and PL/SQL blocks
- concurrently access ORACLE databases on multiple nodes using SQL*Net
- use arrays as input and output program variables
- conditionally precompiled sections of code in your host program so that it can run in different environments
- directly interface with SQL*Forms via user exits written in a high-level language
- handle errors and warnings with the SQL Communications Area (SQLCA) and the WHENEVER statement
- use an enhanced set of diagnostics provided by the ORACLE Communications Area (ORACA)

To sum it up, the ORACLE Recompilers are full-featured tools that support a professional approach to embedded SQL programming.

Do the ORACLE Precompilers Meet Industry Standards?

SQL has become the standard language for relational database management systems. This section describes how the ORACLE Precompilers conform to SQL standards established by the following organizations:

- American National Standards Institute (ANSI)
- International Standards Organization (ISO)
- U.S. National Institute of Standards and Technology (NIST)

These organizations have adopted SQL as defined in the following publications:

- ANSI standard X3.135-1989, *Database Language SQL with Integrity Enhancement*
- ANSI standard X3.168-1989, *Database Language Embedded SQL*
- ISO standard 9075-1989, *Database Language SQL with Integrity Enhancement*
- NIST standard FIPS PUB 127-1, *Database Language SQL* (FIPS is *an* acronym for Federal Information Processing Standards)

Requirements

ANSI standard X3.135-1989 requires a claim of compliance to state the

- type of compliance (Level 1 or Level 2)
- implemented facilities (host language interfaces)

Level 2 is the full standard. It includes the following features, among others, which are not required for Level 1 compliance:

- positioned UPDATE and DELETE statements, which use the CURRENT OF <cursor_name> clause
- indicator variables, which "indicate" the value or status of associated host variables

ANSI standard X3.168-1989 specifies the syntax and semantics for embedding SQL statements in application programs written in a standard programming language such as Ada, C, COBOL, FORTRAN, Pascal, or PL/I.

ISO standard 9075-1989 fully adopts the ANSI standards.

NIST standard FIPS PUB 127-1, which applies to RDBMS software acquired for federal use, also adopts the ANSI standards. In addition, it specifies minimum sizing parameters for database constructs and requires a "FIPS Flagger" to identify ANSI extensions.

For copies of the ANSI standards, write to

American National Standards Institute
1430 Broadway
New York, NY 10018
USA

For a copy of the ISO standard, write to the national standards office of any ISO participant. For a copy of the NIST standard, write to

National Technical Information Service
U.S. Department of Commerce
Springfield, VA 22161
USA

Compliance

Under ORACLE7, the ORACLE Precompilers comply 100% with the ANSI/ISO standard. They conform to Level 2 of X3.135-1989 and support the following host languages:

- Ada
- C
- COBOL
- FORTRAN
- Pascal
- PL/I

However, supported host languages vary by operating system and platform.

The ORACLE Precompilers also comply 100% with the NIST standard. They provide a FIPS Flagger and an option named PIPS, which enables the FIPS Flagger. For more information, see the section 'TIPS Flagger" later in this chapter.

Certification

NIST tested the ORACLE Precompilers for ANSI Level 2 compliance using the *SQL Test Suite,* which consists of nearly 300 test programs. Specifically, the programs tested for conformance to the C, COBOL, FORTRAN, and Pascal embedded SQL standards. The result the ORACLE Precompilers were certified 100% ANSI-compliant.

For more information about the tests, write to

National Computer Systems Laboratory
Attn: Software Standards Testing Program
National Institute of Standards and Technology
Gaithersburg, MD 20899
USA

FIPS Flagger

According to PIPS PUB 127-1, "an implementation that provides additional facilities not specified by this standard shall also provide an option to flag nonconforming SQL language or conforming SQL language that may be processed in a nonconforming manner." To meet this requirement, the ORACLE Precompilers provide the *FIPS Flagger*, *which* flags ANSI extensions. An extension is any SQL element that violates ANSI format or syntax rules, except privilege enforcement rules. For a list of Oracle extensions to standard SQL, see the ORACLE7 Server *SQL Language Reference Manual.*

You can use the FIPS Flagger to identify

- nonconforming SQL elements that might have to be modified if you move the application to a conforming environment
- conforming SQL elements that might behave differently in another processing environment

Thus, the FIPS Flagger helps you develop portable applications.

FIPS Option

A new option named FIPS governs the FIPS Flagger. To enable the FIPS Flagger, you specify FIPS=YES inline or on the command line. For more information about the FIPS option, see the section "Using the Precompiled Options" in Chapter 11.

CHAPTER

2

LEARNING THE BASICS

This chapter explains how embedded SQL programs do their work. You examine the special environment in which they operate and the impact of this environment on the design of your applications.

After covering the key concepts of embedded SQL programming and the steps you take in developing an application, this chapter uses a simple program to illustrate the main points.

Key Concepts of Embedded SQL Programming

This section lays the conceptual foundation on which later chapters build. It discusses the following subjects:

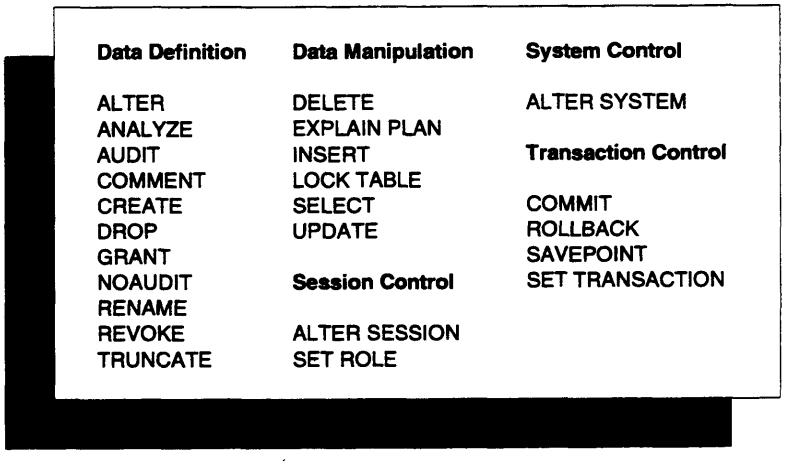
- embedded SQL statements
- executable versus declarative SQL statements
- static versus dynamic SQL statements
- embedded PL/SQL blocks
- host and indicator variables
- ORACLE datatypes
- arrays
- datatype equivalencing
- private SQL areas, cursors, and active sets
- transactions
- errors and warnings

Embedded SQL Statements

The term *embedded SQL* refers to SQL statements placed within an application program. Because it houses the SQL statements, the application program is called a *host program*, and the language in which it is written is called the *host language*. For example, the Pro*C Precompiled allows you to embed SQL statements in a C host program.

Figure 2-1 shows all the interactive SQL statements your application program can execute.

Figure 2-1
SQL Statements in an Application Progra



For example, to manipulate and query ORACLE data, you use the INSERT, UPDATE, DELETE, and SELECT statements. INSERT adds rows of data to database tables, UPDATE modifies rows, DELETE removes unwanted rows, and SELECT retrieves rows that meet your search condition.

The Version 1.5 ORACLE Precompilers support all the ORACLE7 SQL statements. For example, the powerful SET ROLE statement lets you dynamically manage database privileges. A *role is* a named group of related system and /or object privileges granted to users or other roles. Role definitions are stored in the ORACLE data dictionary. Your applications can use the SET ROLE statement to enable and disable roles as needed.

Only SQL statements-not SQL*Plus statements-are valid in an application program. (SQL*Plus has additional statements for setting environment parameters, editing, and report formatting.)

Executable versus Declarative Statements

Embedded SQL includes all the interactive SQL statements plus others that allow you to transfer data between ORACLE and a host program. There are two types of embedded SQL statements: executable and *declarative.* Executable statements result in calls to the runtime library SQLLIB. You use them to connect to ORACLE, to define, query, and manipulate ORACLE data, to control access to ORACLE data, and to process transactions. They can be placed wherever host-language executable statements can be placed.

Declarative statements, on the other hand, do not result in calls to SQLLIB and do not operate on ORACLE data. You use them to declare ORACLE objects, communications areas, and SQL variables. They can be placed wherever host-language declarations can be placed.

Figure 2-2 groups the various embedded SQL statements for you.

Figure 2-2
Embedded SQL Statements Grouped by Type

Declarative

ARRAYLEN*	To use host arrays with PL/SQL
BEGIN DECLARE SECTION * END DECLARE SECTION *	To declare host variables
DECLARE *	To name ORACLE objects
INCLUDE *	To copy in files
TYPE * VAR *	To equivalence datatypes
WHENEVER *	To handle runtime errors

Executable

ALTER ANALYZE AUDIT COMMENT CONNECT * CREATE DROP GRANT NOAUDIT RENAME REVOKE TRUNCATE	To define and control access to ORACLE data
CLOSE * DELETE EXPLAIN PLAN FETCH * INSERT LOCK TABLE OPEN * SELECT UPDATE	To retrieve and manipulate ORACLE data
COMMIT ROLLBACK SAVEPOINT SET TRANSACTION	To process transactions
DESCRIBE * EXECUTE * PREPARE *	To use dynamic SQL
ALTER SESSION SET ROLE	To control sessions

* Has no interactive counterpart

Embedded SQL Syntax

In your application program, you can freely intermix SQL statements with host-language statements and use host-language variables in SQL statements. The only special requirement for building SQL statements into your host program is that you begin them with the keywords EXEC SQL and end them with the SQL statement terminator for your host language. The precompiled translates all EXEC SQL statements into calls to the runtime library SQLLIB.

Most embedded SQL statements differ from their interactive counterparts only through the adding of a new clause or the use of program variables. In the following example, interactive and embedded ROLLBACK statements are compared:

```
ROLLBACK WORK ;
EXEC SQL ROLLBACK WORK;
```

For a summary of embedded SQL syntax, see Appendix B.

Static versus Dynamic SQL Statements

Most application programs are designed to process static SQL statements and fixed transactions. In this case, you know the makeup of each SQL statement and transaction before run time; that is, you know which SQL commands will be issued, which database tables might be changed, which columns will be updated, and so on.

However, some applications might be required to accept and process any valid SQL statement at run time. So, you might not know until run time all the SQL commands, database tables, and columns involved.

Dynamic SQL is an advanced programmingg technique that lets your program accept or build SQL statements at run time and take explicit control over datatype conversion.

Embedded PL/SQL Blocks

The ORACLE Precompilers treat a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in an application program that you can place a SQL statement. To embed PL/SQL in your host program, you simply declare the variables to be shared with PL/SQL and bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END-EXEC.

From embedded PL/SQL blocks, you can manipulate ORACLE data flexibly and safely because PL/SQL supports all SQL data manipulation and transaction processing commands. For more information about PL/SQL, see Chapter 5, “Using Embedded PL/SQL.”

Host and Indicator Variables

Host variables are the key to communication between ORACLE and your program. A *host variable* is a scalar or array variable declared in the host language and shared with ORACLE, meaning that both your program and ORACLE can reference its value.

Your program uses *input* host variables to pass data to ORACLE. ORACLE uses output host variables to pass data and status information to your program. The program assigns values to input host variables; ORACLE assigns values to output host variables.

Host variables can be used anywhere an expression can be used, but, in SQL statements, must be prefixed with a colon (:) to set them apart from ORACLE objects.

You can associate any host variable with an optional indicator variable. An *indicator variable* is an integer variable that "indicates" the value or condition of its host variable. You use indicator variables to assign nulls to input host variables and to detect nulls or truncated values in output host variables. A *null is* a missing, unknown, or inapplicable value.

In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable.

ORACLE Datatypes

Typically, a host program inputs data to ORACLE, and ORACLE outputs data to the program. ORACLE stores input data in database tables and stores output data in program host variables. To store a data item, ORACLE must know its *datatype,* which specifies a storage format and valid range of values.

ORACLE recognizes two kinds of datatypes: *internal* and *external.* Internal datatypes specify how ORACLE stores data in database columns. ORACLE also uses internal datatypes to represent database pseudocolumns, which return specific data items but are not actual columns in a table.

External datatypes specify how data is stored in host variables. When your host program inputs data to ORACLE, if necessary, ORACLE converts between the external datatype of the input host variable and the internal datatype of the target database column. When ORACLE outputs data to your host program, if necessary, ORACLE converts between the internal datatype of the source database column and the external datatype of the output host variable.

Arrays

The ORACLE Precompilers let you define array host variables called *host arrays* and operate on them with a single SQL statement. Using the array SELECT, FETCH, DELETE, INSERT, and UPDATE statements, you can query and manipulate large volumes of data with ease.

Datatype Equivalencing

The ORACLE Precompilers add flexibility to your applications by letting you *equivalence* datatypes. That means you can customize the way ORACLE interprets input data and formats output data.

On a variable-by-variable basis, you can equivalence supported host language datatypes to the ORACLE external datatypes. You can also equivalence user-defined datatypes (available in C and Pascal) to ORACLE external datatypes.

Private SQL Areas, Cursors, and Active Sets

To process a SQL statement, ORACLE opens a work area called a *private SQL area. The* private SQL area stores information needed to execute the SQL statement. An identifier called a *cursor* lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

For static SQL statements, there are two types of cursors: *implicit* and explicit. ORACLE implicitly declares a cursor for all data definition and data manipulation statements, including SELECT statements (queries) that return only one row. However, for queries that return more than one row, to process beyond the first row, you must explicitly declare a cursor (or use host arrays).

The set of rows returned is called the *active set;* its size depends on how many rows meet the query search condition. You use an explicit cursor to identify the row currently being processed, called the *current row.*

Imagine the set of rows being returned to a terminal screen. A screen cursor can point to the first row to be processed, then the next row, and so on. In the same way, an explicit cursor "points" to the current row in the active set. This allows your program to process the rows one at a time.

Transactions

A *transaction* is a series of logically related SQL statements (two UPDATEs that credit one bank account and debit another, for example) that ORACLE treats as a unit, so that all changes brought about by the statements are made permanent or undone at the same time.

All the data manipulation statements executed since the last data definition, COMMIT, or ROLLBACK statement was executed makeup the current transaction.

To help ensure the consistency of your database, the ORACLE Precompilers let you define transactions using the COMMIT, ROLLBACK, and SAVEPOINT statements.

COMMIT makes permanent any changes made during the current transaction. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction; used with ROLLBACK, it undoes part of a transaction.

Errors and Warnings

When you execute an embedded SQL statement, it either succeeds or fails, and might result in an error or warning. You need a way to handle these results. The ORACLE Precompilers provide two error handling mechanisms: the SQL Communications Area (SQLCA) and the WHENEVER statement.

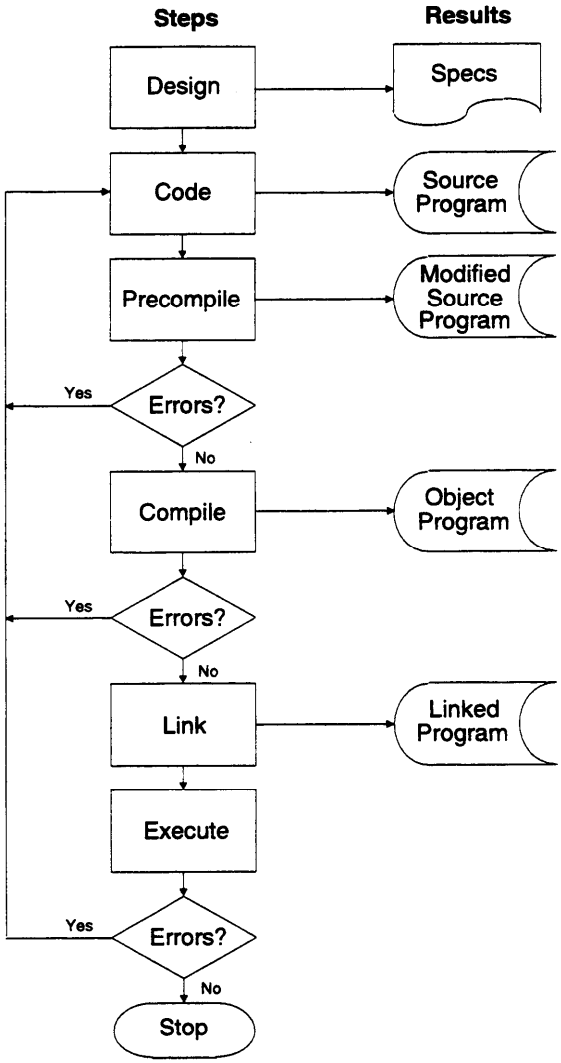
The SQLCA is a data structure copied or hardcoded into your host program. It defines program variables used by ORACLE to pass runtime status information to the program. With the SQLCA, you can take different actions based on feedback from ORACLE about work just attempted. For example, you can check to see if a DELETE statement succeeded and if so, how many rows were deleted.

With the WHENEVER statement, you can specify actions to be taken automatically when ORACLE detects an error or warning condition. These actions include continuing with the next statement, calling a subroutine, branching to a labeled statement, or stopping.

Steps in Developing an Embedded SQL Application

Figure 2-3 walks you through the embedded SQL application development process.

Figure 2-3
Embedded SQL
Application Development Process



As you can see, precompiling results in a modified source file that can be compiled normally. Though precompiling adds a step to the traditional development process, that step is well worth taking because it lets you write very flexible applications.

A Program Example

A good way to get acquainted with embedded SQL is to look at a program example. (Program examples are written in a pseudocode, which is described in the Preface.)

Handling errors with the WHENEVER statement, the following program connects to ORACLE, prompts the user for an employee number, queries the database for the employee's name, salary, and commission, then displays the information and exits.

```
-- declare host and indicator variables
EXEC SQL BEGIN DECLARE SECTION;
    username    CHARACTER (20);
    password    CHARACTER(20);
    emp_number  INTEGER;
    emp_name    CHARACTER (10);
    salary      REAL;
    commission  REAL;
    ind_comm    SMALLINT;     -- indicator variable
EXEC SQL END DECLARE SECTION;

-- copy in the SQL Communications Area
EXEC SQL INCLUDE SQLCA;

display  ' Username? ' ;
read username;
display  ' Password? ' ;
read password;

-- handle processing errors
EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

-- log on to ORACLE
EXEC SQL CONNECT :username IDENTIFIED BY : password;

display  'Connected to ORACLE';

display  'Employee number? ';
read  emp_number;

-- query database for employee's name, salary, and commission
-- and assign values to host variables
EXEC SQL SELECT ENAME, SAL, COMM
    INTO  :emp_name,  :salary,  :commission:ind_ca
    FROM  EMP
    WHERE EMPNO =  :emp_number;
```

```
display  'Employee    Salary    Commission' ;
display  '--------    ------    ----------';

-- display employee's name,  salary, and commission (if not null)
IF ind_comm = -1 THEN    -- commission is null
    display emp_name, salary, 'Not applicable' ;
ELSE
    display emp_name,  salary,  commission;
ENDIF;

-- release resources and log off the database
EXEC SQL COMMIT WORK RELEASE;
display  'Have a good day';
exit program;

sqlerror:
    -- avoid an infinite loop if the rollback results in an error
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    -- release resources and log off the database
    EXEC SQL ROLLBACK WORK RELEASE;
    display  'Processing error';
    exit program with an error;
```

CHAPTER

3

MEETING PROGRAM REQUIREMENTS

Passing data between ORACLE and your application program requires host variables, datatype conversions, event handling, and access to ORACLE. This chapter teaches you how to meet these requirements. You learn the embedded SQL commands that declare variables, declare communications areas, and connect to an ORACLE database. You also learn about the ORACLE datatypes, National Language Support (NLS), data conversion, and how to take advantage of datatype equivalencing. The final two sections show you how to embed OCI calls in your program and how to develop X/Open applications.

The Declare Section

You must declare all program variables to be used in SQL statements (that is, all host variables) in the Declare Section. If you use an undeclared host variable in a SQL statement, an error message is issued at precompiled time. Error messages are listed in Appendix D.

The Declare Section begins with the statement

```
EXEC SQL BEGIN DECLARE SECTION;
```

and ends with the statement

```
EXEC SQL END DECLARE SECTION;
```

Between these two statements only the following are allowed:

- host and indicator variable declarations
- EXEC SQL INCLUDE statements
- host-language comments

With most (but not all) host languages, multiple Declare Sections are allowed per precompiled unit. Furthermore, a host program can contain several independently precompiled units.

You can define Declare Sections locally or globally. But, you must define at least one Declare Section even if it holds no declarations.

An Example

In the following example, four host variables are declared for use later in the program

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_number    INTEGER;
    emp_name      CHARACTER (10);
    salary        REAL;
    commission    REAL;
EXEC SQL END DECLARE SECTION;
```

For more information about declaring host variables, see the section "Declaring and Referencing Host Variables" later in this chapter.

Using the INCLUDE Statement

The INCLUDE statement lets you copy files into your host program. It is similar to the C #include directive or the COBOL COPY command. An example follows:

```
-- copy in the SQLCA file
EXEC SQL INCLUDE SQLCA ;
```

When you precompiled your program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

You can INCLUDE any file. If a file contains embedded SQL, you *must* INCLUDE it because only INCLUDEd files are precompiled. If you do not specify a file extension, the precompiled assumes the default extension for source files. The default extension is system-dependent. Check the ORACLE installation or user's guide for your system.

You can set a directory path for INCLUDEd files by specifying, inline or on the command line, the precompiled option

```
INCLUDE=path
```

where *path* defaults to the current directory. (In this context, a *directoy is an* index of file locations.)

The precompiled searches first in the current directory, then in the directory specified by INCLUDE, and finally in a directory for standard INCLUDE files. So, you need not specify a directory path for standard files such as the SQLCA and ORACA. You must still use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory.

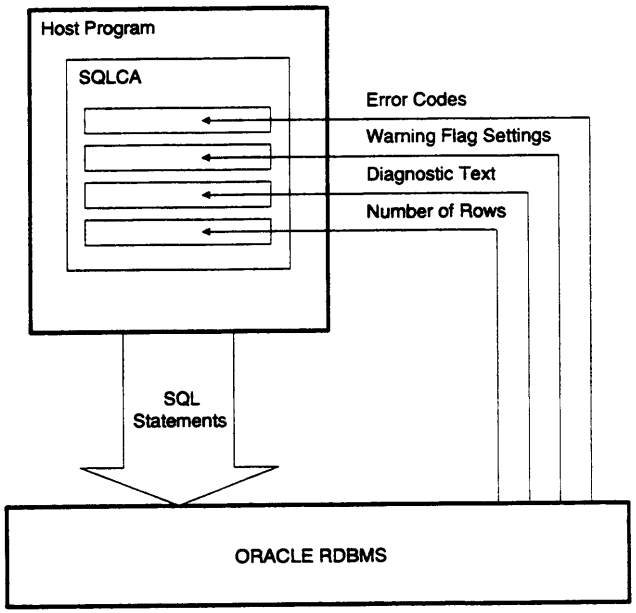
If your operating system is case-sensitive (like UNIX, for example), be sure to specify the same upper/lower case filename under which the file is stored.

The syntax for specifying a directory path is system-specific. Check the ORACLE installation or user's guide for your system.

SQL Communications Area (SQLCA)

The SQLCA is a data structure that provides for diagnostic checking and event handling. At run time, the SQLCA holds status information passed to your program by ORACLE. After executing a SQL statement, ORACLE sets the SQLCA variables to indicate the outcome, as illustrated in Figure 3-1.

Figure 3-1
Updating the SQLCA



Thus, you can check to see if an INSERT, UPDATE, or DELETE statement succeeded and if so, how many rows were affected. Or, if the statement failed, you can get more information about what happened.

When MODE= {ANSI13 | ORACLE}, you must declare the SQLCA by hardcoding it or by copying it into your program with the INCLUDE statement. The section "Using the SQL Communications Area (SQLCA)" in Chapter 7 shows you how to declare and use the SQLCA.

When MODE={ANSI | ANSI14}, declaring the SQLCA is optional. However, you must declare a status variable named SQLCODE. For more information, see the section "Declaring SQLCODE" in Chapter 7.

The ORACLE Datatypes

Recall from Chapter 2 that ORACLE recognizes two kinds of datatypes: internal and external. Internal datatypes specify how ORACLE stores data in database columns. ORACLE also uses internal datatypes to represent database pseudocolumns. External datatypes specify how data is stored in host variables.

At precompiled time, each host variable in the Declare Section is associated with an external datatype code. At run time, the datatype code of every host variable used in a SQL statement is passed to ORACLE. ORACLE uses the codes to convert between internal and external datatypes.

Note: You can override default datatype conversions by using dynamic SQL Method 4 or datatype equivalencing. For information about dynamic SQL Method 4, see the section "Using Method 4" in Chapter 9. Datatype equivalencing is discussed later in this chapter.

Internal Datatypes

For database columns and pseudocolumns, ORACLE uses the following internal datatypes, which were chosen for their efficiency

Name	*Code*	*Description*
VARCHAR2	1	≤ 2000-byte, variable-length string
NUMBER	2	fixed or floating point number
LONG	8	≤ 2147483647-byte, fixed-length string
ROWID	11	fixed-length binary number
DATE	12	7-byte, fixed-length date/time value
RAW	23	≤ 255-byte, fixed-length binary data
LONG RAW	24	≤ 2147483647-byte, fixed-length binary data
CHAR	96	≤ 255-byte, fixed-length string
MLSLABEL	105	≤ 5-byte, variable-length binary label

These internal datatypes can be quite different from host language datatypes. For example, the NUMBER datatype was designed for portability, precision (no rounding error), and correct collating. No host language has an equivalent datatype.

Brief descriptions of the internal datatypes follow. For more information, see the ORACLE7 Server *SQL Language Reference Manual.*

VARCHAR2

You use the VARCHAR2 datatype to store variable-length character strings. How the strings are represented internally depends on the database character set, which might be 7-bit ASCII or EBCDIC Code Page 500 for example.

The maximum width of a VARCHAR2 database column is 2000 bytes. To define a VARCHAR2 column, you use the syntax

```
column_name VARCHAR2 (maximum_width)
```

where *maximum_width* is an integer literal in the range 1..2000.

You specify the maximum width of a VARCHAR2 *(n)* column in bytes, not characters. So, if a VARCHAR2 *(n)* column stores multibyte characters, its maximum width is less than *n* characters.

NUMBER

You use the NUMBER datatype to store fixed or floating point numbers of virtually any size. You can specify *precision,* which is the total number of digits, and *scale,* which determines where rounding occurs.

The maximum precision of a NUMBER value is 38; the magnitude range is 1.0E-129 to 9.99E125. Scale can range from -84 to 127. For example, a scale of -3 means the number is rounded to the nearest thousand (3456 becomes 3000). A scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46).

When you specify precision and scale, ORACLE does extra integrity checks before storing the data. If you do not specify precision and scale, they default to 38 and 10, respectively.

LONG

You use the LONG datatype to store variable-length character strings. LONG columns can store text, arrays of characters, or even short documents. The LONG datatype is like the VARCHAR2 datatype, except that the maximum width of a LONG column is 2147483647 (2^{31}- 1) bytes or two gigabytes.

Restrictions: You can use LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, function calls, or . certain SQL clauses such as WHERE, GROUP BY, and CONNECT BY. Only one LONG column is allowed per database table, and that column cannot be indexed.

ROWID	Internally, every table in an ORACLE database has a pseudocolumn named ROWID, which stores 6-byte values called *rowids.* Rowids uniquely identify rows and provide the fastest way to access particular rows.
DATE	You use the DATE datatype to store dates and times in 7-byte, fixed-length fields. The date portion defaults to the first day of the current month; the time portion defaults to midnight.
RAW	You use the RAW datatype to store binary data or byte strings (a sequence of graphics characters, for example). RAW data is not interpreted by ORACLE. The RAW datatype takes a required parameter that lets you specify a maximum width up to 255 bytes. The syntax follows `RAW (maximum_width)` You cannot use a constant or variable to specify the maximum width; you must use an integer literal. RAW data is like CHAR data, except that ORACLE assumes nothing about the meaning of RAW data and does no character set conversions (from 7-bit ASCII to EBCDIC Code Page 500 for example) when you transmit RAW data from one system to another.
LONG RAW	You use the LONG RAW datatype to store binary data or byte strings. The maximum width of a LONG RAW column is 2147483647 bytes or two gigabytes. LONG RAW data is like LONG data, except that ORACLE assumes nothing about the meaning of LONG RAW data and does no character set conversions when you transmit LONG RAW data from one system to another. The restrictions that apply to LONG data also apply to LONG RAW data.

CHAR

You use the CHAR datatype to store fixed-length character data. How the data is represented internally depends on the database character set.

The CHAR datatype takes an optional parameter that lets you specify a maximum width up to 255 bytes. The syntax follows:

```
CHAR [(maximum_width)]
```

You cannot use a constant or variable to specify the maximum width; you must use an integer literal. If you do not specify the maximum width, it defaults to 1.

Remember, you specify the maximum width of a CHAR *(n)* column in bytes, not characters. So, if a CHAR *(n)* column stores multibyte characters, its maximum width is less than *n* characters.

MLSLABEL

With Trusted ORACLE, you use the MLSLABEL datatype to store variable-length, binary operating system labels. Trusted ORACLE uses labels to control access to data. For more information, see the Trusted ORACLE7 *Server Administrator's Guide.*

You can use the MLSLABEL datatype to define a database column. However, with standard ORACLE, such columns can store only nulls. With Trusted ORACLE, you can insert any valid operating system label into a column of type MLSLABEL. If the label is in text format, Trusted ORACLE converts it to a binary value automatically. The text string can be up to 255 bytes long. However, the internal length of an MLSLABEL value is between 2 and 5 bytes.

With Trusted ORACLE, you can also select values from a MLSLABEL column into a character variable. Trusted ORACLE converts the internal binary value to a VARCHAR2 value automatically.

SQL Pseudocolumns and Functions

SQL recognizes the following pseudocolumns and parameterless functions, which return specific data items:

Pseudocolumn	*Corresponding Internal Datatype*	*Code*
NEXTVAL	NUMBER	2
CURRVAL	NUMBER	2
ROWNUM	NUMBER	2
LEVEL	NUMBER	2
ROWID	ROWTD	11

Fund-on	*Corresponding Internal Datatype*	*Code*
USER	VARCHAR2	1
UID	NUMBER	2
SYSDATE	DATE	12

Pseudocolumns are not actual columns in a table. However, pseudocolumns are treated like columns, so their values must be SELECTed from a table. Sometimes it is convenient to SELECT pseudocolumn values from a dummy table.

You can reference SQL pseudocolumns and functions in SELECT, INSERT, UPDATE, and DELETE statements. In the following example, you use SYSDATE to compute the number of months since an employee was hired:

```
EXEC SQL SELECT MONTHS_BETWEEN (SYSDATE , HIREDATE)
    INTO : months_of_service
    FROM EMP
    WHERE EMPNO =  : emp_number;
```

Brief descriptions of the SQL pseudocolumns and functions follow. For more information, see the ORACLE7 Server *SQL Language Reference Manual.*

NEXTVAL returns the next number in a specified sequence. After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. In the following example, you use the sequence named *partno* to assign part numbers:

```
EXEC SQL INSERT INTO PARTS
    VALUES (partno.NEXTVAL, :description, : quantity, :price) ;
```

If a transaction generates a sequence number, the sequence is incremented when you COMMIT or ROLLBACK the transaction. A reference to NEXTVAL stores the current sequence number in CURRVAL.

CURRVAL returns the current number in a specified sequence. Before you can reference CURRVAL, you must use NEXTVAL to generate a sequence number.

ROWNUM returns a number indicating the sequence in which a row was selected from a table. The first row selected has a ROWNUM of 1, the second row has a ROWNUM of 2, and so on. If a SELECT statement includes an ORDER BY clause, ROWNUMs are assigned to the selected rows *before* the sort is done.

You can use ROWNUM to limit the number of rows returned by a SELECT statement. Also, you can use ROWNUM in an UPDATE statement to assign unique values to each row in a table. Using ROWNUM in the WHERE clause does not stop the processing of a SELECT statement; it just limits the number of rows retrieved. The only meaningful use of ROWNUM in a WHERE clause is

```
... WHERE ROWNUM < constant;
```

because the value of ROWNUM increases only when a row is retrieved. The following search condition can never be met because the first four rows are not retrieved:

```
... WHERE ROWNUM = 5 ;
```

LEVEL returns the level number of a node in a tree structure. The root is level 1, children of the root are level 2, grandchildren are level 3, and so on.

LEVEL is used in the SELECT CONNECT BY statement to incorporate some or all the rows of a table into a tree structure. In an ORDER BY or GROUP BY clause, LEVEL segregates the data at each level in the tree.

You specify the direction in which the query walks the tree (down from the root or up from the branches) with the PRIOR operator. In the START WITH clause, you specify a condition that identifies the root of the tree.

ROWID returns a row address in hexadecimal.

USER returns the username of the current ORACLE user.

UID returns the unique ID number assigned to an ORACLE user.

SYSDATE returns the current date and time.

ROWLABEL Column

SQL also recognizes the special column ROWLABEL, which Trusted ORACLE creates for every database table. Like other columns, ROWLABEL can be referenced in SQL statements. However, with standard ORACLE, ROWLABEL returns a null. With Trusted ORACLE, ROWLABEL returns the operating system label for a row.

A common use of ROWLABEL is to filter query results. For example, the following statement counts only those rows with a security level higher than “unclassified”:

```
EXEC SQL SELECT COUNT ( * ) INTO : head_count FROM EMP
    WHERE ROWLABEL > ‘ UNCLASSIFIED ‘ ;
```

For more information, see the *Trusted 0RACLE7 Server Administrator’s Guide.*

External Datatypes

As the table below shows, the external datatypes include all the internal datatypes plus several datatypes found in popular host languages. For example, the STRING datatype refers to a C null-terminated string, and the DECIMAL datatype refers to a COBOL packed decimal.

Name	*Code*	*Description*
VARCHAR2	1	variable+length character string
NUMBER	2	binary number
INTEGER	3	signed integer
FLOAT	4	floating point number
STRING	5	null-terminated character string
VARNUM	6	variable-length binary number
DECIMAL	7	COBOL or PL/I packed decimal
LONG	8	fixed-length character string
VARCHAR	9	variable-length character string
ROWID	11	binary value
DATE	12	fixed-length date/time value
VARRAW	15	variable-length binary data
RAW	23	fixed-length binary data
LONG RAW	24	fixed-length binary data
UNSIGNED	68	unsigned integer
DISPLAY	91	COBOL numeric character string
LONG VARCHAR	94	variable-length character string
LONG VARRAW	95	variable-length binary data
CHAR	96	fixed-length character string
CHARZ	97	C fixed-length, null-terminated character string
MLSLABEL	106	variable-length binary data

Brief descriptions of the external datatypes follow. For more information, see your *Supplement to the ORACLE7 Precompilers Guide.*

VARCHAR2

By default, unless MODE=ANSI, ORACLE assigns the VARCHAR2 datatype to all character host variables. You use the VARCHAR2 datatype to store variable-length character strings. The maximum length of a VARCHAR2 value is 2000 bytes. To define a VARCHAR2 variable, you use the syntax

```
VARCHAR2  (maximum_length)
```

where *maximum_length* is an integer literal in the range 1..2000.

You specify the maximum length of a VARCHAR2 *(n)* value in bytes, not characters. So, if a VARCHAR2 *(n)* variable stores multibyte characters, its maximum length is less than *n* characters.

On Input

ORACLE reads the number of bytes specified for the input host variable, strips any trailing blanks, then stores the input value in the target database column. Be careful. An uninitialized host variable can contain nulls. So, always blank-pad a character input host variable to its declared length.

If the input value is longer than the defined width of the database column, ORACLE generates an error. If the input value is all-blank, ORACLE treats it like a null.

ORACLE can convert a character value to a NUMBER column value if the character value represents a valid number. Otherwise, ORACLE generates an error.

On Output

ORACLE returns the number of bytes specified for the output host variable, blank-padding if necessary, then assigns the output value to the target host variable. If a null is returned, ORACLE fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, ORACLE truncates the value before assigning it to the host variable. If an indicator variable is available, ORACLE sets it to the original length of the output value.

ORACLE can convert NUMBER column values to character values. The length of the character host variable determines precision. If the host variable is too short for the number, scientific notation is used. For example, if you SELECT the column value 123456789 into a host variable of length 6, ORACLE returns the value ‘1.2E08’.

NUMBER

You use the NUMBER datatype to store fixed or floating point ORACLE numbers. You can specify precision and scale. The maximum precision of a NUMBER value is 38; the magnitude range is 1.0E-129 to 9.99E125. Scale can range from -84 to 127.

NUMBER values are stored invariable-length format, starting with an exponent byte and followed by up to 20 mantissa bytes. The high-order bit of the exponent byte is a sign bit, which is set for positive numbers. The low-order 7 bits represent the exponent, which is a base-100 digit with an offset of 65.

Each mantissa byte is a base-100 digit in the range 1..100. For positive numbers, 1 is added to the digit. For negative numbers, the digit is subtracted from 101, and, unless there are 20 mantissa bytes, a byte containing 102 is appended to the data bytes. Each mantissa byte can represent two decimal digits. The mantissa is normalized, and leading zeros are not stored. You can use up to 20 data bytes for the mantissa, but only 19 are guaranteed accurate. The 19 bytes, each representing a base-100 digit, allow a maximum precision of 38 digits.

On output, the host variable contains the number as represented internally by ORACLE. To accommodate the largest possible number, the output host variable must be 21 bytes long. Only the bytes used to represent the number are returned. ORACLE does not blank-pad or null-terminate the output value. If you need to know the length of the returned value, use the VARNUM datatype instead.

Normally, there is little reason to use this datatype.

INTEGER

You use the INTEGER datatype to store numbers that have no fractional part. An integer is a signed, 2-byte or 4-byte binary number. The order of the bytes in a word is system-dependent. You must specify a length for input and output host variables. On output, if the column value is a floating point number, ORACLE truncates the fractional part.

FLOAT

You use the FLOAT datatype to store numbers that have a fractional part or that exceed the capacity of the INTEGER datatype. The number is represented using the floating-point format of your computer and typically requires 4 or 8 bytes of storage. You must specify a length for input and output host variables.

ORACLE can represent numbers with greater precision than floating point implementations because the internal format of ORACLE numbers is decimal.

STRING

The STRING datatype is like the VARCHAR2 datatype, except that a STRING value is always null-terminated.

On Input
ORACLE uses the specified length to limit the scan for the null terminator. If a null terminator is not found, ORACLE generates an error. If you do not specify a length, ORACLE assumes the maximum length of 2000 bytes.

The minimum length of a STRING value is 2 bytes. If the first character is a null terminator and the specified length is 2, ORACLE inserts a null unless the column is defined as NOT NULL. An all-blank value is stored intact.

On Output
ORACLE appends a null byte to the last character returned. If the string length exceeds the specified length, ORACLE truncates the output value and appends a null byte. If a null is SELECTed, ORACLE returns a null byte in the first character position.

VARNUM

The VARNUM datatype is like the NUMBER datatype, except that the first byte of a VARNUM variable stores the length of the value.

On input, you must set the first byte of the host variable to the length of the value. On output, the host variable contains the length followed by the number as represented internally by ORACLE. To accommodate the largest possible number, the host variable must be 22 bytes long. After SELECTing a column value into a VARNUM host variable, you can check the first byte to get the length of the value.

Normally, there is little reason to use this datatype.

DECIMAL

With Pro*COBOL and Pro*PL/I, you use the DECIMAL datatype to store packed decimal numbers for calculation. In COBOL, the host variable must be a signed COMP-3 field with an implied decimal point. You can use the returned DECIMAL value “as is” in COBOL calculations or move it to a computational field. If the output host variable is too short for the number, scientific notation is *not* used. If significant digits are lost during data conversion, ORACLE fills the host variable with asterisks.

LONG

You use the LONG datatype to store variable-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 2147483647 bytes or two gigabytes.

VARCHAR

You use the VARCHAR datatype to store variable-length character strings. VARCHAR variables have a 2-byte length field followed by a ≤ 65533-byte string field. However, for VARCHAR array elements, the maximum length of the string field is 65530 bytes. When you specify the length of a VARCHAR variable, be sure to include 2 bytes for the length field. For longer strings, use the LONG VARCHAR datatype.

ROWID

You use the ROWID datatype to store binary rowids in (typically 13-byte) fixed-length fields. The field size is port-specific. So, check the ORACLE installation or user's guide for your system.

You can use VARCHAR2 host variables to store rowids in a readable format. When you SELECT or FETCH a rowid into a VARCHAR2 host variable, ORACLE converts the binary value to an 18-byte character string and returns it in the format

```
BBBBBBBB.RRRR.FFFF
```

where BBBBBBBB is the block in the database file, RRRR is the row in the block (the first row is 0), and FFFF is the database file. These numbers are hexadecimal. For example, the rowid

```
0000000E.000A.0007
```

points to the 11th row in the 15th block in the 7th database file.

Typically, you FETCH a rowid into a VARCHAR2 host variable, then compare the host variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement. That way, you can identify the latest row fetched by a cursor. For an example, see the section" Mimicking CURRENT OF' in Chapter 7.

Note: If you need full portability or your application communicates with a non-ORACLE database via SQL*Connect, specify a maximum length of 2000 (not 18) bytes when declaring the VARCHAR2 host variable. If your application communicates with a non-ORACLE database via Oracle Open Gateway, specify a maximum length of 256 bytes. Though you can assume nothing about its contents, the host variable will behave normally in SQL statements.

DATE

You use the DATE datatype to store dates and times in 7-byte, fixed-length fields. As Table 3-1 shows, the century, year, month, day, hour (in 24-hour format), minute, and second are stored in that order from left to right.

Table 3-1
DATE Format

Byte	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example 06-DEC-1990	119	190	12	6	1	1	1

The century and year bytes are in excess-100 notation. The hour, minute, and second are in excess-1 notation. Dates before the Common Era (B.C.E.) are less than 100. The epoch begins on January 1,4712 B.C. For this date, the century byte is 53 and the year byte is 88. The hour byte ranges from 1 to 24. The minute and second bytes range from 1 to 60. The time defaults to midnight (1,1, 1).

Normally, there is little reason to use this datatype.

VARRAW

You use the VARRAW datatype to store binary data or byte strings. The VARRAW datatype is like the RAW datatype, except that VARRAW variables have a 2-byte length field followed by a ≤ 65533-byte data field. For longer strings, use the LONG VARRAW datatype.

When you specify the length of a VARRAW variable, be sure to include 2 bytes for the length field. The first two bytes of the variable must be interpretable as an integer.

To get the length of a VARRAW variable, simply refer to its length field.

RAW

You use the RAW datatype to store binary data or byte strings. The maximum length of a RAW value is 255 bytes.

RAW data is like CHAR data, except that ORACLE assumes nothing about the meaning of RAW data and does no character set conversions when you transmit RAW data from one system to another.

LONG RAW	You use the LONG RAW datatype to store binary data or byte strings. The maximum length of a LONG RAW value is 2147483647 bytes or two gigabytes. LONG RAW data is like LONG data, except that ORACLE assumes nothing about the meaning of LONG RAW data and does no character set conversions when you transmit LONG RAW data from one system to another.
UNSIGNED	You use the UNSIGNED datatype to store unsigned integers. An unsigned integer is a binary number of 2 or 4 bytes. The order of the bytes in a word is system-dependent. You must specify a length for input and output host variables. On output, if the column value is a floating point number, ORACLE truncates the fractional part.
DISPLAY	With Pro*COBOL, you use the DISPLAY datatype to store numeric character data. The DISPLAY datatype refers to a COBOL "DISPLAY SIGN LEADING SEPARATE" number, which typically requires $n + 1$ bytes of storage for PIC S9 *(n)*, and $n + d + 1$ bytes of storage for PIC S9 *(n)* V9 *(d)*.
LONG VARCHAR	You use the LONG VARCHAR datatype to store variable-length character strings. LONG VARCHAR variables have a 4-byte length field followed by a string field. The maximum length of the string field is 2147483643($2^{31} - 5$) bytes. When you specify the length of a LONG VARCHAR variable, be sure to include 4 bytes for the length field.
LONG VARRAW	You use the LONG VARRAW datatype to store binary data or byte strings. LONG VARRAW variables have a 4-byte length field followed by a data field. The maximum length of the data field is 2147483643 bytes. When you specify the length of a LONG VARRAW variable, be sure to include 4 bytes for the length field.

CHAR

By default, when MODE=ANSI, ORACLE assigns the CHAR datatype to all character host variables. You use the CHAR datatype to store fixed-length character strings. The maximum length of a CHAR value is 255 bytes.

On Input
ORACLE reads the number of bytes specified for the input host variable, does *not* strip trailing blanks, then stores the input value in the target database column.

If the input value is longer than the defined width of the database column, ORACLE generates an error. If the input value is all-blank, ORACLE treats it like a character value.

On Output
ORACLE returns the number of bytes specified for the output host variable, blank-padding if necessary, then assigns the output value to the target host variable. If a null is returned, ORACLE fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, ORACLE truncates the value before assigning it to the host variable. If an indicator variable is available, ORACLE sets it to the original length of the output value.

CHARZ

By default, when MODE=ANSI, ORACLE assigns the CHARZ datatype to all character host variables in a Pro*C program. You use the CHARZ datatype to store fixed-length, null-terminated character strings. The maximum length of a CHARZ value is 255 bytes.

On input, the CHARZ and STRING datatypes work the same way. You must null-terminate the input value. The null terminator serves only to delimit the string; it is not part of the data.

On output, the CHARZ and CHAR datatypes work the same way. ORACLE appends a null terminator to the output value, which is also blank-padded if necessary.

MLSLABEL

You use the MLSLABEL datatype to store variable-length, binary operating system labels. Trusted ORACLE uses labels to control access to data. For more information, see the *Trusted ORACLE7 Server Administrator's Guide.*

You can use the MLSLABEL datatype to define a column. However, with standard ORACLE, such columns can store nulls only. With Trusted ORACLE, you can insert any valid operating system label into a column of type MLSLABEL.

On Input
Trusted ORACLE translates the input value into a binary label, which must be a valid operating system label. If it is not, Trusted ORACLE issues an error message. If the label is valid, Trusted ORACLE stores it in the target database column.

On Output
Trusted ORACLE converts the binary label to a character string, which can be of type CHAR, CHARZ, STRING, VARCHAR, or VARCHAR2.

VARCHAR2 versus CHAR

The VARCHAR2 and CHAR datatypes differ in subtle but significant ways. CHAR semantics have changed slightly to comply with the current ANSI/ISO SQL standard. The changes come into play when you compare, INSERT, UPDATE, SELECT, or FETCH character values.

On Input

When MODE=ANSI, if both values being compared in a SQL statement belong to type CHAR, *blank-padding semantics* are used. That is, before comparing character values of unequal length, ORACLE blank-pads the shorter value to the length of the longer value. For example, if ENAME is a CHAR database column and *emp_name* is a CHAR host variable (by default or via datatype equivalencing), the following search condition is TRUE when the column value 'BELL' and the host value 'BELL ' are compared:

```
... WHERE ENAME = : emp_name;
```

When MODE= {ANSI14 | ANSI13 | ORACLE}, if either or both values in a comparison belong to type VARCHAR2, *non-blank-padding* semantics are used. That is, when comparing character values of unequal length, ORACLE makes no adjustments and uses the exact lengths. For example, if JOB is a CHAR column and *job_title* is a VARCHAR2 host variable, the following search condition is FALSE when the column value 'CLERK' and the host value 'CLERK ' are compared:

```
... WHERE JOB = : job_title;
```

When you INSERT a character value into a CHAR database column, if the value is shorter than the defined width of the column, ORACLE blank-pads the value to the defined width. As a result, information about trailing blanks is lost. If the character value is longer than the defined width of the CHAR column, ORACLE generates an error. ORACLE neither truncates the value nor tries to trim trailing blanks.

When you INSERT a character value into a VARCHAR2 database column, if the value is shorter than the defined width of the column, ORACLE does *not* blank-pad the value. Nor does ORACLE strip trailing blanks. Character values are stored intact, so no information is lost. If the character value is longer than the defined width of the VARCHAR2 column, ORACLE generates an error. ORACLE neither truncates the value nor tries to trim trailing blanks.

The same rules apply when UPDATEing.

On Output

When you SELECT a column value into a CHAR host variable, if the value is shorter than the declared length of the variable, ORACLE blank-pads the value to the declared length. For example, if *emp_name* is a CHAR(15) host variable (by default or via datatype equivalencing), and you SELECT a 10-byte column value into it, ORACLE adds 5 blanks. If the column value is longer than the declared length of the CHAR host variable, ORACLE truncates the value, stores it, and generates a warning.

When you SELECT a column value into a VARCHAR2 host variable, if the value is shorter than the declared length of the variable, ORACLE does *not* blank-pad the value. Nor does ORACLE strip trailing blanks. If the column value is longer than the declared length of the VARCHAR2 host variable, ORACLE truncates the value, stores it, and generates a warning.

The same rules apply when FETCHing.

Datatype Conversion

At precompiled time, an external datatype is assigned to each host variable in the Declare Section. For example, the precompiled assigns the VARCHAR2 external datatype to character host variables. At run time, the datatype code of every host variable used in a SQL statement is passed to ORACLE. ORACLE uses the codes to convert between internal and external datatypes.

Before assigning a SELECTed column (or pseudocolumn) value to an output host variable, if necessary, ORACLE converts the internal datatype of the source column to the datatype of the host variable. Likewise, before assigning or comparing the value of an input host variable to a database column, if necessary, ORACLE converts the external datatype of the host variable to the internal datatype of the target column.

However, the datatype of the host variable must be compatible with that of the source or target database column. It is your responsibility to make sure that values are convertible. For example, if you try to convert the string value 'YESTERDAY' to a DATE column value, you get an error.

Conversions between internal and external datatypes follow the usual data conversion rules. For instance, you can convert a CHAR value of '1234' to a 2-byte integer. But, you cannot convert a CHAR value of '65543' (number too large) or '10F' (number not decimal) to a 2-byte integer. Likewise, you cannot convert a string value that contains alphabetic characters to a NUMBER value.

Number conversion follows the conventions specified by NLS (national language support) parameters in the ORACLE initialization file. For example, your system might be configured to recognize ',' instead of '.' as the decimal character. For more information about NLS, see the *ORACLE7 Server Application Developer's Guide.*

Table 3-2 shows the supported conversions between internal and external datatypes; Table 3-3 shows the additional conversions supported by Trusted ORACLE.

Table 3-2 Supported Datatype Conversions

EXTERNAL	INTERNAL 1 VARCHAR2	2 NUMBER	8 LONG	11 ROWID	12 DATE	23 RAW	24 LONG RAW	96 CHAR
1 VARCHAR2	↵↑	↵↑	↵↑	↵↑ 1	↵↑ 2	↵↑ 3	↑ 3	↵↑
2 NUMBER	↵↑ 4	↵↑	↑					↵↑ 4
3 INTEGER	↵↑ 4	↵↑	↑					↵↑ 4
4 FLOAT	↵↑ 4	↵↑	↑					↵↑ 4
5 STRING	↵↑	↵↑	↵↑	↵↑ 1	↵↑ 2	↵↑ 3	↑ 3,5	↵↑
6 VARNUM	↵↑ 4	↵↑	↑					↵↑ 4
7 DECIMAL	↵↑ 4	↵↑	↑					↵↑ 4
8 LONG	↵↑	↵↑	↵↑	↵↑ 1	↵↑ 2	↵↑ 3	↑ 3,5	↵↑
9 VARCHAR	↵↑	↵↑	↵↑	↵↑ 1	↵↑ 2	↵↑ 3	↑ 3,5	↵↑
11 ROWID	↑		↑	↵↑				↑
12 DATE	↵↑		↑		↵↑			↵↑
15 VARRAW	↵↑ 6		↑ 5,6			↵↑	↵↑	↵↑ 6
23 RAW	↵↑ 6		↑ 5,6			↵↑	↵↑	↵↑ 6
24 LONG RAW	↵ 6		↑ 5,6			↵↑	↵↑	↵ 6
68 UNSIGNED	↵↑ 4	↵↑	↑					↵↑ 4
91 DISPLAY	↵↑ 4	↵↑	↑					↵↑ 4
94 LONG VARCHAR	↵↑	↵↑	↵↑	↵↑ 1	↵↑ 2	↵↑ 3	↑ 3,5	↵↑
95 LONG VARRAW	↵↑ 6		↑ 5,6			↵↑	↵↑	↵↑ 6
96 CHAR	↵↑	↵↑	↵↑	↵↑ 1	↵↑ 2	↵↑ 3	↑ 3	↵↑
97 CHARZ	↵↑	↵↑	↵↑	↵↑ 1	↵↑ 2	↵↑ 3	↑ 3	↵↑

Notes:

1. For input, host string must be in ORACLE 'BBBBBBBB.RRRR.FFFF' format. On output, column value is returned in same format.
2. For input, host string must be the default DATE character format. On output, column value is returned in same format.
3. For input, host string must be in hex format. On output, column value is returned in same format.
4. For output, column value must represent a valid number.
5. Length must be less than or equal to 2000.
6. On input, column value is stored in hex format. For output, column value must be in hex format.

Legend:

↑ = input only

↵ = output only

↵↑ = input or output

Table 3-3
Trusted ORACLE
Additional Conversions

EXTERNAL	INTERNAL 1 VARCHAR2	8 LONG	96 CHAR	106 MLSLABEL
1 VARCHAR2				 [1]
5 STRING				 [1]
8 LONG				 [1]
9 VARCHAR				 [1]
94 LONG VARCHAR				 [1]
96 CHAR				 [1]
97 CHARZ				 [1]
106 MLSLABEL	 [2]	 [2]	 [2]	

Notes:

1. For input, host string must be a valid OS label in text format.
   On output, column value is returned in same format.
2. For input, host string must be a valid OS label in raw format.
   On output, column value is returned in same format.

DATE Values

When you SELECT a DATE column value into a character host variable, ORACLE must convert the internal binary value to an external character value. So, ORACLE implicitly calls the SQL function TO_CHAR, which returns a character string in the default date format. To get other information such as the time or Julian date, you must explicitly use TO_CHAR with a format mask.

A conversion is also necessary when you INSERT a character host value into a DATE column. ORACLE implicitly calls the SQL function TO_DATE, which expects the default date format. To INSERT dates in other formats, you must explicitly use TO_DATE with a format mask.

RAW and LONG RAW Values

When you SELECT a RAW or LONG RAW column value into a character host variable, ORACLE must convert the internal binary value to an external character value. In this case, ORACLE returns each binary byte of RAW or LONG RAW data as a pair of characters. Each character represents the hexadecimal equivalent of a nibble (half a byte). For example, ORACLE returns the binary byte 11111111 as the pair of characters 'FF'. The SQL function RAWTOHEX does the same conversion.

A conversion is also necessary when you INSERT a character host value into a RAW or LONG RAW column. Each pair of characters in the host variable must represent the hexadecimal equivalent of a binary byte. If a character does not represent the hexadecimal value of a nibble, ORACLE issues the following error message:

```
ORA-01465: invalid hex number
```

Declaring and Referencing Host Variables

Every program variable used in a SQL statement must be declared as a host variable. You declare a host variable in the Declare Section according to the rules of the host language. Normal scoping rules apply.

Host variable names can be any length, but only the first 31 characters are significant. For ANSI/ISO compliance, a host variable name must be ≤ 18 characters long, begin with a letter, and not contain consecutive or trailing underscores.

The external datatype of a host variable and the internal datatype of its source or target database column need not be the same, but they must be compatible. Table 3-2 shows the compatible datatypes between which ORACLE converts automatically when necessary.

The ORACLE Precompilers support most built-in host language datatypes. For a list of supported datatypes, see your *Supplement to the ORACLE Precompilers Guide.* User-defined datatypes are not supported. However, the Pro*C and Pro*Pascal Precompilers allow you to equivalence user-defined datatypes to ORACLE external datatypes. Datatype equivalencing is discussed in the next section.

Although references to a user-defined structure are not allowed, the Pro*COBOL and Pro*PL/I Precompilers let you reference individual elements of the structure as if they were host variables. You can use such references wherever host variables are allowed.

Some Examples

In the following example, you declare three host variables, then use a SELECT statement to search the database for an employee number matching the value of host variable *emp_number*. When a matching row is found, ORACLE sets output host variables *dept_number* and *emp_name* to the values of columns DEPTNO and ENAME in that row.

```
-- declare host variables
EXEC SQL BEGIN DECLARE SECTION;
    emp_number     INTEGER;
    emp_name       CHARACTER (10);
    dept_number    INTEGER;
EXEC SQL END DECLARE SECTION;
...
display  ' Employee number? ';
read emp_number;

EXEC SQL SELECT DEPTNO, ENAME INTO : dept_number, : emp_name
    FROM EMP
    WHERE EMPNO =  : emp_number ;
```

For more information about using host variables in your program, see the section "Using Host Variables" in Chapter 4.

Pointer Variables

C, Pascal, and PL/I support a special class of variables called *pointers*, which "point" to other host language variables. A pointer holds the address (storage location) of a variable, not its value. Some host languages identify pointers by prefixing them with a special character such as an asterisk or caret. As the following example shows, you can define pointers as host variables in the Declare Section:

```
EXEC SQL BEGIN DECLARE SECTION;
    ^int_ptr   INTEGER;
    ^char_ptr CHARACTER (10);
EXEC SQL END DECLARE SECTION;
```

However, the use of pointer variables is not ANSI-compliant. You can use the FIPS option to identify such extensions. For more information, see the section "Using the Precompiled Options" in Chapter 11.

If you use them in SQL statements, prefix pointers with a colon instead of the special character, as follows:

```
EXEC SQL SELECT INTCOL INTO : int_ptr FROM MYTAB;
```

Except for string values, the size of the referenced value is that of its declared base type. ORACLE determines the size of string values at run time by calling a string-length function.

VARCHAR Variables

You can use the VARCHAR pseudotype to declare variable-length character strings. (A pseudotype is a datatype not native to your host language.) Recall that VARCHAR variables have a 2-byte length field followed by a string field. For example, the Pro*C Precompiled expands the VARCHAR declaration

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR username [20];
EXEC SQL END DECLARE SECT ION;
```

into the following struct with array and length members:

```
struct {
    unsigned short  len;
    unsigned char   arr[20];
} username;
```

To get the length of a VARCHAR, you simply refer to its length field. You need not use a string function or character-counting algorithm.

For more information about VARCHARs, see your *Supplement to the ORACLE Precompilers Guide.*

Guidelines

The following guidelines apply to declaring and referencing host variables.

A host variable must be

- explicitly declared in the Declare Section
- prefixed with a colon (:) in SQL statements and PL/SQL blocks
- of a datatype supported by the host language
- of a datatype compatible with that of its source or target database column

A host variable must *not* be

- subscripted
- prefixed with a colon in host language statements
- used to identify a column, table, or other ORACLE object
- used in data definition statements such as ALTER, CREATE, and DROP
- an ORACLE reserved word (refer to Appendix C)

A host variable can be

- used anywhere an expression can be used in a SQL statement
- associated with an indicator variable

Indicator Variables

You can associate every host variable with an optional indicator variable. You use indicator variables to assign nulls to input host variables and detect null or truncated values in output host variables.

An Example

In the following example, you declare three host variables and one indicator variable, then use a SELECT statement to search the database for an employee number matching the value of host variable *emp_number*. When a matching row is found, ORACLE sets output host variables *salary* and *commission* to the values of columns SAL and COMM in that row and stores a return code in indicator variable *ind_comm*. The next statement uses *ind_comm* to select a course of action.

```
-- declare host and indicator variables
EXEC SQL BEGIN DECLARE SECTION;
    emp_number    INTEGER ;
    salary        REAL ;
    commission    REAL ;
    ind_comm      SMALLINT ; -- indicator variable
EXEC SQL END DECLARE SECTION;

    pay           REAL ;   -- not used in a SQL statement

display  ' Employee number? ' ;
read emp_number;

EXEC SQL SELECT SAL, COMM
    INTO :salary,  :commission:ind_connn
    FROM EMP
    WHERE EMPNO =  :emp_number;

IF ind_comm = -1 THEN -- commission is null
    set pay = salary;
ELSE
    set pay =  salary + commission;
ENDIF;
...
```

To improve readability, you can precede any indicator variable with the optional keyword INDICATOR. You must still prefix the indicator variable with a colon. The correct syntax is

```
:host_variable  INDICATOR  :indicator_variable
```

which is equivalent to

```
:host_variable:indicator_variable
```

You can use both forms of expression in your host program.

For more information about using indicator variables in your program, see the section "Using Indicator Variables" in Chapter 4.

Guidelines

The following guidelines apply to declaring and referencing indicator variables.

An indicator variable must be

- explicitly declared in the Declare Section as a 2-byte integer
- prefixed with a colon (:) in SQL statements
- appended to its associated host variable in SQL statements and PL/SQL blocks

An indicator variable must *not* be

- used in the WHERE clause of a SQL statement
- prefixed with a colon in host language statements
- appended to its associated host variable in host language statements
- an ORACLE reserved word

Datatype Equivalencing

Datatype equivalencing lets you customize the way ORACLE interprets input data and the way ORACLE formats output data.

On a variable-by-variable basis, you can equivalence supported host language datatypes to the ORACLE external datatypes. With Pro*C and Pro*Pascal, you can also equivalence user-defined datatypes to ORACLE external datatypes. In both cases, you can specify any external datatype except DECIMAL, DISPLAY, and NUMBER. DECIMAL is available only with Pro*COBOL and Pro*PL/I. DISPLAY is available only with Pro*COBOL. NUMBER is not needed because you can use VARNUM instead.

Why Equivalence Datatypes?

Datatype equivalencing is useful in several ways. For example, suppose you want to use a null-terminated host string in your program. You can declare a string host variable, then equivalence it to the external datatype STRING, which is always null-terminated.

You can use datatype equivalencing when you want ORACLE to store but not interpret data. For example, if you want to store an integer host array in a LONG RAW database column, you can equivalence the host array to the external datatype LONG RAW.

Also, you can use datatype equivalencing to override default datatype conversions. For instance, unless NLS parameters in the ORACLE initialization file specify otherwise, if you SELECT a DATE column value into a character host variable, ORACLE returns a 9-byte string formatted as follows:

```
DD-MON-YY
```

But, if you equivalence the host variable to the DATE external datatype, ORACLE returns a 7-byte value formatted as shown in Table 3-1.

In Pro*C and Pro*Pascal, you can equivalence user-defined datatypes. For example, suppose you need a variable-length string datatype to hold graphics characters. You can define a new datatype with a 2-byte length field followed by as a ≤ 65533-byte data field, then equivalence it to the VARRAW external datatype.

Host Variable Equivalencing

By default, the ORACLE Precompilers assign a specific external datatype to every host variable. (These default assignments are tabulated in your *Supplement to the ORACLE Precompilers Guide.)* You can override-the default assignments by equivalencing host variables to ORACLE external datatypes in the Declare Section. This is called *host variable equivalencing.* The syntax you use is

```
EXEC SQL VAR host_variable
    IS type_name [ ( {length |precision, scale} ) 1 ;
```

where:

host_variable Is an input or output host variable (or host array) declared *earlier* in the Declare Section.

The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by an *n* -byte data field, where *n* lies in the range 1..65533. So, if *type_name* is VARCHAR or VARRAW, *host_variable* must be at least 3 bytes long.

The LONG VARCHAR and LONG VARRAW external datatypes have a 4-byte length field followed by an *n* -byte data field, where n lies in the range 1..2147483643. So, if *type_name is* VARCHAR or VARRAW, *host_variable* must be at least 5 bytes long.

type_name Is the name of a valid external datatype such as RAW or STRING.

length Is an integer literal specifying a valid length in bytes. The value of *length* must be large enough to accommodate the external datatype.

When *type_name* is DECIMAL or DISPLAY, you must specify *precision* and *scale* instead of *length* When *type_name* is VARNUM, ROWID, or DATE, you cannot specify *length* because it is predefine. For other external datatypes, *length* is optional. It defaults to the length of *host_variable.*

When specifying *length*, if *type_name* is VARCHAR, VARRAW, LONG VARCHAR, or LONG VARRAW, use the maximum length of the data field only. The precompiled accounts for the length field.

precision and *scale* — Are integer literals specifying a precision and scale allowed by the equivalence external datatype. You can specify a *precision* of 1..99 and a *scale* of -84..99.

However, the maximum precision and scale of a database column are 38 and 127, respectively. So, if *precision* exceeds 38, you cannot INSERT the value of *host_variable* into a database column. On the other hand, if the scale of a column value exceeds 99, you cannot SELECT or FETCH the value into *host_variable.*

This parameter is for COBOL and PL/I packed decimals or COBOL "USAGE IS DISPLAY" numbers. Specify *precision* and scale only when *type_name* is DECIMAL or DISPLAY.

Table 3-3 shows which parameters to use with each external datatype.

An Example

Suppose you want to SELECT employee names from the EMP table, then pass them to a routine that expects null-terminated strings. You need not explicitly null-terminate the names. Simply equivalence a host variable to the STRING external datatype, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    emp_name CHARACTER (11) ;
    EXEC SQL VAR emp_name IS STRING (11) ;
EXEC SQL END DECLARE SECTION;
```

The width of the ENAME column is 10 characters, so you allot the new *emp_name* 11 characters to accommodate the null terminator. (Here, *length* is optional because it defaults to the length of the host variable.) When you SELECT a value from the ENAME column into *emp_name,* ORACLE null-terminates the value for you.

Table 3-4 Parameters for External Datatypes

Datatype	Length	Precision	Scale	Defaults
1 VARCHAR2	optional (max is 2000)	n/a	n/a	declared length of variable
2 NUMBER	n/a	n/a	n/a	not available
3 INTEGER	optional (1, 2, or 4)	n/a	n/a	declared length of variable
4 FLOAT	optional (4 or 8)	n/a	n/a	declared length of variable
5 STRING	optional (max is 65535)	n/a	n/a	declared length of variable
6 VARNUM	n/a	n/a	n/a	22 (variable-length)
7 DECIMAL	n/a	required	optional	none
8 LONG	optional (max is 2147463647)	n/a	n/a	declared length of variable
9 VARCHAR	required (max is 65533)	n/a	n/a	none
11 ROWID	n/a	n/a	n/a	port specific (fixed-length)
12 DATE	n/a	n/a	n/a	(fixed-length)
15 VARRAW	required (max is 65533)	n/a	n/a	none
23 RAW	optional (max is 255)	n/a	n/a	declared length of variable
24 LONG RAW	optional (max is 2147483647)	n/a	n/a	declared length of variable
66 UNSIGNED	optional (1, 2, or 4)	n/a	n/a	declared length of variable
91 DISPLAY	n/a	required	optional	none
84 LONG VARCHAR	required (max is 2147463643)	n/a	n/a	none
95 LONG VARRAW	required (max is 2147483643)	n/a	n/a	none
96 CHAR	optional (max is 255)	n/a	n/a	declared length of variable
97 CHARZ	optional (max is 255)	n/a	n/a	declared length of variable
106 MLSLABEL	required (max is 255)	n/a	n/a	none

User-defined Type Equivalencing

With Pro*C and Pro*Pascal, you can assign an ORACLE external datatype to a whole class of host variables using a technique called *user-defined type equivalencing.* First, define a new datatype structured like the external datatype that suits your needs. Then, equivalence your new datatype to the external datatype in the Declare Section. The syntax you use is

```
EXEC SQL TYPE user_type IS type_name [ ( length ) 1 [REFERENCE I ;
```

where:

user_type — Is a user-defined (not built-in) datatype identifier declared *earlier* inside or outside the Declare Section.

The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by an n-byte data field, where *n* lies in the range 1..65533. So, if *type_name* is VARCHAR or VARRAW, *user_type* must beat least 3 bytes long.

The LONG VARCHAR and LONG VARRAW external datatypes have a 4-byte length field followed by an *n* -byte data field, where *n* lies in the range 1..2147483643. So, if *type_name* is VARCHAR or VARRAW, *user_type* must be at least 5 bytes long.

type_name — Is the name of a valid external datatype such as RAW or STRING.

length — Is an integer literal specifying a valid length in bytes. The value of *length* must be large enough to accommodate the external datatype.

You must specify *length unless type_name is* VARNUM, ROWID, or DATE, in which case you cannot specify *length* because it is predefine.

When specifying *length*, if *type_name* is VARCHAR, VARRAW, LONG VARCHAR, or LONG VARRAW, use the maximum length of the data field only. The precompiled accounts for the length field.

REFERENCE — Is an optional keyword that denotes a pointer *user_type.*

Table 3-3 shows which parameters to use with each external datatype.

Some Examples

In Pro*C, suppose you need a variable-length string datatype to hold graphics characters. First, declare a **struct** with a 2-byte length field followed by a ≤ 65533-byte data field. Second, use **typedef** to define a new datatype based on the struct. Then, equivalence your new user-defined datatype to the VARRAW external datatype, as shown in the following example:

```
struct  screen { short  len;
                 char buff [4000] ;
               };
typedef  struct screen graphics;

EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL TYPE graphics IS VARRAW (4000);
    graphics  crt; -- host variable of type graphics
    ...
EXEC SQL END DECLARE SECTION;
```

You specify a length of 4000 bytes for the new *graphics* type because that is the maximum length of the data field in your **struct.** The precompiler allocates 2 bytes for the length field and up to 4000 bytes for the data field depending on the alignment requirements of your system.

In Pro*Pascal, suppose you want to store a two-dimensional array of 4-byte integers in a RAW database column. First, define an array type, then equivalence it to the RAW external datatype, as shown in the next example:

```
Type
    RawArray = Array [1. . 10, 1..20] of Integer;

Var
    EXEC SQL BEGIN DECLARE SECTION;
        EXeC SQL TYPE RawArray IS RAW (800);
        Matrix: RawArray;   -- host variable of type RawArray
        ...
    EXEC SQL END DECLARE SECTION;
```

In Pro*C, suppose you need a datatype that specifies pointers to integers. First, use **typedef** to define a new **int** pointer type. Then, equivalence your new user-defined datatype to the external datatype INTEGER, as shown in the following example:

```
typedef  int * my_int;

EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL TYPE my_int IS INTEGER REFERENCE;
    my_int ptr; -- pointer host variable of type my_int
    ...
EXEC SQL END DECLARE SECTION;
```

Guidelines

To input VARNUM or DATE values, you must use the ORACLE internal format. Keep in mind that ORACLE uses the internal format to output VARNUM and DATE values.

If no ORACLE external datatype suits your needs exactly, use a VARCHAR2-based or RAW-based external datatype.

For EXEC SQL TYPE Only

Place all your EXEC SQL TYPE statements right after the BEGIN DECLARE SECTION statement. That way, you cannot make an illegal forward reference when defining host variables.

With arrays, *length* must match exactly the buffer size required to hold the user-defined datatype. Also, ORACLE expects VARCHAR and VARRAW arrays to be word-aligned. So, when you equivalence an array type to the VARCHAR or VARRAW datatype, make sure that *length* + 2 is divisible by 4, as illustrated in the following Pro*C example:

```
typedef   char  my_array [20];

EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL TYPE my_array IS VARCRAR (18) ;    - - 1 8 + 2 = 2 0
    my_array  ary;   -- host variable of type my_array

EXEC SQL END DECLARE SECTION;
```

National Language Support

Although the widely-used 7- or 8-bit ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, some Asian languages, such as Japanese, contain thousands of characters. These languages require 16 bits (two bytes) to represent each character. How does ORACLE deal with such dissimilar languages?

ORACLE provides National Language Support (NLS), which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments. With NLS, number and date formats adapt automatically to the language conventions specified for a user session. Thus, NLS allows users around the world to interact with ORACLE in their native languages.

You control the operation of language-dependent features by specifying various NLS parameters. Default values for these parameters can be set in the ORACLE initialization file. The following table shows what each NLS parameter specifies.

NLS Parameter	*Specifies . . .*
NLS_LANGUAGE	language-dependent conventions
NLS_TERRITORY	territory-dependent conventions
NLS_DATE_FORMAT	date format
NLS_DATE_LANGUAGE	language for day and month names
NLS_NUMERIC_CHARACTERS	decimal character and group separator
NLS_CURRENCY	local currency symbol
NLS_ISO_CURRENCY	ISO currency symbol
NLS_SORT	sort sequence

The main parameters are NLS_LANGUAGE and NLS_TERRITORY. NLS_LANGUAGE specifies the default values for language-dependent features, which include

- language for Server messages
- language for day and month names
- sort sequence

NLS_LANGUAGE specifies the default values for territory-dependent features, which include

- date format
- decimal character
- group separator
- local currency symbol
- ISO currency symbol

You can control the operation of language-dependent NLS features for a user session by specifying the parameter NLS_LANG as follows

```
NLS_LANG = <language>_<territory>. <character set>
```

where *language* specifies the value of NLS_LANGUAGE for the user session, *territory* specifies the value of NLS_TERRITORY, and *character set* specifies the encoding scheme used for the terminal. An encoding *scheme* (usually called a character set or code page) is a range of numeric codes that corresponds to the set of characters a terminal can display. It also includes codes that control communication with the terminal.

You define NLS_LANG as an environment variable (or the equivalent on your system). For example, on UNIX using the C shell, you might define NLS_LANG as follows:

```
setenv NLS_LANG French_France. WE8ISO8859P1
```

To change the values of NLS parameters during a session, you use the ALTER SESSION statement as follows:

```
ALTER SESSION SET <nls_parameter> = <value>
```

The ORACLE Precompilers fully support all the NLS features that allow your applications to process multilingual data stored in an ORACLE database. For example, you can declare foreign-language character variables and pass them to string functions such as INSTRB, LENGTHB, and SUBSTRB. These functions have the same syntax as the INSTR, LENGTH, and SUBSTR functions, respectively, but operate on a per-byte basis rather than a per-character basis.

You can use the functions NLS_INITCAP, NLS_LOWER, and NLS_UPPER to handle special instances of case conversion. And, you can use the function NLSSORT to specify WHERE-clause comparisons based on linguistic rather than binary ordering. You can even pass NLS parameters to the TO_CHAR, TO_DATE, and TO_NUMBER functions. For more information about NLS, see the *ORACLE7 Server Application Developer's Guide.*

Caution

Remember, you specify the maximum length of a VARCHAR2 *(n)* variable in bytes, not characters. So, if a VARCHAR2 *(n)* variable stores multibyte characters, its maximum length is less than *n* characters. This also applies to CHAR(n) variables.

Connecting to ORACLE

Your host program must log onto ORACLE before querying or manipulating data. To log on, simply use the following CONNECT statement

```
EXEC SQL CONNECT :username IDENTIFIED BY : password;
```

Or, you can use the statement

```
EXEC SQL CONNECT : usr_pwd;
```

where *usr_pwd* contains *username/password* (make sure to include the slash).

The CONNECT statement must be the *first* executable SQL statement in the program. Only declarative SQL statements and host language code can logically precede the CONNECT statement.

To supply the ORACLE username and password separately, you must define two host variables in the Declare Section as character strings. If you supply a userid containing both username and password, only one host variable is needed.

Make sure to set the username and password variables before the CONNECT is executed, or it will fail. You can hardcode the values into your program or have the program prompt for them, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    username   CHARACTER (20);
    password   CHARACTER (20);
EXEC SQL END DECLARE SECTION;

display  ' Username? ' ;
read username;
display  ' Password? ' :
read password;

-- handle processing errors
EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

-- connect to local database
EXEC SQL CONNECT : username IDENTIFIED BY : password
...
sqlerror:
    ...
```

Automatic Logons

You can automatically log onto ORACLE with the userid

```
OPS$username
```

where *username* is your current operating system user or task name and OPS$ *username* is a valid ORACLE userid.

To take advantage of the automatic logon feature, you simply pass a slash (/) character to the precompiled, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    oracleid   CHARACTER ( 1 ) ;
EXEC SQL END DECLARE SECTION;
...
set oracleid = ' /' ;

EXEC SQL CONNECT : oracleid;
```

This automatically connects you as user OPS$username. For example, if your operating system userid is JBASS, and OPS$JBASS is a valid ORACLE userid, connecting with "/" automatically logs you on to ORACLE as user OPS$JBASS.

The character string that you pass to the precompiled cannot contain blanks. For example, the following CONNECT statement will fail:

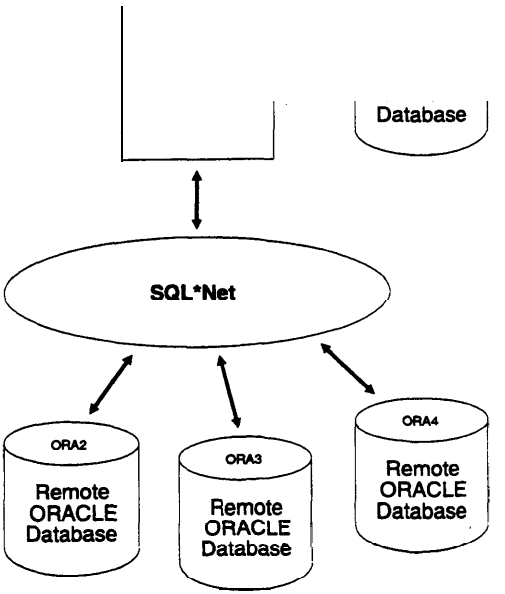
```
EXEC SQL BEGIN DECLARE SECTION;
    oracleid   CHARACTER ( 5 ) ;
EXEC SQL END DECLARE SECTION;
...
set oracleid = ' / ' ;

EXEC SQL CONNECT : oracleid;
```

Concurrent Logons

The ORACLE Precompilers support distributed processing via SQL*Net. Your application can concurrently access any combination of local and remote databases or make multiple connections to the same database. In Figure 3-2, an application program communicates with one local and three remote ORACLE databases. ORA2, ORA3, and ORA4 are simply logical names used in CONNECT statements.

Figure 3-2
Connecting via SQL*Net



By eliminating the boundaries in a network between different machines and operating systems, SQL*Net provides a distributed processing environment for ORACLE tools. This section shows you how the ORACLE Precompilers support distributed processing via SQL*Net. You learn how your application can

- directly or indirectly access other databases
- concurrently access any combination of local and remote databases
- make multiple connections to the same database

For details on installing SQL*Net and identifying available databases, refer to the *SQL*Net User's Guide* and the ORACLE installation or user's guide for your system.

Some Preliminaries

The communicating points in a network are called *nodes.* SQL*Net lets you transmit information (SQL statements, data, and status codes) over the network from one node to another.

A *protocol* is a set of rules for accessing a network. The rules establish such things as procedures for recovering after a failure and formats for transmitting data and checking errors. All the examples in this section use the DECnet network access protocol.

The SQL*Net syntax for connecting to the default database on a remote node via DECnet is

```
d: node
```

where *d* specifies the network (DECnet) and *node* specifies the remote node.

The syntax for connecting to a non-default database on a remote node via DECnet is

```
d: node-database
```

where *database* specifies the non-default database.

See the protocol-specific *SQL*Net User's Guide* for the syntax required when using other protocols.

Default Databases and Connections

Each node has a *default* database. If you specify a node but no database **in your** CONNECT statement, you connect to the default database on the named local or remote node. If you specify no database and no node, you connect to the default database on the *current* node. Although it is unnecessary, you can specify the default database and current node in your CONNECT statement.

A *default* connection is made by a CONNECT statement that has no AT clause. The connection can be to any default or non-default database at any local or remote node. SQL statements without an AT clause are executed against the default connection. Conversely, a *non-default* connection is made by a CONNECT statement that has an AT clause. SQL statements with an AT clause are executed against the non-default connection.

All database names must be unique, but two or more database names can specify the same connection. That is, you can have multiple connections to any database on any node.

Explicit Logons

Usually, you establish a connection to ORACLE as follows:

```
EXEC SQL CONNECT :username IDENTIFIED BY : password;
```

You can also use

```
EXEC SQL CONNECT :usr_pwd;
```

where *usr_pwd* contains *username/password.*

You can automatically log onto ORACLE with the userid

```
OPS$username
```

where *username* is your current operating system user or task name and *OPS$username* is a valid ORACLE userid. You simply pass to the precompiled a slash (/) character, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    oracleid      CHARACTER( 1 ) ;
EXEC SQL END DECLARE SECTION;

set oracleid = ' /' ;

EXEC SQL CONNNECT : oracleid;
```

This automatically connects you as user *OPS$username.*

If you do not specify a database and node, you are connected to the default database at the current node. If you want to connect to a different database, you must explicitly identify that database.

With explicit logons, you connect to another database directly, giving the connection a name that will be referenced in SQL statements. You can connect to several databases at the same time and to the same database multiple times.

Single Explicit Logons

In the following example, you connect to a single non-default database at a remote node:

```
-- declare needed host variables
EXEC SQL BEGIN DECLARE SECTION;
    username    CHARACTER (10 ) ;
    password    CHARACTER (10 ) ;
    db_string   CHARACTER;
EXEC SQL END DECLARE SECTION;

set username = 'Scott' ;
set password = 'tiger';
set db_string . 'd:newyork-nondef';

-- give the database connection a unique name
EXEC SQL DECLARE db_name DATABASE;

-- connect to the non-default database
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT db_name USING :db_string;
```

The identifiers in this example serve the following purposes:

- The host variables *username* and *password* identify a valid user.
- The host variable *db_string* contains the SQL*Net syntax for logging onto a non-default database at a remote node using the DECnet protocol.
- The undeclared identifier *db_name* names a non-default connection; it is an identifier used by ORACLE, *not* a host or program variable.

The USING clause specifies the network, machine, and database to be associated with *db_name.* Later, SQL statements using the AT clause (with *db_name)* are executed at the database specified by *db_string.*

Alternatively, you can use a character host variable in the AT clause, as the following example shows:

```
-- declare needed host variables
EXEC SQL BEGIN DECLARE SECTION;
    username   CHARACTER (10 ) ;
    password   CHARACTER(10 );
    db_name    CHARACTER(10 );
    db_string  CHARACTER;
EXEC SQL END DECLARE SECTION;

set username = 'scott';
set password = 'tiger';
set db_name = 'oracle1';
set db_string = 'd:newyork-nondef';

-- connect to the non-default database
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT :db_name USING :db_string;
...
```

If *db_name* is a host variable, the DECLARE DATABASE statement is not needed. Only if *db_name* is an undeclared identifier must you execute a DECLARE *db_name* DATABASE statement before executing CONNECT...AT *db_name* statement.

SQL Operations

If granted the privilege, you can execute any SQL data manipulation statement at the non-default connection. For example, you might execute the following sequence of statements:

```
EXEC SQL AT db_name SELECT . . .
EXEC SQL AT db_name INSERT . . .
EXEC SQL AT db_name UPDATE . . .
```

In the next example, *db_name* is a host variable:

```
EXEC SQL AT : db_name DELETE . . .
```

If *db_name* is a host variable, all database tables referenced by the SQL statement must be defined in DECLARE TABLE statements. Otherwise, the precompiled issues a warning.

Cursor Control

Cursor control statements such as OPEN, FETCH, and CLOSE are exceptions-they never use an AT clause. If you want to associate a cursor with an explicitly identified database, use the AT clause in the DECLARE CURSOR statement, as follows:

```
EXEC SQL AT : db_name DECLARE emp_cursor CURSOR FOR . . .
EXEC SQL OPEN emp_cursor . . .
EXEC SQL FETCH emp_cursor . . .
EXEC SQL CLOSE emp_cursor;
```

If *db_name* is a host variable, its declaration must be within the scope of all SQL statements that refer to the DECLAREd cursor. For example, if you OPEN the cursor in one subprogram, then FETCH from it in another subprogram, you must declare *db_name* globally.

When OPENing, CLOSing, or FETCHing from the cursor, you do not use the AT clause. The SQL statements are executed at the database named in the AT clause of the DECLARE CURSOR statement or at the default database if no AT clause is used in the cursor declaration.

The AT *:host_variable* clause allows you to change the connection associated with a cursor. However, you cannot change the association while the cursor is open. Consider the following example

```
EXEC SQL AT : db_name DECLARE emp_cursor CURSOR FOR . . .

set db_name = ' oracle1' ;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO . . .

set db_name = ' oracle2' ;
EXEC SQL OPEN emp_cursor; -- illegal, cursor still open
EXEC SQL FETCH emp_cursor INTO . . .
```

This is illegal because *emp_cursor* is still open when you try to execute the second OPEN statement. Separate cursors are not maintained for different connections; there is only one *emp_cursor,* which must be closed before it can be reopened for another connection. To debug the last example, simply close the cursor before reopening it, as follows:

```
...
EXEC SQL CLOSE emp_cursor; -- close cursor first

set db_name = ' oracle2' ;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO . . .
```

Dynamic SQL

Dynamic SQL statements are similar to cursor control statements in that some never use the AT clause.

For dynamic SQL Method 1, you must use the AT clause if you want to execute the statement at a non-default connection. An example follows:

```
EXEC SQL AT : db_name EXECUTE IMMEDIATE :slq_stmt;
```

For Methods 2,3, and 4, you use the AT clause only in the DECLARE STATEMENT statement if you want to execute the statement at a non-default connection. All other dynamic SQL statements such as PREPARE, DESCRIBE, OPEN, FETCH, and CLOSE never use the AT clause. The next example shows Method 2:

```
EXEC SQL AT : db_name DECLARE slq_stmt STATEMENT;
EXEC SQL PREPARE slq_stmt FROM : sql_string;
EXEC SQL EXECUTE slq_stmt;
```

The following example shows Method 3:

```
EXEC SQL AT : db_name DECLARE slq_stmt STATEMENT;
EXEC SQL PREPARE slq_stmt FROM : sql_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR slq_stmt;
EXEC SQL OPEN emp_cursor . . .
EXEC SQL FETCH emp_cursor INTO . . .
EXEC SQL CLOSE emp_cursor;
```

Multiple Explicit Logons

You can use the AT *db_name* clause for multiple explicit logons, just as you would for a single explicit logon. In the following example, you connect to two non-default databases concurrently

```
-- declare needed host variables
EXEC SQL BEGIN DECLARE SECTION;
    username     CHARACTER (10 ) ;
    password     CHARACTER (10 );
    db_string1 CHARACTER ( 20 );
    db_string2 CHARACTER ( 20 ) ;
EXEC SQL END DECLARE SECTION;
...
set username = ‘ scott’ ;
set password = ‘ tiger’ ;
set db_string1 = ‘d: newyork-nondef1‘ ;
set db_string2 = ‘d: chicago-nondef2 ‘ ;

-- give each database connect ion a unique name
EXEC SQL DECLARE db_name1 DATABASE;
EXEC SQL DECLARE db_name2 DATABASE;
```

```
-- connect to the two non-default databases
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT db_name1 USING :db_string1;
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT db_name2 USING :db_string2;
```

The undeclared identifiers *db_name1* and *db_name2* are used to name the default databases at the two non-default nodes so that later SQL statements can refer to the databases by name.

Alternatively, you can use a host variable in the AT clause, as the following example shows:

```
-- declare needed host variables
EXEC SQL BEGIN DECLARE SECTION;
    username   CHARACTER(10) ;
    password   CHARACTER(10 );
    db_name    CHARACTER  ;
    db_string  CHARACTER;
EXEC SQL END DECLARE SECTION;
...
set username = 'Scott' ;
set password = 'tiger';

FOR EACH non-default database
    -- get next database name and SQL*Net string
    display 'Database Name? ';
    read db_name;
    display 'SQL*Net String? ';
    read db_string;
    -- connect to the non-default database
    EXEC SQL CONNECT :username IDENTIFIED BY :password
        AT :db_name USING :db_string;
ENDFOR;
...
```

You can also use this method to make multiple connections to the same database, as the following example shows:

```
set username = 'scott';
set password = 'tiger';
set db_string . 'd:newyork-nondef ';

FOR EACH non-default database
    -- get next database name
    display 'Database Name? ';
    read db_name;
    -- connect to the non-default database
    EXEC SQL CONNECT :username IDENTIFIED BY :password
        AT :db_name USING :db_string;
ENDFOR;
```

You must use different database names for the connections, even though they use the same SQL*Net string. However, you can connect twice to the same database using just one database name because that name identifies the default and non-default databases.

Ensuring Data Integrity

Your application program must ensure the integrity of transactions that manipulate data at two or more remote databases. That is, the program must commit or rollback *all* SQL statements in the transactions. This might be impossible if the network fails or one of the systems crashes.

For example, suppose you are working with two accounting databases. You debit an account on one database and credit an account on the other database, then issue a COMMIT at each database. It is up to your program to ensure that both transactions are committed or rolled back.

Implicit Logons

Implicit logons are supported through the ORACLE distributed query facility, which does not require explicit logons, but only supports the SELECT statement. A distributed query allows a single SELECT statement to access data on one or more non-default databases.

The distributed query facility depends on database links, which assign a name to a CONNECT *statement* rather than to the connection itself. At run time, the embedded SELECT statement is executed by the specified ORACLE Server, which *implicitly connects* to the non-default database(s) to get the required data.

Single Implicit Logons

In the next example, you connect to a single non-default database. First, your program executes the following statement to define a database link (database links are usually established interactively by the DBA or user):

```
EXEC SQL CREATE DATABASE LINK db_link
    CONNECT TO username IDENTIFIED BY password
        USING 'd: newyork-nondef';
```

Then, the program can query the non-default EMP table using the database link, as follows:

```
EXEC SQL SELECT ENAME, JOB INTO : emp_name, : job_title
    FROM emp@db_link
    WHERE DEPTNO = : dept_number;
```

The database link is not related to the database name used in the AT clause of an embedded SQL statement It simply tells ORACLE where the non-default database is located, the path to it, and what ORACLE username and password to use. The database link is stored in the data dictionary until it is explicitly dropped.

In our example, the default ORACLE Server logs on to the non-default database via SQL*Net using the database link *db_link*. The query is submitted to the default Server, but is "forwarded" to the non-default database for execution.

To make referencing the database link easier, you can create a synonym as follows (again, this is usually done interactively):

```
EXEC SQL CREATE SYNONYM emp FOR emp@db_link;
```

Then, your program can query the non-default EMP table, as follows:

```
EXEC SQL SELECT ENAME, JOB INTO : emp_name, : job_title
    FROM emp
    WHERE DEPTNO = : dept_number;
```

This provides location transparency for *emp*.

Multiple Implicit Logons

In the following example, you connect to two non-default databases concurrently. First, you execute the following sequence of statements to define two database links and create two synonyms:

```
EXEC SQL CREATE DATABASE LINK db_link1
    CONNECT TO username1 IDENTIFIED BY password1
        USING 'd: newyork-nondef ' ;
EXEC SQL CREATE DATABASE LINK db_link2
    CONNECT TO username2 IDENTIFIED BY password2
        USING 'd: chicago-nondef ' ;
EXEC SQL CREATE SYNONYM emp FOR emp@3b_link1;
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2;
```

Then, your program can query the non-default EMP and DEPT tables, as follows:

```
EXEC SQL SELECT ENAME , JOB , SAL, LOC
    FROM emp, dept
    WHERE emp.DEPTNO = dept.DEPTNO AND DEPTNO = :dept_number;
```

ORACLE executes the query by performing a join between the non-default EMP table at *db_link1* and the non-default DEPT table at *db_link2*

Embedding OCI (ORACLE Call Interface) Calls

The ORACLE Precompilers let you embed OCI calls in your host program. Just take the following steps:

1. Declare the OCI Logon Data Area (LDA) outside the Declare Section. Typically, the LDA is declared as a 32-element array of 2-byte integers. For details, see the *Programmer's Guide to the ORACLE Call Interfaces.*
2. Connect to ORACLE using the embedded SQL statement CONNECT, *not* the OCI call OLON or ORLON.
3. Call the ORACLE runtime library routine SQLLDA to store the connect information in the LDA.

That way, the ORACLE Precompiled and the OCI "know" that they are working together. However, there is no sharing of ORACLE cursors.

You need not worry about declaring the OCI Host Data Area (HDA) because the ORACLE runtime library manages connections and maintains the HDA for you.

Setting Up the LDA

You set up the LDA by issuing the OCI call

```
SQLLDA ( lda ) ;
```

where *lda* identifies the LDA data structure. The format of this call is language- and system-dependent. Refer to the *Programmer's Guide to the ORACLE Call Interfaces* and the ORACLE installation or user's guide for your system.

If the CONNECT statement fails, the *lda_rc* field in the *lda* is set to 1012 to indicate the error.

Remote and Multiple Connections

A call to SQLLDA sets up an LDA for the connection used by the most recently executed SQL statement. To setup the different LDAs needed for additional connections, just call SQLLDA with a different *lda* after each CONNECT. In the following example, you connect to two non-default databases concurrently

```
EXEC SQL BEGIN DECLARE SECTION;
    username     CHARACTER (10 ) ;
    password     CHARACTER (10 ) ;
    db_string1 CHARACTER ( 20 ) ;
    db_string2 CHARACTER ( 20 ) ;
EXEC SQL END DECLARE SECTION;

lda1 INTEGER ( 32 ) ; -- array of 2-byte integers
lda2 INTEGER ( 32 ) ;
...
set username = 'SCOTT' ;
set password = 'TIGER' ;

set db_string1 =  ' D :NEWYORK-NONDEF1 ' ;
set db_string2 =  'D: CHICAGO-NONDEF2' ;

-- give each database connection a unique name
EXEC SQL DECLARE db_name1 DATABASE;
EXEC SQL DECLARE db_name2 DATABASE;

-- connect to first non-default database
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT db_name1 USING :db_string1;
-- set up first LDA for OCI use
SQLLDA(lda1);

-- connect to second non-default database
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT db_name2 USING :db_string2;
-- set up second LDA for OCI use
SQLLDA(lda2);
```

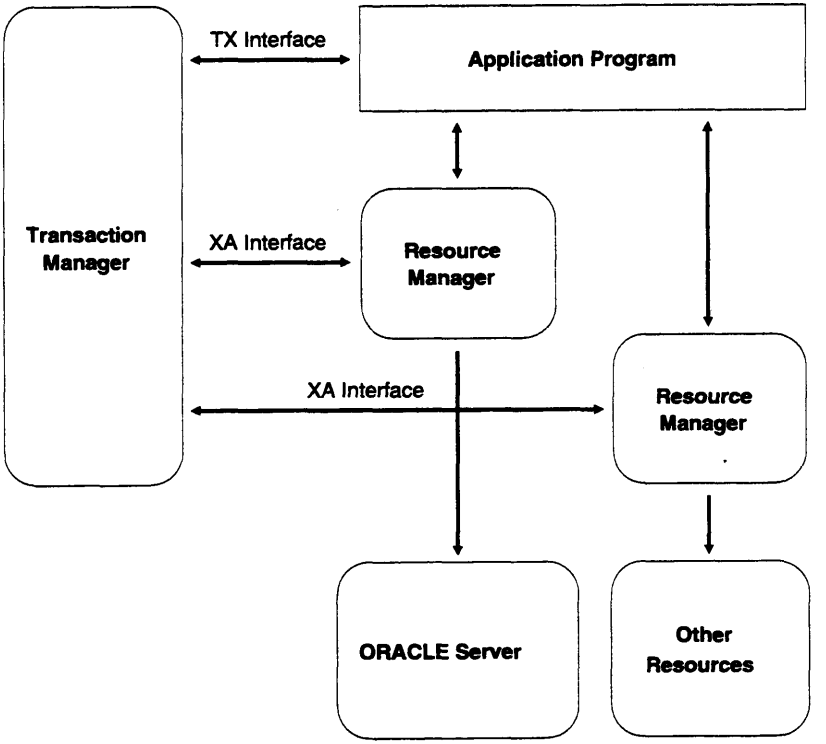
Remember, do not declare *db_name1* and *db_name2* in the Declare Section because they are not host variables. You use them only to name the default databases at the two non-default nodes, so that later SQL statements can refer to the databases by name.

Developing X/Open Applications

X/Open applications run in a distributed transaction processing (DTP) environment. In an abstract model, an X/Open application calls on *resource managers* (RMs) to provide a variety of services. For example, a database resource manager provides access to data in a database. Resource managers interact with a transaction *manager* (TM), which controls all transactions for the application.

Figure 3-3 shows one way that components of the DTP model can interact to provide efficient access to data in an ORACLE database. The DTP model specifies the *XA interface* between resource managers and the transaction manager. Oracle supplies an XA-compliant library, which you must link to your X/Open application. Also, you must specify the *native inferface* between your application program and the resource managers.

Figure 3-3 Hypothetical DTP Model



The DTP model that specifies how a transaction manager and resource managers interact with an application program is described in the X/Open guide *Distributed Transaction Processing Reference Model* and related publications, which you can obtain by writing to

X/Open Company Ltd.
1010 El Camino Real, Suite 380
Menlo Park, CA 94025

For instructions on using the XA interface, see your Transaction Processing (TP) Monitor user's guide.

ORACLE-specific Issues

You can use the ORACLE Precompilers to develop applications that comply with the X/Open standards. However, you must meet the following requirements.

Connecting to ORACLE

The X/Open application does not establish and maintain connections to a database. Instead, the transaction manager and the XA interface, which is supplied by Oracle, handle database connections and disconnections transparently. So, normally an X/Open-compliant application does not execute CONNECT statements.

Transaction Control

The X/Open application must not execute statements such as COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION that affect the state of global transactions. For example, the application must not execute the COMMIT statement because the transaction manager handles commits. Also, the application must not execute SQL data definition statements such as CREATE, ALTER, and RENAME because they issue an implicit COMMIT.

The application can execute an internal ROLLBACK statement if it detects an error that prevents further SQL operations. However, this might change in later versions of the XA interface.

OCI Calls

If you want your X/Open application to issue OCI calls, you must use the runtime library routine SQLLD2, which sets up an LDA for a specified connection established through the XA interface. For a description of the SQLLD2 call, see the *Programmer's Guide to the ORACLE Call Interlaces.* Note that the following OCI calls cannot be issued by an X/Open application. OCOM, OCON, OCOF, ORLON, OLON, OLOGOF.

Linking

To get XA functionality, you must link the XA library to your X/Open application object modules. For instructions, see the ORACLE installation or user's guide for your system.

CHAPTER

4

USING EMBEDDED SQL

This chapter helps you to understand and apply the basic techniques of embedded SQL programming. You learn how to use host variables, indicator variables, and the fundamental SQL commands that insert, update, select, and delete ORACLE data.

Using Host Variables

ORACLE uses host variables to pass data and status information to you program; your program uses host variables to pass data to ORACLE.

Output versus Input Host Variables

Depending on how they are used, host variables are called output or input host variables.

Host variables in the INTO clause of a SELECT or FETCH statement are called *output* host variables because they hold column values output by ORACLE. ORACLE assigns the column values to corresponding output host variables in the INTO clause.

All other host variables in a SQL statement are called *input* host variables because your program inputs their values to ORACLE. For example, you use input host variables in the VALUES clause of an INSERT statement and in the SET clause of an UPDATE statement. They are also used in the WHERE, HAVING, and FOR clauses. Input host variables can appear in a SQL statement wherever a value or expression is allowed.

You cannot use input host variables to supply SQL keywords or the names of database objects. Thus, you cannot use input host variables in data definition statements such as ALTER, CREATE, and DROP. In the following example, the DROP TABLE statement is *invalid:*

```
EXEC SQL BEGIN DECLARE SECTION;
    table_name    CHARACTER (30) ;
EXEC SQL END DECLARE SECTION;

display  'Table name? ' ;
read table_name;

EXEC SQL DROP TABLE : table_name; -- host variable not allowed
```

Before ORACLE executes a SQL statement containing input host variables, your program must assign values to them. An example follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_number    INTEGER ;
    emp_name      CHARACTER (20) ;
EXEC SQL END DECLARE SECTION;
```

```
.. get values for input host variables
display 'Employee number? ';
read emp_number;
display 'Employee name? ';
read emp_name;

EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
    VALUES (:emp_number, :emp_name);
```

Notice that the input host variables in the VALUES clause of the INSERT statement are prefixed with colons.

Using Indicator Variables

You can associate any host variable with an optional indicator variable. Each time the host variable is used in a SQL statement, a result code is stored in its associated indicator variable. Thus, indicator variables let you monitor host variables.

You use indicator variables in the VALUES or SET clause to assign nulls to input host variables and in the INTO clause to detect nulls or truncated values in output host variables.

For input host variables, the values your program can assign to an indicator variable have the following meanings:

-1	ORACLE will assign a null to the column, ignoring the value of the host variable.
≥ 0	ORACLE will assign the value of the host variable to the column.

For output host variables, the values ORACLE can assign to an indicator variable have the following meanings:

-1	The column value is null, so the value of the host variable is indeterminate.
0	ORACLE assigned an intact column value to the host variable.
>0	ORACLE assigned a truncated column value to the host variable. The integer returned by the indicator variable is the original length of the column value, and SQLCODE in SQLCA is set to zero.

Remember, an indicator variable must be defined in the Declare Section as a 2-byte integer and, in SQL statements, must be prefixed with a colon and appended to its host variable.

Inserting Nulls

You can use indicator variables to INSERT nulls. Before the INSERT, for each column you want to be null, set the appropriate indicator variable to -1, as shown in the following example:

```
set ind_comm = -1;

EXEC SQL INSERT INTO EMP (EMPNO, COMM)
    VALUES (:emp_number, : commission: ind_com);
```

The indicator variable *ind_comm* specifies that a null is to be stored in the COMM column.

You can hardcode the null instead, as follows:

```
EXEC SQL INSERT INTO EMP (EMPNO, COMM)
    VALUES (:emp_number, NULL ) ;
```

While this is less flexible, it might be more readable.

Typically, you insert nulls conditionally, as the next example shows:

```
display  ' Enter employee number or 0 if not available: ' ;
read   emp_number;

IF emp_number = 0 THEN
    set ind_empnum = -1;
ELSE
    set ind_empnum = 0;
ENDIF;

EXEC SQL INSERT INTO EMP ( EMPNO, SAL)
    VALUES (: emp_number: ind_empnum, : salary);
```

Handling Returned Nulls

You can also use indicator variables to manipulate returned nulls, as the following example shows:

```
EXEC SQL SELECT ENAME, SAL, COMM
    INTO :emp_name, : salary, : commission:ind_comm
    FROM EMP
    WHERE  EMPNO=  :emp_number;

IF ind_comm = -1 THEN
    set pay = salary;    -- commission is null; ignore it
ELSE
    set pay = salary + commission;
ENDIF;
```

Fetching Nulls

When MODE= {ANSI13 | ORACLE}, you can SELECT or FETCH nulls into a host variable not associated with an indicator variable, as the following example shows:

```
-- assume that commission is NULL
EXEC SQL SELECT ENAME , SAL , COMM
    INTO : emp_name, : salary, : commission
    FROM EMP
    WHERE EMPNO =  : emp_number;
```

SQLCODE in the SQLCA is set to zero indicating that ORACLE executed the statement without detecting an error or exception.

However, when MODE={ANSI | ANSI14}, if you SELECT or FETCH nulls into a host variable not associated with an indicator variable, ORACLE issues the following error message:

```
ORA-01405: fetched column value is NULL
```

For more information about the MODE option, see the section "Using the Precompiled Options" in Chapter 11.

Testing for Nulls

You can use indicator variables in the WHERE clause to test for nulls, as the following example shows:

```
EXEC SQL SELECT ENAME, SAL
    INTO : emp_name, : salary
    FROM EMP
    WHERE : commission: ind_comm IS NULL . . .
```

However, you cannot use a relational operator to compare nulls with each other or with other values. For example, the following SELECT statement fails if the COMM column contains one or more nulls:

```
EXEC SQL SELECT ENAME , SAL
    INTO : emp_name, : salary
    FROM EMP
    WHERE COMM =  : commission: ind_comm;
```

The next example shows how to compare values for equality when some of them might be nulls:

```
EXEC SQL SELECT ENAME , SAL
    INTO : emp_name, : salary
    FROM EMP
    WHERE ( COMM =  : commission) OR ( (COMM IS NULL) AND
        (commission: ind_comm IS NULL) ) ;
```

Fetching Truncated Values

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host variable not associated with an indicator variable, ORACLE issues the following error message:

```
ORA-01406: fetched column value was truncated
```

However, when MODE= {ANSI | ANSI14 | ANSI13}, no error is generated.

The Basic SQL Statements

Executable SQL statements let you query, manipulate, and control ORACLE data and create, define, and maintain ORACLE objects such as tables, views, and indexes. This chapter focuses on the statements that query and manipulate data.

When executing a data manipulation statement such as INSERT, UPDATE, or DELETE, your only concern, besides setting the values of any input host variables, is whether the statement succeeds or fails. To find out, you simply check the SQLCA. (Executing any SQL statement sets the SQLCA variables.) You can check in the following two ways:

- implicit checking with the WHENEVER statement
- explicit checking of SQLCA variables

For more information about the SQLCA and the WHENEVER statement, see Chapter 7, “Handling Runtime Errors.”

When executing a SELECT statement (query), however, you must also deal with the rows of data it returns. Queries can be classified as follows:

- queries that return no rows (that is, merely check for existence)
- queries that return only one row
- queries that return more than one row

Queries that return more than one row require explicitly declared cursors or the use of *host arrays* (host variables declared as arrays).

Note Host arrays let you process “batches” of rows. For more information, see Chapter 8, “Using Host Arrays.” This chapter assumes the use of scalar host variables.

The following embedded SQL statements let you query and manipulate ORACLE data:

SELECT	Returns rows from one or more tables.
INSERT	Adds new rows to a table.
UPDATE	Modifies rows in a table.
DELETE	Removes unwanted rows from a table.

The following embedded SQL statements let you define and manipulate an explicit cursor:

DECLARE	Names the cursor and associates it with a query.
OPEN	Executes the query and identifies the active set.
FETCH	Advances the cursor and retrieves each row in the active set, one by one.
CLOSE	Disables the cursor (the active set becomes undefined).

In the coming sections, first you learn how to code INSERT, UPDATE, DELETE, and single-row SELECT statements. Then, you progress to multirow SELECT statements. For a detailed discussion of each statement and its clauses, see the *ORACLE7 Server SQL Language Reference Manual.* For the syntax of each statement, refer to Appendix B.

Using the SELECT Statement

Querying the database is a common SQL operation. To issue a query you use the SELECT statement. In the following example, you query the EMP table:

```
EXEC SQL SELECT ENAME , JOB , SAL + 2000
    INTO : emp_name, : job_title, : salary
    FROM EMP
    WHERE EMPNO = : emp_number;
```

The column names and expressions following the keyword SELECT make up the *select list*. The select list in our example contains three items. Under the conditions specified in the WHERE clause (and following clauses, if present), ORACLE returns column values to the host variables in the INTO clause.

The number of items in the select list should equal the number of host variables in the INTO clause, so there is a place to store every returned value.

In the simplest case, when a query returns one row, its form is that shown in the last example. However, if a query can return more than one row, you must FETCH the rows using a cursor or SELECT them into a host-variable *array*. *Cursors* and the FETCH statement are discussed later in this chapter; array processing is discussed in Chapter 8, “Using Host Arrays.”

If a query is written to return only one row but might actually return several rows, the result of the SELECT is indeterminate. Whether this causes an error depends on how you specify the SELECT_ERROR option. The default value, YES, generates an error if more than one row is returned.

Available Clauses

You can use all of the following standard SQL clauses in your SELECT statements:

- INTO
- FROM
- WHERE
- CONNECT BY
- START WITH
- GROUP BY
- HAVING
- ORDER BY
- FOR UPDATE OF

Except for the INTO clause, the text of embedded SELECT statements can be executed and tested interactively using SQL*Plus. In SQL*Plus, you use substitution variables or constants instead of input host variables.

Using the INSERT Statement

You use the INSERT statement to add rows to a table or view. In the following example, you add a row to the EMP table:

```
EXEC SQL INSERT INTO EMP ( EMPNO , ENAME , SAL , DEPTNO )
    VALUES ( : emp_number, : emp_name, : salary, : dept_number ) ;
```

Each column you specify in the *column list* must belong to the table named in the INTO clause. The VALUES clause specifies the row of values to be inserted. The values can be those of constants, host variables, SQL expressions, or pseudocolumns such as USER and SYSDATE.

The number of values in the VALUES clause must equal the number of names in the column list. However, you can omit the column list if the VALUES clause contains a value for each column in the table.

Using Subqueries

A *subquery* is a nested SELECT statement. Subqueries let you conduct multipart searches. They can be used to

- supply values for comparison in the WHERE, HAVING, and START WITH clauses of SELECT, UPDATE, and DELETE statements
- define the set of rows to be inserted by a CREATE TABLE or INSERT statement
- define values for the SET clause of an UPDATE statement

In the following example, you use a subquery in the VALUES clause of an INSERT statement to copy rows from one table to another:

```
EXEC SQL INSERT INTO EMP2 ( EMPNO , ENAME , SAL , DEPTNO )
    SELECT EMPNO , ENAME , SAL , DEPTNO FROM EMP
        WHERE JOB = : job_title;
```

Notice how the INSERT statement uses the subquery to obtain intermediate results.

Using the UPDATE Statement

You use the UPDATE statement to change the values of specified columns in a table or view. In the following example, you UPDATE the SAL and COMM columns in the EMP table:

```
EXEC SQL UPDATE EMP
    SET SAL = : salary, COMM = : commission
    WHERE EMPNO = :emp_number;
```

You can use the optional WHERE clause to specify the conditions under which rows are UPDATEd. See the section “Using the WHERE Clause” later in this chapter.

The SET clause lists the names of one or more columns for which you must provide values. You can use a subquery to provide the values, as the following example shows:

```
EXEC SQL UPDATE EMP
    SET SAL = ( SELECT AVG ( SAL ) *1.1 FROM EMP WHERE DEPTNO = 20 )
    WHERE EMPNO = :emp_number;
```

Using the DELETE Statement

You use the DELETE statement to remove rows from a table or view. In the following example, you delete all employees in a given department from the EMP table:

```
EXEC SQL DELETE FROM EMP
    WHERE DEPTNO = : dept_number;
```

You can use the optional WHERE clause to specify the condition under which rows are DELETEd.

Using the WHERE Clause

You use the WHERE clause to SELECT, UPDATE, or DELETE only those rows in a table or view that meet your search condition. The WHERE-clause *search condition* is a Boolean expression, which can include scalar host variables, host arrays (not in SELECT statements), and subqueries.

If you omit the WHERE clause, all rows in the table or view are processed. If you omit the WHERE clause in an UPDATE or DELETE statement, ORACLE sets the fifth element of SQLWARN in the SQLCA to 'W' to warn that all rows were processed.

Using Cursors

When a query returns multiple rows, you can explicitly define a cursor to

- process beyond the first row returned by the query
- keep track of which row is currently being processed

Or, you can use host arrays; see Chapter 8, "Using Host Arrays."

A cursor identifies the current row in the set of rows returned by the query. This allows your program to process the rows one at a time. The following statements let you define and manipulate a cursor:

- DECLARE
- OPEN
- FETCH
- CLOSE

First you use the DECLARE statement to name the cursor and associate it with a query.

The OPEN statement executes the query and identifies all the rows that meet the query search condition. These rows form a set called the active set of the cursor. After OPENing the cursor, you can use it to retrieve the rows returned by its associated query.

Rows of the active set are retrieved one by one (unless you use host arrays). You use a FETCH statement to retrieve the current row in the active set. You can execute FETCH repeatedly until all rows have been retrieved.

When done FETCHing rows from the active set, you disable the cursor with a CLOSE statement, and the active set becomes undefined.

The following sections show you how to use these cursor control statements in your application program.

Using the DECLARE Statement

You use the DECLARE statement to define a cursor by giving it a name and associating it with a query, as the following example shows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , EMPNO , SAL
    FROM EMP
    WHERE DEPTNO =  : dept_number;
```

The cursor name is an identifier used by the precompiled, not a host or program variable, and should not be defined in the Declare Section. Cursor names cannot be hyphenated. They can be any length, but only the first 31 characters are significant. For ANSI compatibility, use cursor names no longer than 18 characters.

The SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement.

Because it is declarative, the DECLARE statement must physically (not just logically) precede all other SQL statements referencing the cursor. That is, forward references to the cursor are not allowed. In the following example, the OPEN statement is misplaced:

```
...
EXEC SQL OPEN emp_cursor;

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , EMPNO , SAL
    FROM EMP
    WHERE ENAME = : emp_name;
```

The cursor control statements (DECLARE, OPEN, FETCH, CLOSE) must all occur within the same precompiled unit. For example, you cannot DECLARE a cursor in file A, then OPEN it in file B.

Your host program can DECLARE as many cursors as it needs. However, in a given file, every DECLARE statement must be unique. That is, you cannot DECLARE two cursors with the same name in one precompilation unit, even across blocks or procedures, because the scope of a cursor is global within a file.

If you will be using many cursors, you might want to specify the MAXOPENCURSORS option. For more information, see Chapter 11, "Running the ORACLE Precompilers," and Appendix E, "Performance Tuning."

Using the OPEN Statement

You use the OPEN statement to execute the query and identify the active set. In the following example, you OPEN a cursor named *emp_cursor*.

```
EXEC SQL OPEN emp_cursor;
```

OPEN positions the cursor just before the first row of the active set. It also zeroes the rows-processed count kept by the third element of SQLERRD in the SQLCA. However, none of the rows is actually retrieved at this point. That will be done by the FETCH statement.

Once you OPEN a cursor, the query's input host variables are not reexamined until you reOPEN the cursor. Thus, the active set does not change. To change the active set, you must reOPEN the cursor.

Generally, you should CLOSE a cursor before reOPENing it. However, if you specify MODE=ORACLE (the default), you need not CLOSE a cursor before reOPENing it. This can boost performance; for details, see Appendix E, "Performance Tuning."

The amount of work done by OPEN depends on the values of three precompiled options: HOLD_CURSOR, RELEASE_CURSOR, and MAXOPENCURSORS. For more information, see the section "Using the Precompiled Options" in Chapter 11.

Using the FETCH Statement

You use the FETCH statement to retrieve rows from the active set and specify the output host variables that will contain the results. Recall that the SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement. In the following example, you FETCH INTO three host variables:

```
EXEC SQL FETCH emp_cursor
    INTO : emp_name, : emp_number, : salary;
```

The cursor must have been previously DECLAREd and OPENed. The first time you execute FETCH, the cursor moves from before the first row in the active set to the first row. This row becomes the current row. Each subsequent execution of FETCH advances the cursor to the next row in the active set, changing the current row. The cursor can only move forward in the active set. To return to a row that has already been FETCHed, you must reOPEN the cursor, then begin again at the first row of the active set.

If you want to change the active set, you must assign new values to the input host variables in the query associated with the cursor, then reOPEN the cursor. When MODE= {ANSI | ANSI14 | ANSI13}, you must CLOSE the cursor before reOPENing it.

As the next example shows, you can FETCH from the same cursor using different sets of output host variables. However, corresponding host variables in the INTO clause of each FETCH statement must have the same datatype.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , SAL FROM EMP WHERE DEPTNO = 20 ;
...
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO . . .
LOOP
    EXEC SQL FETCH emp_cursor INTO : emp_name1, : salary1;
    EXEC SQL FETCH emp_cursor INTO : emp_name2, : salary2;
    EXEC SQL FETCH emp_cursor INTO : emp_name3, : salary3;
    ...
ENDLOOP;
```

If the active set is empty or contains no more rows, FETCH returns the "no data found" ORACLE error code to SQLCODE in the SQLCA. The status of the output host variables is indeterminate. (In a typical program, the WHENEVER NOT FOUND statement detects this error.) To continue using the cursor, you must reOPEN it.

Using the CLOSE Statement

When done FETCHing rows from the active set, you CLOSE the cursor to free the resources, such as storage, acquired by OPENing the cursor. When a cursor is closed, parse locks are released. What resources are freed depends on how you specify the HOLD_CURSOR and RELEASE_CURSOR options. In the following example, you CLOSE the cursor named *emp_cursor:*

```
EXEC SQL CLOSE emp_cursor;
```

You cannot FETCH from a closed cursor because its active set becomes undefined. If necessary, you can reOPEN a cursor (with new values for the input host variables, for example).

When MODE={ANSI13 | ORACLE}, issuing a COMMIT or ROLLBACK closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by COMMIT or ROLLBACK and if open, remain open. However, when MODE= {ANSI | ANSI14}, issuing a COMMIT or ROLLBACK closes *all* explicit cursors. For more information about COMMIT and ROLLBACK, see Chapter 6, "Defining and Controlling Transactions." For more information about the CURRENT OF clause, see the next section.

Using the CURRENT OF Clause

You use the CURRENT OF *cursor_name* clause in a DELETE or UPDATE statement to refer to the latest row FETCHed from the named cursor. The cursor must be open and positioned on a row. If no FETCH has been done or if the cursor is not open, the CURRENT OF clause results in an error and processes no rows.

The FOR UPDATE OF clause is optional when you DECLARE a cursor . that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement. The CURRENT OF clause signals the precompiled to add a FOR UPDATE clause if necessary. For more information, see the section "Using the FOR UPDATE OF Clause" in Chapter 6.

In the following example, you use the CURRENT OF clause to refer to the latest row FETCHed from a cursor named *emp_cursor:*

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , SAL FROM EMP WHERE JOB = ' CLERK '
        FOR UPDATE OF SAL;
...
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO . . .
LOOP
    EXEC SQL FETCH emp_cursor INTO : emp_name, : salary;
    ...
    EXEC SQL UPDATE EMP SET SAL = : new_salary
        WHERE CURRENT OF emp_cursor;
ENDLOOP;
```

Restrictions

An explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. AU rows are locked at the OPEN, not as they are FETCHed, and are released when you COMMIT or ROLLBACK. Therefore, you cannot FETCH from a FOR UPDATE cursor after a COMMIT.

Also, you cannot use host arrays with the CURRENT OF clause. For an alternative, see the section "Mimicking CURRENT OF' in Chapter 8.

Furthermore, you cannot reference multiple tables in an associated FOR UPDATE OF clause, which means that you cannot do joins with the CURRENT OF clause.

Finally, you cannot use dynamic SQL with the CURRENT OF clause.

Using All the Cursor Statements

The following example shows the typical sequence of cursor control statements in an application program:

```
...
-- define a cursor
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, JOB
    FROM EMP
    WHERE EMPNO = :emp_number
    FOR UPDATE OF JOB;

-- open the cursor and identify the active set
EXEC SQL OPEN emp_cursor;

-- exit if the last row was already fetched
EXEC SQL WHENEVER NOT FOUND GOTO no_more;

-- fetch and process data in a loop
LOOP
    EXEC SQL FETCH emp_cursor INTO :emp_name, :job_title;

    -- optional host-language statements that operate on
       the FETCHed data

    EXEC SQL UPDATE EMP
        SET JOB = :new_job_title
        WHERE CURRENT OF emp_cursor;
ENDLOOP;
...
no_more:
    -- disable the cursor
    EXEC SQL CLOSE emp_cursor;
    EXEC SQL COMMIT WORK RELEASE;
    ...
```

A Complete Example

The following program illustrates the use of a cursor and the FETCH statement. The program prompts for a department number, then displays the names of all employees in that department.

All FETCHes except the final one return a row and, if no errors were detected during the FETCH, a success status code. The final FETCH fails and returns the "no data found" ORACLE error code to SQLCODE in the SQLCA. The cumulative number of rows actually FETCHed is found in the third element of SQLERRD in the SQLCA.

```
-- declare host variables
EXEC SQL BEGIN DECLARE SECTION;
    username       CHARACTER (20);
    password       CHARACTER (20);
    emp_name       CHARACTER (10);
    dept_number    INTEGER;
EXEC SQL END DECLARE SECTION;

-- copy in the SQL Communicant ions Area
EXEC SQL INCLUDE SQLCA;

display  ' Username? ' ;
read username;
display  ' Password? ' ;
read password;

-- handle processing errors
EXEC SQL WHENEVER SQLERROR GOTO sql_error;

-- log on to ORACLE
EXEC SQL CONNECT : username IDENTIFIED BY : password;

display  ' Connected to ORACLE' ;

-- declare a cursor
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME
    FROM EMP
    WHERE DEPTNO =  :dept_number;

display  'Department number? ' ;
read  dept_number;

-- open the cursor and identify the active set
EXEC SQL OPEN emp_cursor;

-- exit if the last row was already fetched
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
```

```
display ‘Employee Name’;
display '-------------  ';

-- fetch and process data in a loop
LOOP
    EXEC SQL FETCH emp_cursor INTO :emp_name;
    display emp_name;
ENDLOOP;

no_more:
    EXEC SQL CLOSE emp_cursor;
    EXEC SQL COMMIT WORK RELEASE;
    display ‘End of program’;
    exit program;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    display ‘Processing error’;
    exit program with an error;
```

CHAPTER

5

USING EMBEDDED PL/SQL

This chapter shows you how to improve performance by embedding PL/SQL transaction processing blocks in your program. After pointing out the advantages of PL/SQL, this chapter discusses the following subjects:

- embedding PL/SQL blocks
- using host variables
- using indicator variables
- using host arrays
- using cursors
- creating and calling PL/SQL stored subprograms
- using dynamic SQL

Advantages of PL/SQL

This section looks at some of the features and benefits offered by PL/SQL, such as

- better performance
- integration with ORACLE
- cursor FOR loops
- procedures and functions
- packages
- PL/SQL tables
- user-defined records

For more information about PL/SQL, seethe *PL/SQL User's Guide and Reference.*

Better Performance

PL/SQL can help you reduce overhead, improve performance, and increase productivity. For example, without PL/SQL, ORACLE must process SQL statements one at a time. Each SQL statement results in another call to the Server and higher overhead. However, with PL/SQL, you can send an entire block of SQL statements to the Server. This minimizes communication between your application and ORACLE.

Integration with ORACLE

PL/SQL is tightly integrated with the ORACLE Server. For example, most PL/SQL datatypes are native to the ORACLE data dictionary. Furthermore, you can use the % TYPE attribute to base variable declarations on column definitions stored in the data dictionary, as the following example shows: ,

```
job_title  emp.job%TYPE;
```

That way, you need not know the exact datatype of the column. Furthermore, if a column definition changes, the variable declaration changes accordingly and automatically. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes.

Cursor FOR Loops

With PL/SQL, you need not use the DECLARE, OPEN, FETCH, and CLOSE statements to define and manipulate a cursor. Instead, you can use a cursor FOR loop, which implicitly declares its loop index as a record, opens the cursor associated with a given query, repeatedly fetches data from the cursor into the record, then closes the cursor. An example follows:

```
DECLARE
    ...
BEGIN
    FOR emprec IN (SELECT empno, sal, comm FROM emp) LOOP
        IF emprec. comm / emprec. sal > 0.25 THEN . . .
        ...
    END LOOP;
END;
```

Notice that you use dot notation to reference fields in the record.

Procedures and Functions

PL/SQL has two types of subprograms called *procedures* and *functions,* which aid application development by letting you isolate operations. Generally, you use a procedure to perform an action and a function to compute a value.

Procedures and functions provide *extensibility*. That is, they let you tailor the PL/SQL language to suit your needs. For example, if **you** need a procedure that creates anew department, just write your own as follows:

```
PROCEDURE  create_dept
    (new_dname   IN CHAR(14),
     new_loc     IN CHAR(13),
     new_dept no OUT NUMBER(2)) IS
BEGIN
    SELECT  deptno_seq.NEXTVAL  INTO  new_deptno  FROM  dual;
    INSERT INTO dept VALUES ( new_deptno, new_dname, new_loc);
END create_dept;
```

When called, this procedure accepts anew department name and location, selects the next value in a department-number database sequence, inserts the new number, name, and location into the *dept* table, then returns the new number to the caller.

You use *parameter modes* to define the behavior of formal parameters. There are three parameter modes IN (the default), OUT, and IN OUT. An IN parameter lets you pass values to the subprogram being called. An OUT parameter lets you return values to the caller of a subprogram. An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

The datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Table 3-2 in Chapter 3 shows the legal conversions between datatypes.

Packages

PL/SQL lets you bundle logically related types, program objects, and subprograms into a *package. If* you have the Procedural Database Extension, packages can be compiled and stored in an ORACLE database, where their contents can be shared by many applications.

Packages usually have two parts: a specification and a body. The specification is the interface to your applications; it declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The *body* defines cursors and subprograms and so implements the specification. In the following example, you "package" two employment procedures:

```
PACKAGE emp_actions IS  -- package specification
    PROCEDURE hire_employee ( empno NUMBER, ename CHAR, . . . ) ;

    PROCEDURE fire_employee ( emp_id NUMBER) ;
END emp_actions;

PACKAGE BODY emp_actions IS  -- package body
    PROCEDURE hire_employee ( empno NUMBER, ename CHAR, . . . ) IS
    BEGIN
        INSERT INTO emp VALUES ( empno, ename, . . . ) ;
    END hire_employee;

    PROCEDURE fire_employee ( emp_id NUMBER ) IS
    BEGIN
        DELETE FROM emp WHERE empno = emp_id;
    END fire_employee;
END emp_actions;
```

Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible.

PL/SQL Tables

PL/SQL provides a composite datatype named TABLE. Objects of type TABLE are called PL/SQL *tables*, which are modelled as (but not the same as) database tables. PL/SQL tables have only one column and use a primary key to give you array-like access to rows. The column can belong to any scalar type (such as CHAR, DATE, or NUMBER), but the primary key must belong to type BINARY_INTEGER.

You can declare PL/SQL table types in the declarative part of any block, procedure, function, or package. In the following example, you declare a TABLE type called *NumTabTyp:*

```
...
EXEC SQL EXECUTE
    DECLARE
        TYPE NumTabTyp IS TABLE OF NUMBER
            INDEX BY BINARY_INTEGER;
        ...
    BEGIN
        ...
    END;
END-EXEC;
...
```

Once you define type *NumTabTyp,* you can declare PL/SQL tables of that type, as the next example shows:

```
num_tab NumTabTyp;
```

The identifier *num_tab* represents an entire PL/SQL table.

You reference rows in a PL/SQL table using array-like syntax to specify the primary key value. For example, you reference the ninth row in the PL/SQL table named *num_tab* as follows:

```
num_tab(9) . . .
```

User-defined Records

You can use the %ROWTYPE attribute to declare a record that represents a row in a table or a row fetched by a cursor. However, you cannot specify the datatypes of fields in the record or define fields of your own. The composite datatype RECORD lifts those restrictions.

Objects of type RECORD are called *records*. Unlike PL/SQL tables, records have uniquely named fields, which can belong to different datatypes. For example, suppose you have different kinds of data about an employee such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such fields as the name, salary, and hire date of an employee would let you treat the data as a logical unit.

You can declare record types and objects in the declarative part of any block, procedure, function, or package. In the following example, you declare a RECORD type called *DeptRecTyp:*

```
DECLARE
    TYPE  DeptRecTyp  IS  RECORD
         (deptno  NUMBER(4)  NOT  NULL,
          dname    CHAR(9),
          1oc      CHAR(14));
```

Notice that the field declarations are like variable declarations. Each field has a unique name and specific datatype. You can add the NOT NULL option to any field declaration and so prevent the assigning of nulls to that field.

Once you define type *DeptRecTyp*, you can declare records of that type, as the next example shows

```
dept_rec   DeptRecTyp;
```

The identifier *dept_rec* represents an entire record.

You use dot notation to reference individual fields in a record. For example, you reference the *dname* field in the *dept_rec* record as follows:

```
dept_rec. dname . . .
```

Embedding PL/SQL Blocks

The ORACLE Precompilers treat a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in a host program that you can place a SQL statement.

To embed a PL/SQL block in your host program, simply bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END-EXEC as follows:

```
EXEC SQL EXECUTE
    DECLARE
        ...
    BEGIN
        ...
    END;
END-EXEC;
```

The keyword END-EXEC must be followed by the statement terminator for your host language.

After writing your program, you precompiled the source file in the usual way. You can have the precompiled check the syntax and semantics of embedded PL/SQL blocks by specifying the SQLCHECK option. For more information, see the section “Using the Precompiled Options” in Chapter 11.

Using Host Variables

Host variables are the key to communication between a host language and a PL/SQL block. Host variables can be shared with PL/SQL, meaning that PL/SQL can set and reference host variables.

For example, you can prompt a user for information and use host variables to pass that information to a PL/SQL block. Then, PL/SQL can access the database and use host variables to pass the results back to your host program.

Inside a PL/SQL block, host variables are treated as global to the entire block and can be used anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.

An Example

The following example illustrates the use of host variables with PL/SQL. The program prompts the user for an employee number, then displays the job title, hire date, and salary of that employee.

```
EXEC SQL BEGIN DECLARE SECTION;
    username     CHARACTER (20);
    password     CHARACTER (20);
    emp_number   INTEGER;
    job_title    CHARACTER (20);
    hire_date    CHARACTER (9);
    salary       REAL ;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

display 'Username? ' ;
read username;
display ' Password? ' ;
read password;

EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL CONNECT : username IDENTIFIED BY : password;
display ' Connected to ORACLE' ;
```

```
LOOP
    display 'Employee Number (0 to end)? ';
    read emp_number;

    IF emp_number = 0 THEN
        EXEC SQL COMMIT WORK RELEASE;
        display 'Exiting program';
        exit program;
    ENDIF;

    ---------------- begin PL/SQL block -----------------
    EXEC SQL EXECUTE
        BEGIN
            SELECT job, hiredate, sal
                INTO :job_title, :hire_date, :salary
                FROM emp
                WHERE empno = :emp_number;
        END;
    END-EXEC;
    ---------------- end PL/SQL block -----------------

    display 'Number  Job Title Hire Date  Salary' ;
    display ' ---------------------------------- ' ;
    display emp_number, job_title, hire_date, salary;
ENDLOOP;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    display 'Processing error';
    exit program with an error;
```

Notice that the host variable *emp_number* is set before the PL/SQL block is entered, and the host variables *job_title*, *hire_date*, and *salary* are set inside the block.

A More Complex Example

In the example below, you prompt the user for a bank account number, transaction type, and transaction amount, then debit or credit the account. If the account does not exist, you raise an exception. When the transaction is complete, you display its status.

```
EXEC SQL BEGIN DECLARE SECTION;
    username    CHARACTER (20);
    password    CHARACTER (20);
    acct_num    INTEGER;
    trans_type  CHARACTER (1);
    trans_amt   REAL ;
    status      CHARACTER (80);
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

display  ' Username? ' ;
read username;
display  ' Password? ' ;
read password;

EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL CONNECT : username IDENTIFIED BY :password;
display  ' Connected to ORACLE' ;

LOOP
    display  'Account Number (0 to end) ? ' ;
    read acct_num;

    IF acct_num = 0 THEN
        EXEC SQL COMMIT WORK RELEASE;
        display  ' Exiting program' ;
        exit program;
    ENDIF;

    display 'Transaction Type - D)ebit or C)redit?'
    read trans_type;

    display 'Transaction  Amount?
    read trans_amt;
```

```
    ---------------------- begin PL/SQL block -------- - ----------
    EXEC SQL EXECUTE
        DECLARE
            old_bal       NUMBER(9,2);
            err_msg       CHAR(70);
            nonexistent   EXCEPTION;
        BEGIN
            :trans_type := UPPER(:trans_type);
            IF :trans_type = 'C' THEN       -- credit the account
                UPDATE accts SET bal = bal + :trans_amt
                    WHERE acctid = :acct_num;
                IF SQL%ROWCOUNT = 0 THEN    -- no rows affected
                    RAISE nonexistent;
                ELSE
                    :status := 'credit applied';
                END IF;
            ELSIF :trans_type = 'D' THEN    -- debit the account
                SELECT bal INTO old_bal FROM accts
                    WHERE acctid = :acct_num;
                IF old_bal >= :trans_amt THEN   -- enough funds
                    UPDATE accts SET bal = bal - :trans_amt
                        WHERE acctid = :acct_num;
                    :status := 'Debit applied' ;
                ELSE
                    :status := 'Insufficient funds';
                END IF;
            ELSE
                : status := 'Invalid type: ' ||:trans_type;
            END IF;
            COMMIT;
        EXCEPTION
            WHEN NO_DATA_FOUND OR nonexistent THEN
                :status := 'Nonexistent account';
            -  OTHERS  THEN
                err_msg := SUBSTR(SQLERRM, 1, 70);
                :status := 'Error: '  I I err_msg;
        END;
    END-EXEC;
    ------------------- end PL/SQL block -----------------------

    display 'Status: ', status;
ENDLOOP;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    display 'Processing error';
    exit program with an error;
```

VARCHAR Pseudotype

Recall from Chapter 3 that you can use the VARCHAR pseudotype to declare variable-length character strings. If the VARCHAR is an input host variable, you must tell ORACLE what length to expect. So, set the length field to the actual length of the value stored in the string field.

If the VARCHAR is an output host variable, ORACLE automatically sets the length field. However, to use a VARCHAR output host variable in your PL/SQL block, you must initialize the length field *before* entering the block. So, set the length field to the declared (maximum) length of the VARCHAR, as shown in the following example

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_number   INTEGER;
    emp_name     VARCHAR(10) ;
    salary       REAL ;
    ...
EXEC SQL END DECLARE SECTION;
    ...
    set emp_name.len = 10;   -- initialize length field

    EXEC SQL EXECUTE
        BEGIN
            SELECT ename, sal INTO : emp_name, : salary
                FROM emp
                WHERE empno = : emp_number;
            ...
        END;
    END-EXEC;
    ...
```

Using Indicator Variables

PL/SQL does not need indicator variables because it can manipulate nulls. For example, within PL/SQL, you can use the IS NULL operator to test for nulls, as follows:

```
IF variable IS NULL THEN . . .
```

And, you can use the assignment operator (:=) to assign nulls, as follows:

```
variable : = NULL;
```

However, host languages need indicator variables because they cannot manipulate nulls. Embedded PL/SQL meets this need by letting you use indicator variables to

- accept nulls input from a host program
- output nulls or truncated values to a host program

When used in a PL/SQL block, indicator variables are subject to the following rules:

- You cannot refer to an indicator variable by itself; it must be appended to its associated host variable.
- If you refer to a host variable with its indicator variable, you must always refer to it that way in the same block.

In the following example, the indicator variable *ind_comm* appears with its host variable commission in the SELECT statement, so it must appear that way in the IF statement

```
...
EXEC SQL EXECUTE
    BEGIN
        SELECT ename, comm
            INTO : emp_name,  : commission: ind_comm FROM emp
            WHERE empno = : emp_number;
        IF : commission: ind_comm IS NULL THEN . . .
        ...
    END;
END-EXEC;
```

Notice that PL/SQL treats *:commission:ind_comm* like any other simple variable. Though you cannot refer directly to an indicator variable inside a PL/SQL block, PL/SQL checks the value of the indicator variable when entering the block and sets the value correctly when exiting the block.

Handling Nulls

When entering a block, if an indicator variable has a value of -1, PL/SQL automatically assigns a null to the host variable. When exiting the block, if a host variable is null, PL/SQL automatically assigns a value of-1 to the indicator variable. In the next example, if *ind_sal* had a value of-1 before the PL/SQL block was entered, the *salary_missing* exception is raised. An exception is a named error condition.

```
...
EXEC SQL EXECUTE
    BEGIN
        IF :salary: ind_sal IS NULL THEN
            RAISE  salary_missing;
        END IF;
        ...
    END;
END-EXEC;
...
```

Handling Truncated Values

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string. In the following example, the host program will be able to tell, by checking the value of *ind_name*, if a truncated value was assigned to *emp_name:*

```
...
EXEC SQL EXECUTE
    DECLARE
        ...
        new_name   CHAR(10);
    BEGIN
        ...
        : emp_name: ind_name : = new_name;
        ...
    END;
END-EXEC;
```

Using Host Arrays

You can pass input host arrays and indicator arrays to a PL/SQL block. They can be indexed by a PL/SQL variable of type BINARY_INTEGER or by a host variable compatible with that type. Normally, the entire host array is passed to PL/SQL, but you can use the ARRAYLEN statement (discussed later) to specify a smaller array dimension.

Furthermore, you can use a procedure call to assign all the values in a host array to rows in a PL/SQL table. Given that the array subscript range is *m*...*n*, the corresponding PL/SQL table index range is always 1.. *n* - *m* + 1. For example, if the array subscript range is 5..10, the corresponding PL/SQL table index range is 1.. (10 - 5 + 1) or 1..6.

In the example below, you pass a host array named *salary* to a PL/SQL block, which uses the host array in a function call. The function is named *median* because it finds the middle value in a series of numbers. Its formal parameters include a PL/SQL table named *num_tab*. The function call assigns all the values in the actual parameter *salary* to rows in the formal parameter *num_tab*.

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    salary (100)  REAL;
EXEC SQL END DECLARE SECTION;

-- populate the host array

EXEC SQL EXECUTE
    DECLARE
        TYPE NumTabTyp IS TABLE OF REAL
            INDEX BY BINARY_INTEGER ;
        median_salary  REAL;
        n  BINARY_INTEGER;
        ...
        FUNCTION median ( num_tab NumTabTyp, n INTEGER)
          RETURN REAL IS
        BEGIN
            -- compute median
        END;
    BEGIN
        n := 100;
        median_salary  : = median(:salary, n) ;
        ...
    END;
END-EXEC;
...
```

You can also use a procedure call to assign all row values in a PL/SQL table to corresponding elements in a host array. For an example, see the section "Stored Procedures" later in this chapter.

Table 5-1 shows the legal conversions between row values in a PL/SQL table and elements in a host array. For example, a host array of type LONG is compatible with a PL/SQL table of type VARCHAR2, LONG, RAW, or LONG RAW. Notably, it is not compatible with a PL/SQL table of type CHAR.

Table 5-1
Legal Datatype Conversions

PL/SQL Table \ Host Array	VARCHAR2	NUMBER	LONG	ROWID	DATE	RAW	LONG RAW	CHAR
VARCHAR2	↵		↵			↵	↵	
NUMBER		↵						
LONG	↵		↵			↵	↵	
ROWID				↵				
DATE					↵			
RAW	↵		↵			↵	↵	
LONG RAW	↵		↵			↵	↵	
CHAR								↵

The ORACLE Precompilers do not check your usage of host arrays. For instance, no index range-checking is done.

ARRAYLEN Statement

Suppose you must pass an input host array to a PL/SQL block for processing. By default, when binding such a host array, the ORACLE Precompilers use its declared dimension. However, you might not want to process the entire array. In that case, you can use the ARRAYLEN statement to specify a smaller array dimension. ARRAYLEN associates the host array with a host variable, which stores the smaller dimension. The statement syntax is

```
EXEC SQL ARRAYLEN host_array ( dimension ) ;
```

where *dimension* is a 4-byte, integer host variable, *not* a literal or expression.

The ARRAYLEN statement must appear in the Declare Section along with, but somewhere after, the declarations of *host_array* and *dimension*. You cannot specify an offset into the host array. However, you might be able to use host-language features for that purpose. In the following example, you use ARRAYLEN to override the default dimension of a host array named *bonus:*

```
EXEC SQL BEGIN DECLARE SECTION;
    bonus (100)   REAL ;
    my_dim        INTEGER ;
    EXEC SQL ARRAYLEN bonus (my_dim) ;
EXEC SQL END DECLARE SECTION;
-- populate the host array
...
set my_dim = 25; -- set smaller array dimension
EXEC SQL EXECUTE
    DECLARE
        TYPE NumTabTyp IS TABLE OF REAL
            INDEX BY BINARY_INTEGER;
        median_bonus  REAL;
        FUNCTION median (num_tab NumTabTyp, n INTEGER)
          RETURN REAL IS
        BEGIN
            -- compute median
        END;
    BEGIN
        median_bonus : = median (: bonus, :my_dim;
        ...
    END;
END-EXEC;
```

Only 25 array elements are passed to the PL/SQL block because ARRAYLEN downsizes the host array from 100 to 25 elements. As a result, when the PL/SQL block is sent to ORACLE for execution, a much smaller host array is sent along. This saves time and, in a networked environment, reduces network traffic.

Using Cursors

Every embedded SQL statement is assigned a cursor, either explicitly by you in a DECLARE CURSOR statement or implicitly by the precompiled. Internally, the ORACLE Precompilers maintain a cache, called the *cursor cache,* to control the execution of embedded SQL statements. When executed, every SQL statement is assigned an entry in the cursor cache. This entry is linked to a private SQL area in your Program Global Area (PGA) within ORACLE.

Various precompiled options, including MAXOPENCURSORS, HOLD_CURSOR, and RELEASE_CURSOR, let you manage the cursor cache to improve performance. For example, RELEASE_CURSOR controls what happens to the link between the cursor cache and private SQL area. If you specify RELEASE_CURSOR=YES, the link is removed after ORACLE executes the SQL statement. This frees memory allocated to the private SQL area and releases parse locks.

For purposes of cursor cache management, an embedded PL/SQL block is treated just like a SQL statement. At run time, a cursor, called a *parent cursor*, is associated with the entire PL/SQL block. A corresponding entry is made to the cursor cache, and this entry is linked to a private SQL area in the PGA.

Each SQL statement inside the PL/SQL block also requires a private SQL area in the PGA. So, PL/SQL manages a separate cache, called the *child cursor cache*, for these SQL statements. Their cursors are called *child cursors.* Because PL/SQL manages the child cursor cache, you do not have direct control over child cursors.

The maximum number of cursors your program can use simultaneously is set by the ORACLE initialization parameter OPEN_CURSORS. Figure 5-1 shows you how to calculate the maximum number of cursors in use:

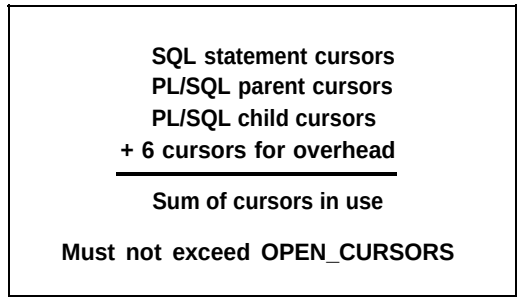


Figure 5-1
Maximum Cursors in Use

If your program exceeds the limit imposed by OPEN_CURSORS, you get the following ORACLE error:

```
ORA-01000: maximum open cursors exceeded
```

You can avoid this error by specifying the RELEASE_CURSOR=YES option. If you do not want to precompiled the entire program with RELEASE_CURSOR set to YES, simply reset it to NO after each PL/SQL block, as follows:

```
EXEC ORACLE OPTION ( RELEASE_CURSOR=YES ) ;
     -- first embedded PL/SQL block
EXEC ORACLE OPTION ( RELEASE_CURSOR=NO ) ;
     -- embedded SQL statements
EXEC ORACLE OPTION ( RELEASE_CURSOR=YES ) ;
     -- second embedded PL/SQL block
EXEC ORACLE OPTION ( RELEASE_CURSOR=NO ) ;
     -- embedded SQL statements
...
```

An Alternative

The MAXOPENCURSORS option specifies the initial size of the cursor cache. For example, when MAXOPENCURSORS=10, the cursor cache can hold up to 10 entries. If a new cursor is needed and there are no free cache entries, the precompiled tries to reuse an entry. If you specify a very low value for MAXOPENCURSORS, the precompiled is forced to reuse the parent cursor more often. All the child cursors are released as soon as the parent cursor is reused.

Stored Subprograms

Unlike anonymous blocks, PL/SQL subprograms (procedures and functions) can be compiled separately, stored in an ORACLE database, and invoked. A subprogram explicitly CREATEd using an ORACLE tool such as SQL*Plus or SQL*DBA is called a *stored* subprogram. Once compiled and stored in the data dictionary, it is a database object, which can be reexecuted without being recompiled.

When a subprogram within a PL/SQL block or stored procedure is sent to ORACLE by your application, it is called an *inline* subprogram. ORACLE compiles the inline subprogram and caches it in the System Global Area (SGA), but does not store the source or object code in the data dictionary.

Subprograms defined within a package are considered part of the package, and so are called *packaged* subprograms. Stored subprograms not defined within a package are called *stand-alone* subprograms.

Creating Stored Subprograms

You can embed the SQL statements CREATE FUNCTION, CREATE PROCEDURE, and CREATE PACKAGE in a host program, as the following example shows:

```
EXEC SQL CREATE
    FUNCTION sal_ok (salary REAL, title CHAR)
      RETURN BOOLEAN AS
        min_sal  REAL ;
        max_sal  REAL ;
    BEGIN
        SELECT losal, hisal INTO min_sal, max_sal
            FROM sals
            WHERE job = title;
        RETURN (salary >= min_sal) AND
               (salary <= max_sal);
    END sal_ok;
END-EXEC ;
```

Notice that the embedded CREATE {FUNCTION | PROCEDURE I PACKAGE} statement is a hybrid. Like all other embedded CREATE statements, it begins with the keywords EXEC SQL (not EXEC SQL EXECUTE). But, unlike other embedded CREATE statements, it ends with the PL/SQL terminator END-EXEC.

In the example below, you create a package that contains a procedure named *get_employees*, which fetches a batch of rows from the *emp* table. The batch size is determined by the caller of the procedure, which might be another stored subprogram or a client application program.

The procedure declares three PL/SQL tables as OUT formal parameters, then fetches a batch of employee data into the PL/SQL tables. The matching actual parameters are host arrays. When the procedure finishes, it automatically assigns all row values in the PL/SQL tables to corresponding elements in the host arrays.

```
EXEC SQL CREATE OR REPLACE PACKAGE emp_actions AS
        TYPE CharArrayTyp IS TABLE OF VARCHAR2 (10)
            INDEX BY BINARY_ INTEGER ;
        TYPE NumArrayTyp IS TABLE OF FLOAT
            INDEX BY BINARY_INTEGER ;
```

```
        PROCEDURE  get_employees(
            dept_number  IN     INTEGER,
            batch_size  IN     INTEGER,
            found        IN OUT INTEGER,
            done_fetch  OUT    INTEGER,
            emp_name     OUT    CharArrayTyp,
            job-title    OUT    CharArrayTyp,
            salary       OUT    NumArrayTyp)  ;

    END emp_actions;
END-EXEC;

EXEC SQL CREATE OR REPLACE PACKAGE BODY emp_actions AS

        CURSOR get_emp (dept_number IN INTEGER) IS
            SELECT ename,  job, sal FROM emp
                WHERE deptno = dept_number;

        PROCEDURE  get_employees(
            dept_number  IN     INTEGER,
            batch_size  IN     INTEGER,
            found        IN OUT INTEGER,
            done_fetch  OUT    INTEGER,
            emp_name     OUT    CharArrayTyp,
            job_title    OUT    CharArrayTyp,
            salary       OUT    NumArrayTyp)  IS

        BEGIN
            IF  NOT  get_emp%ISOPEN  THEN
                OPEN  get_emp  (dept_number);
            END IF;
            done_fetch := 0;
            found := 0;
            FOR i IN 1..batch_size LOOP
                FETCH  get_emp  INTO  emp_name(i)  ,
                         job_title(i),  salary(i);
                IF  get_emp%NOTFOUND  THEN
                    CLOSE get_emp;
                    done_fetch := 1;
                    EXIT ;
                ELSE
                    found  := found + 1;
                END IF;
            END  LOOP;
        END  get_employees;
    END  emp_actions;
END- EXEC ;
```

You specify the REPLACE clause in the CREATE statement to redefine an existing package without having to drop the package, recreate it, and regrant privileges on it. For the full syntax of the CREATE statement see the *ORACLE7 Server SQL Language Reference Manual.*

If an embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement fails, ORACLE generates a warning, not an error.

Calling a Stored Subprogram

To invoke (call) a stored subprogram from your host program, you must use an anonymous PL/SQL block. In the following example, you call a stand-alone procedure named *raise_salary:*

```
EXEC SQL EXECUTE
    BEGIN
        raise_ salary ( :emp_id,  : increase) ;
    END ;
END- EXEC;
```

Notice that stored subprograms can take parameters. In this example, the actual parameters *emp_id* and *increase* are host variables.

In the next example, the procedure *raise_salary* is stored in a package named *emp_actions,* so you must use dot notation to fully qualify the procedure call:

```
EXEC SQL EXECUTE
    BEGIN
        emp_actions. raise_salary ( : emp_id, : increase) ;
    END;
END-EXEC;
```

An actual IN parameter can be a literal, host variable, host array, PL/SQL constant or variable, PL/SQL table, PL/SQL user-defined record, procedure call, or expression. However, an actual OUT parameter cannot be a literal, procedure call, or expression.

In the Pro*C example below, three of the formal parameters are PL/SQL tables, and the corresponding actual parameters are host arrays. The program calls the stored procedure *get_employees* repeatedly, displaying each batch of employee data, until no more data is found.

```
#include  <stdio.h>
#include  <string.h>

typedef char asciz;

EXEC SQL BEGIN DECLARE SECTION;
    /* Define type for null-terminated strings */
    EXEC SQL TYPE asciz IS STRING;
    asciz   username [20] ;
    asciz   password [20] ;
    int     dept_no;     /* which department to query */
    char    emp_name[10] [21];
    char    job[10] [21];
    float   salary[10] ;
    int     done_flag;
    int     array_size;
    int     num_ret;     /* number of rows returned */
    int     SQLCODE;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;

int print_rows() ;  /* produces program output      */
int sqlerror() ;    /* handles unrecoverable errors */

main( )
{
    int i;

    /* Connect to ORACLE. */
    strcpy(username,  "SCOTT");
    strcpy(password,  "TIGER");

    EXEC SQL WHENEVER SQLERROR DO sqlerror() ;

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user: %s\n", username);

    printf("enter department number: ");
    scanf("%d", &dept_no) ;
    fflush(stdin)  ;
```

```
    /* Set the array size. */
    array_size = 10;

    done_flag = 0;
    num_ret = 0;

    /* Array fetch loop - ends when NOT FOUND is true. */
    for (;;)
    {
        EXEC SQL EXECUTE
            BEGIN  emp_actions.get_employees
                (:dept_no,  :array_size,  :num_ret,
                 :done_flag, :emp_name, :job, :salary);
            END;
        END-EXEC;
        print_rows(num_ret);
        if (done_flag)
            break;
    }

    /* Disconnect from the database. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

print_rows(n)
int n;
{
    int i;

    if (n == 0)
    {
        printf("No rows retrieved.\n");
        return;
    }

    printf("\n\nGot %d row%c\n", n, n == 1 ? '\0' : 's');
    printf( "%-20.20s%-20.20s%s\n", "Ename", "Job", 'Salary');
    for (i = 0; i < n; i++)
        printf("%20.20s%20.20s%6.2f\n",
               emp_name[i], job[i], salav[i]);
}

sqlerroro
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\nORACLE error detected:");
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc) ;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(l);
```

Remember, the datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Also, before a stored procedure is exited, all OUT formal parameters must be assigned values. Otherwise, the values of corresponding actual parameters are indeterminate.

Remote Access

PL/SQL lets you access remote databases via *database links.* Typically, database links are established by your DBA and stored in the ORACLE data dictionary. A database link tells ORACLE where the remote database is located, the path to it, and what ORACLE username and password to use. In the following example, you use the database link *dallas* to call the *raise_salary* procedure:

```
EXEC SQL EXECUTE
    BEGIN
        raise_ salary@dallas ( :emp_id, : increase) ;
    END ;
END-EXEC ;
```

You can create synonyms to provide location transparency for remote subprograms, as the following example shows:

```
CREATE PUBLIC SYNONYM raise_salary
    FOR  raise_salary@dallas;
```

Getting Information about Stored Subprograms

In Chapter 3, you learned how to embed OCI calls in your host program. After calling the library routine SQLLDA to set up the LDA, you can use the OCI call ODESSP to get useful information about a stored subprogram. When you call ODESSP, you must pass it a valid LDA and the name of the subprogram. For packaged subprograms, you must also pass the name of the package. ODESSP returns information about each subprogram parameter such as its datatype, size, position, and so on. For details, see the *Programmer's Guide to the ORACLE Call Interfaces.*

Using Dynamic SQL

Recall that the ORACLE Precompilers treat an entire PL/SQL block like a single SQL statement. Therefore, you can store a PL/SQL block in a string host variable. Then, if the block contains no host variables, you can use dynamic SQL Method 1 to EXECUTE the PL/SQL string. Or, if the block contains a known number of host variables, you can use dynamic SQL Method 2 to PREPARE and EXECUTE the PL/SQL string. If the block contains an unknown number of host variables, you must use dynamic SQL Method 4. For more information, refer to Chapter 9, "Using Dynamic SQL."

CHAPTER

6

DEFINING AND CONTROLLING TRANSACTIONS

This chapter explains how to do transaction processing. You learn the basic techniques that safeguard the consistency of your database, including how to control whether changes to ORACLE data are made permanent or undone. The following topics are discussed:

- how transactions guard your database
- how transactions begin and end
- making transactions permanent
- undoing transactions
- setting read-only transactions
- overriding default locking
- fetching across COMMIT'S
- handling distributed transactions
- guidelines

Some Terms You Should Know

Before delving into the subject of transactions, you should know the terms defined in this section.

The jobs or tasks that ORACLE manages are called *sessions.* A *user session* is invoked when you run an application program or a tool such as SQL*Forms.

ORACLE allows user sessions to work "simultaneously" and share computer resources. To do this, ORACLE must control *concurrency, the* accessing of the same data by many users. Without adequate " concurrency controls, there might be a loss of data *integrity*. That is, changes to data or structures might be made in the wrong order.

ORACLE uses *locks* (sometimes called *enqueues)* to control concurrent access to data. A lock gives you temporary ownership of a database resource such as a table or row of data. Thus, data cannot be changed by other users until you finish with it.

You need never explicitly lock a resource, because default locking mechanisms protect ORACLE data and structures. However, you can request *data locks* on tables or rows when it is to your advantage to override default locking. You can choose from several *modes* of locking such as row *share* and *exclusive.*

A *deadlock can* occur when two or more users try to access the same database object. For example, two users updating the same table might wait if each tries to update a row currently locked by the other. Because each user is waiting for resources held by another user, neither can continue until ORACLE breaks the deadlock. ORACLE signals an error to the participating transaction that had completed the least amount of work, and the "deadlock detected while waiting for resource" ORACLE error code is returned to SQLCODE in the SQLCA.

When a table is being queried by one user and updated by another at the same time, ORACLE generates a *read-consistent* view of the table's data for the query. That is, once a query begins and as it proceeds, the data read by the query does not change. As update activity continues, ORACLE takes *snapshots* of the table's data and records changes in a *rollback segment.* ORACLE uses information in the rollback segment to build read-consistent query results and to undo changes if necessary.

How Transactions Guard Your Database

ORACLE is transaction oriented; that is, it uses transactions to ensure data integrity. A transaction is a series of one or more logically related SQL statements you define to accomplish some task. ORACLE treats the series of SQL statements as a unit so that all the changes brought about by the statements are either *committed* (made permanent) or *rolled back* (undone) at the same time. If your application program fails in the middle of a transaction, the database is automatically restored to its former (pre-transaction) state.

The coming sections show you how to define and control transactions. Specifically, you learn how to

- begin and end transactions
- use the COMMIT statement to make transactions permanent
- use the SAVEPOINT statement with the ROLLBACK TO statement to undo parts of transactions
- use the ROLLBACK statement to undo whole transactions
- specify the RELEASE option to free resources and log off the database
- use the SET TRANSACTION statement to set read-only transactions
- use the FOR UPDATE clause or LOCK TABLE statement to override default locking

For details about the SQL statements discussed in this chapter, see the ORACLE7 Server *SQL Language Reference Manual.*

How to Begin and End Transactions

You begin a transaction with the first executable SQL statement (other than CONNECT) in your program. When one transaction ends, the next executable SQL statement automatically begins another transaction. Thus, every executable statement is part of a transaction. Because they cannot be rolled back and need not be committed, declarative SQL statements are not considered part of a transaction.

You end a transaction in one of the following ways:

- Code a COMMIT or ROLLBACK statement, with or without the RELEASE option. This *explicitly* makes permanent or undoes changes to the database.
- Code a data definition statement (ALTER, CREATE, or GRANT, for example), which issues an automatic COMMIT before *and* after executing. This *implicitly makes* permanent changes to the database.

A transaction also ends when there is a system failure or your user session stops unexpectedly because of software problems, hardware problems, or a forced interrupt. ORACLE rolls back the transaction.

If your program fails in the middle of a transaction, ORACLE detects the error and rolls back the transaction. If your operating system fails, ORACLE restores the database to its former (pre-transaction) state.

Using the COMMIT Statement

If you do not subdivide your program with the COMMIT or ROLLBACK statement, ORACLE treats the whole program as a single transaction (unless the program contains data definition statements, which issue automatic COMMITS).

You use the COMMIT statement to make changes to the database permanent. Until changes are COMMITted, other users cannot access the changed data; they see it as it was before your transaction began. Specifically, the COMMIT statement

- makes permanent all changes made to the database during the current transaction
- makes these changes visible to other users
- erases all savepoints (see the next section)
- releases all row and table locks, but not parse locks
- closes cursors referenced in a CURRENT OF clause or, when MODE= {ANSI | ANSI14}, closes all explicit cursors
- ends the transaction

The COMMIT statement has no effect on the values of host variables or on the flow of control in your program.

When MODE= {ANSI13 | ORACLE}, explicit cursors not referenced in a CURRENT OF clause remain open across COMMITS. This can boost performance. For an example, see the section “Fetching Across COMMITs” later in this chapter.

Because they are part of normal processing, COMMIT statements should be placed inline, on the main path through your program. Before your program terminates, it must explicitly COMMIT pending changes. Otherwise, ORACLE rolls them back. In the following example, you commit your transaction and disconnect from ORACLE:

```
EXEC SQL COMMIT WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all ORACLE resources (locks and cursors) held by your program and logs off the database.

You need not follow a data definition statement with a COMMIT statement because data definition statements issue an automatic COMMIT before ***and*** after executing. So, whether they succeed or fail, the prior transaction is committed.

Using the SAVEPOINT Statement

You use the SAVEPOINT statement to mark and name the current point in the processing of a transaction. Each marked point is called a *savepoint*. For example, the following statement marks a savepoint named *start_deletc*

```
EXEC SQL SAVEPOINT start_delete:
```

The savepoint name is an identifier used by the precompiled, *not* a host or program variable, and should not be defined in the Declare Section.

Savepoints let you divide long transactions, giving you more control over complex procedures. For example, if a transaction performs several functions, you can mark a savepoint before each function. Then, if a function fails, you can easily restore the ORACLE data to its former state, recover, then reexecute the function.

To undo part of a transaction, you use savepoints with the ROLLBACK statement and its TO SAVEPOINT clause. In the following example, you access the table MAIL_LIST to insert new listings, update old listings, and delete (a few) inactive listings. After the delete, you check the third element of SQLERRD in the SQLCA for the number of rows deleted. If the number is unexpectedly large, you rollback to the savepoint *start_delete*, undoing just the delete.

```
...
FOR EACH new customer
    display  ‘ Customer number? ‘ ;
    read  cust_number;
    display  ‘ Customer name? ‘ ;
    read  cust_name;
    EXEC SQL INSERT INTO MAIL_LIST (CUSTNO, CNAME, STAT)
        VALUES ( :cust_number, : cust_name , 'ACTIVE') ;
ENDFOR;

 FOR EACH revised status
    display ‘ Customer number? ‘ ;
    read  cust_number;
    display  ‘New status? ‘ ;
    read  new_status;
    EXEC SQL UPDATE MAIL_LIST
        SET STAT =  :new_status
        WHERE CUSTNO = : cust_number;
ENDFOR;
```

```
-- mark savepoint
EXEC SQL SAVEPOINT start_delete;

EXEC SQL DELETE FROM MAIL_LIST
    WHERE STAT = 'INACTIVE' ;

IF sqlca.sqlerrd (3) < 25 THEN -- check number of rows deleted
    display  'Number of rows deleted is ' , sqlca.sqlerrd(3) ;
ELSE
    display 'Undoing deletion of ' , sqlca.sqlerrd(3), ' rows' ;
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;
    EXEC SQL ROLLBACK TO SAVEPOINT start_delete ;
ENDIF;

EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL COMMIT WORK RELEASE;
exit program;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE ;
    display  'Processing error' ;
    exit program with an error;
```

Rolling back to a savepoint erases any savepoints marked after that savepoint. The savepoint to which you rollback, however, is not erased. For example, if you mark five savepoints, then rollback to the third, only the fourth and fifth are erased.

If you give two savepoints the same name, the earlier savepoint is erased. A COMMIT or ROLLBACK statement erases all savepoints.

By default, the number of active savepoints peruser session is limited to 5. An *active* savepoint is one that you marked since the last commit or rollback. Your Database Administrator (DBA) can raise the limit (up to 255) by increasing the value of the ORACLE initialization parameter SAVEPOINTS.

Using the ROLLBACK Statement

You use the ROLLBACK statement to undo pending changes made to the database. For example, if you make a mistake, such as deleting the wrong row from a table, you can use ROLLBACK to restore the original data. The TO SAVEPOINT clause lets you rollback to an intermediate statement in the current transaction, so you do not have to undo all your changes.

If you start a transaction that you cannot finish (a SQL statement might not execute successfully, for example), ROLLBACK lets you return to the starting point, so that the database is not left in an inconsistent state. Specifically, the ROLLBACK statement

- undoes all changes made to the database during the current transaction
- erases all savepoints
- ends the transaction
- releases all row and table locks, but not parse locks
- closes cursors referenced in a CURRENT OF clause or, when MODE= {ANSI | ANSI14}, closes *all* explicit cursors

The ROLLBACK statement has no effect on the values of host variables or on the flow of control in your program.

When MODE= {ANSI13 | ORACLE}, explicit cursors not referenced in a CURRENT OF clause remain open across ROLLBACKS.

Specifically, the ROLLBACK TO SAVEPOINT statement

- undoes changes made to the database since the specified savepoint was marked
- erases all savepoints marked after the specified savepoint
- releases all row and table locks acquired since the specified savepoint was marked

Note that you cannot specify the RELEASE option in a ROLLBACK TO SAVEPOINT statement.

Because they are part of exception processing, ROLLBACK statements should be placed in error handling routines, off the main path through your program. In the following example, you rollback your transaction and disconnect from ORACLE:

```
EXEC SQL ROLLBACK WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all resources held by your program and logs off the database.

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes a ROLLBACK statement, your program might enter an infinite loop if the ROLLBACK fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the ROLLBACK statement, as shown in the following example:

```
...
EXEC SQL WHENEVER SQLERROR GOTO sql_error;

FOR EACH new employee
    display ' Employee number? ' ;
    read emp_number;
    display ' Employee name? ' ;
    read emp_name;
    EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
        VALUES (: emp_number, : emp_name );
ENDFOR;
...
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    display ' Processing error' ;
    exit program with an error;
```

ORACLE automatically rolls back transactions if your program terminates abnormally. Refer to the section "Using the Release Parameter" later in this chapter.

Statement-Level Rollbacks

Before executing any SQL statement, ORACLE marks an implicit savepoint (not available to you). Then, if the statement fails, ORACLE automatically rolls it back and returns the applicable error code to SQLCODE in the SQLCA. For example, if an INSERT statement causes an error by trying to insert a duplicate value in a unique index, the statement is rolled back.

ORACLE can also roll back single SQL statements to break deadlocks. ORACLE signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Only work started by the failed SQL statement is lost; work done before that statement in the current transaction is saved. Thus, if a data definition statement fails, the automatic commit that precedes it is not undone.

Before executing a SQL statement, ORACLE must parse it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. Errors detected while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

Using the RELEASE Option

ORACLE automatically rolls back changes if your program terminates abnormally. Abnormal termination occurs when your program does not explicitly commit or rollback work and disconnect from ORACLE using the RELEASE option. Normal termination occurs when your program runs its course, closes open cursors, explicitly commits or rolls back work, disconnects from ORACLE, and returns control to the user.

Your program will exit gracefully if the last SQL statement it executes is either

```
EXEC SQL COMMIT RELEASE;
```

or

```
EXEC SQL ROLLBACK RELEASE;
```

Otherwise, locks and cursors acquired by your user session are held after program termination until ORACLE recognizes that the user session is no longer active. This might cause other users in a multiuser environment to wait longer than necessary for the locked resources.

Using the SET TRANSACTION Statement

You use the SET TRANSACTION statement to begin a read-only transaction. Because they allow "repeatable reads," read-only transactions are useful for running multiple queries against one or more tables while other users update the same tables. An example of the SET TRANSACTION statement follows:

```
EXEC SQL SET TRANSACTION READ ONLY;
```

The SET TRANSACTION statement must be the first SQL statement in a read-only transaction and can appear only once in a transaction. The READ ONLY parameter is required. Its use does not affect other transactions.

Only the SELECT, COMMIT, and ROLLBACK statements are allowed in a read-only transaction. For example, including an INSERT, DELETE, or SELECT FOR UPDATE OF statement causes an error.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multitable, multiquery, read-consistent view. Other users can continue to query or update data as usual.

A COMMIT, ROLLBACK, or data definition statement ends a read-only transaction. (Recall that data definition statements issue an implicit COMMIT.)

In the following example, as a store manager, you check sales activity for the day, the past week, and the past month by using a read-only transaction to generate a summary report. The report is unaffected by other users updating the database during the transaction.

```
EXEC SQL SET TRANSACTION READ ONLY;

EXEC SQL SELECT SUM (SALEAMT) INTO : daily FROM SALES
    WHERE SALEDATE = SYSDATE ;

EXEC SQL SELECT SUM (SALEAMT)  INTO : weekly FROM SALES
    WHERE SALEDATE > SYSDATE - 7;

EXEC SQL SELECT SUM (SALEAMT)  INTO : monthly FROM SALES
    WHERE SALEDATE > SYSDATE - 30;

EXEC SQL COMMIT WORK;
    -- simply ends the transaction since there are no changes
    -- to make permanent

-- format and print report
```

Overriding Default Locking

By default, ORACLE implicitly (automatically) locks many data structures for you. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking. Explicit locking lets you share or deny access to a table for the duration of a transaction or ensure multitable and multiquery read consistency.

With the SELECT FOR UPDATE OF statement, you can explicitly lock specific rows of a table to make sure they do not change before an UPDATE or DELETE is executed. However, ORACLE automatically obtains row-level locks at UPDATE or DELETE time. So, use the FOR UPDATE OF clause only if you want to lock the rows *before* the UPDATE or DELETE.

You can explicitly lock entire tables using the LOCK TABLE statement.

Using FOR UPDATE OF

When you DECLARE a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you use the FOR UPDATE OF clause to acquire exclusive row locks. SELECT FOR UPDATE OF identifies the rows that will be updated or deleted, then locks each row in the active set. This is useful, for example, when you want to base an update on the existing values in a row. You must make sure the row is not changed by another user before your update.

The FOR UPDATE OF clause is optional. For example, instead of coding

```
EXEC SQL DECLARE emp.cursor CURSOR FOR
    SELECT ENAME , JOB , SAL FROM EMP WHERE DEPTNO = 20
        FOR UPDATE OF SAL;
```

you can drop the FOR UPDATE OF clause and simply code

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , JOB , SAL FROM EMP WHERE DEPTNO = 20 ;
```

The CURRENT OF clause signals the precompiled to add a FOR UPDATE clause if necessary. You use the CURRENT OF clause to refer to the latest row FETCHed from a cursor. For an example, see the section "Using the CURRENT of Clause" in Chapter 4.

Restrictions

If you use the FOR UPDATE OF clause, you cannot reference multiple tables.

An explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. All rows are locked at the OPEN, not as they are FETCHed. Row locks are released when you COMMIT or ROLLBACK (except when you ROLLBACK to a savepoint). Therefore, you cannot FETCH from a FOR UPDATE cursor after a COMMIT.

Using LOCK TABLE

You use the LOCK TABLE statement to lock one or more tables in a specified lock mode. For example, the statement below locks the EMP table in *row share* mode. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use.

```
EXEC SQL LOCK TABLE EMP IN ROW SHARE MODE NOWAIT ;
```

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an *exclusive* lock. While one user has an exclusive lock on a table, no other users can INSERT, UPDATE, or DELETE rows in that table.

For more information about lock modes, see the *ORACLE7 Server Application Developer's Guide.*

The optional keyword NOWAIT tells ORACLE not to wait for a table if it has been locked by another user. Control is immediately returned to your program, so it can do other work before trying again to acquire the lock. (You can check SQLCODE in the SQLCA to see if the LOCK TABLE failed.) If you omit NOWAIT, ORACLE waits until the table is available; the wait has no set limit.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. So, a query never blocks another query or an update, and an update never blocks a query. Only if two different transactions try to update the same row will one transaction wait for the other to complete.

Table locks are released when your transaction issues a COMMIT or ROLLBACK.

Fetching Across COMMITS

If you want to intermix COMMITS and FETCHes, do not use the CURRENT OF clause. Instead, SELECT the ROWID of each row, then use that value to identify the current row during the update or delete. An example follows:

```
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , SAL , ROWID FROM EMP WHERE JOB = ' CLERK ' ;
...
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO . . .
LOOP
    EXEC SQL FETCH emp_cursor INTO : emp_name, : salary, : row_id;
    ...
    EXEC SQL UPDATE EMP SET SAL = : new_salary
        WHERE ROWID = : row_id;
    EXEC SQL COMMIT;
ENDLOOP;
```

Note, however, that the FETCHed rows are *not* locked. So, you might get inconsistent results if another user modifies a row after you read it but before you update or delete it.

Handling Distributed Transactions

A *distributed database* is a single logical database comprising multiple physical databases at different nodes. A *distributed statement* is any SQL statement that accesses a remote node using a database link. A *distributed transaction* includes at least one distributed statement that updates data at multiple nodes of a distributed database. If the update affects only one node, the transaction is non-distributed.

When you issue a COMMIT, changes to each database affected by the distributed transaction are made permanent. If instead you issue a ROLLBACK, all the changes are undone. However, if a network or machine fails during the commit or rollback, the state of the distributed transaction might be unknown or *in doubt*. In such cases, if you have FORCE TRANSACTION system privileges, you can manually commit or rollback the transaction at your local database by using the FORCE clause. The transaction must be identified by a quoted literal containing the transaction ID, which can be found in the data dictionary view DBA_2PC_PENDING. Some examples follow:

```
EXEC SQL COMMIT FORCE '22.31.83 ' ;
...
EXEC SQL ROLLBACK FORCE '25.33 .86 ' ;
```

FORCE commits or rolls back only the specified transaction and does not affect your current transaction. Note that you cannot manually roll back in-doubt transactions to a savepoint.

The COMMENT clause in the COMMIT statement lets you specify a comment to be associated with a distributed transaction. If ever the transaction is in doubt, ORACLE stores the text specified by COMMENT in the data dictionary view DBA_2PC_PENDING along with the transaction ID. The text must be a quoted literal< 50 characters in length. An example follows:

```
EXEC SQL COMMIT COMMENT ' In-doubt trans; notify Order Entry' ;
```

For more information about distributed transactions, see the *ORACLE7 Server Application Developer's Guide.*

Guidelines

The following guidelines will help you avoid some common problems.

Designing Applications

When designing your application, group logically related actions together in one transaction. A well-designed transaction includes all the steps necessary to accomplish a given task—no more and no less.

Data in the tables you reference must be left in a consistent state. So, the SQL statements in a transaction should change the data in a consistent way. For example, a transfer of funds between two bank accounts should include a debit to one account and a credit to another. Both updates should either succeed or fail together. An unrelated update, such as a new deposit to one account, should not be included in the transaction.

Obtaining Locks

If your application programs include SQL locking statements, make sure the ORACLE users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other users can lock tables they own or tables for which they have a privilege, such as ALTER, SELECT, INSERT, UPDATE, or DELETE.

Using PL/SQL

If a PL/SQL block is part of a transaction, COMMITS and ROLLBACKs inside the block affect the whole transaction. In the following example, the ROLLBACK undoes changes made by the UPDATE *and* the INSERT:

```
...
EXEC SQL INSERT INTO EMP . . .

EXEC  SQL  EXECUTE
    BEGIN
        UPDATE emp . . .
        ...
    EXCEPTION
        WHEN  DUP_VAL_ON_INDEX  THEN
            ROLLBACK ;
        ...
    END;
END-EXEC;
...
```

Migrating from Earlier Versions

Versions 1.4 and 1.5 of the ORACLE Precompilers generate faster, terser code than their predecessors. Moreover, with Version 1.5, you can check your old programs for ANSI/ISO compliance by specifying the FIPS option on the command line. So, when migrating from earlier versions, it is a good idea to re-precompile your programs.

In some cases, you might need to revise a program before re-precompiling it. Specifically, you might need to rename identifiers that conflict with new symbols (such as SQLBUF) used by the Version 1.4/1.5 runtime library.

When you re-precompile your programs, specify the same option settings that you specified originally. That way, your applications will run the way they always have.

CHAPTER

7

HANDLING RUNTIME ERRORS

An application program is more reliable if it anticipates runtime errors and attempts to recover from them. This chapter provides an in-depth discussion of error reporting and recovery. You learn how to handle errors and status changes using the SQL Communications Area (SQLCA) and the WHENEVER statement. You also learn how to diagnose problems using the ORACLE Communications Area (ORACA). The following topics are discussed:

- the need for error handling
- error handling alternatives
- key components of error reporting
- using the SQLCA
- declaring SQLCODE
- using the WHENEVER statement
- using the ORACA

Note: The SQLCA and ORACA variable names used in this chapter are generic. To see the names in a particular host language, refer to your *Supplement to the ORACLE Precompilers Guide.*

The Need for Error Handling

A significant part of every application program must be devoted to error handling. The main benefit of error handling is that it allows your program to continue operating in the presence of errors. Errors arise from design faults, coding mistakes, hardware failures, invalid user input, and many other sources.

You cannot anticipate all possible errors, but you can plan to handle certain kinds of errors meaningful to your program. For the ORACLE Precompilers, error handling means detecting and recovering from SQL statement execution errors.

You can also prepare to handle warnings such as "value truncated" and status changes such as "end of data."

It is especially important to check for error and warning conditions after every data manipulation statement, because an INSERT, UPDATE, or DELETE statement might fail before processing all eligible rows in a table.

Error Handling Alternatives

The SQLCA is a record-like, host-language data structure. ORACLE updates the SQLCA after every *executable* SQL statement. (SQLCA values are undefined after a declarative statement.) By checking ORACLE return codes stored in the SQLCA, your program can determine the outcome of a SQL statement. This can be done in the following two ways:

- implicit checking with the WHENEVER statement
- explicit checking of SQLCA variables

You can use WHENEVER statements, code explicit checks on SQLCA variables, or do both. Generally, using WHENEVER statements is preferable because it is easier, more portable, and ANSI-compliant.

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA. The ORACA is a host-language data structure that handles ORACLE communication. It contains cursor statistics, information about the current SQL statement, option settings, and system statistics.

Key Components of Error Reporting

Error reporting depends on variables in the SQLCA. This section highlights the key components of error reporting. The next section takes a close look at the SQLCA.

Status Codes

Every executable SQL statement returns a status code to the SQLCA variable SQLCODE, which you can check implicitly with the WHENEVER statement or explicitly with your own code.

A zero status code means that ORACLE executed the statement without detecting an error or exception. A positive status code means that ORACLE executed the statement but detected an exception. A negative status code means that ORACLE did not execute the SQL statement because of an error.

Warning Flags

Warning flags are returned in the SQLCA variables SQLWARN(1) through SQLWARN(8), which you can check implicitly or explicitly. These warning flags are useful for runtime conditions not considered errors by ORACLE. For example, when MODE=ORACLE, if an indicator variable is available, ORACLE signals a warning after assigning a truncated column value to a host variable. (If no indicator variable is available, ORACLE issues an error message.)

Rows-Processed Count

The number of rows processed by the most recently executed SQL statement is returned in the SQLCA variable SQLERRD(3), which you can check explicitly.

Speaking strictly, this variable is not for error reporting, but it can help you avoid mistakes. For example, suppose you expect to delete about ten rows from a table. After the deletion, you check SQLERRD(3) and find that 75 rows were processed. To be safe, you might want to roll back the deletion and examine your WHERE-clause search condition.

Parse Error Offset

Before executing a SQL statement, ORACLE must *parse* it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. If ORACLE finds an error, an offset is stored in the SQLCA variable SQLERRD(5), which you can check explicitly. The offset specifies the character position in the SQL statement at which the parse error begins. The first character occupies position zero. For example, if the offset is 9, the parse error begins at the 10th character.

By default, static SQL statements are checked for syntactic errors at precompiled time. So, SQLERRD(5) is most useful for debugging dynamic SQL statements, which your program accepts or builds at run time.

Parse errors arise from missing, misplaced, or misspelled keywords, invalid options, nonexistent tables, and the like. For example, the dynamic SQL statement

```
' UPDATE EMP SET JIB =  : job_title WHERE EMPNO = : emp_number'
```

causes the parse error

```
ORA-00904: invalid column name
```

because the column name JOB is misspelled. The value of SQLERRD(5) is 15 because the erroneous column name JIB begins at the 16th character.

If your SQL statement does not cause a parse error, ORACLE sets SQLERRD(5) to zero. ORACLE also sets SQLERRD(5) to zero if a parse error begins at the first character (which occupies position zero). So, check SQLERRD(5) only if SQLCODE is negative, which means that an error has occurred.

Error Message Text

The error code and message for ORACLE errors are available in the SQLCA variable SQLERRMC. At most, the first 70 characters of text are stored. To get the full text of messages longer than 70 characters, you use the SQLGLM function. See the section “Getting the Full Text of Error Messages” later in this chapter.

Using the SQL Communications Area (SQLCA)

The SQLCA is a record-like data structure. Its fields contain error, warning, and status information updated by ORACLE whenever a SQL statement is executed. Thus, the SQLCA always reflects the outcome of the most recent SQL operation. To determine the outcome, you can check variables in the SQLCA.

In host languages that allow local as well as global declarations, your program can have more than one SQLCA. For example, it might have one global SQLCA and several local ones. Access to a local SQLCA is limited by its scope within the program. ORACLE returns information only to the "active" SQLCA.

Note: When your application uses SQL*Net to access a combination of local and remote databases concurrently, all the databases write to one SQLCA. There is *not* a different SQLCA for each database. For more information, see the section "Concurrent Logons" in Chapter 3.

Declaring the SQLCA

When MODE= {ANSI13 | ORACLE}, declaring the SQLCA is required. To declare the SQLCA, you must hardcode it or copy it into your program with the INCLUDE statement, as follows:

```
EXEC SQL INCLUDE SQLCA ;
```

The SQLCA must be declared *outside* the Declare Section. Not declaring the SQLCA results in compile-time errors.

When you precompiled your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow ORACLE to communicate with the program.

When MODE={ANSI | ANSI14}, declaring the SQLCA is optional. However, you must declare a status variable named SQLCODE (SQLCOD in FORTRAN). If you declare SQLCODE instead of the SQLCA in a particular compilation unit, the precompiled allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA. If you declare the SQLCA *and* SQLCODE, ORACLE returns the same status code to both after every SQL operation.

Note: Declaring the SQLCA is optional when MODE={ANSI | ANSI14} but you cannot use the WHENEVER SQLWARNING statement without the SQLCA. So, if you want to use the WHENEVER SQLWARNING statement, you must declare the SQLCA.

What's in the SQLCA?

The SQLCA contains the following runtime information about the outcome of SQL statements:

- ORACLE error codes
- warning flags
- event information
- rows-processed count
- diagnostics

Figure 7-1 shows all the variables in the SQLCA. To see the SQLCA structure and variable names in a particular host language, refer to your *Supplement to the ORACLE Precompilers Guide.*

Figure 7-1
The SQLCA Variables

SQLCA

SQLCAID	**Character string "SQLCA"**
SQLCABC	**Length of SQLCA data structure in bytes**
SQLCODE	**ORACLE error message code**
SQLERRM	**Subrecord for storing error message**
SQLERRML	**Length of error message**
SQLERRMC	**Text of error message**
SQLERRP	**Reserved for future use**
SQLERRD	**Array of six integer status codes**
SQLERRD(1)	**Reserved for future use**
SQLERRD(2)	**Reserved for future use**
SQLERRD(3)	**Number of rows processed**
SQLERRD(4)	**Reserved for future use**
SQLERRD(5)	**Parse error offset**
SQLERRD(6)	**Reserved for future use**
SQLWARN	**Array of eight warning flags**
SQLWARN(1)	**Another warning flag set**
SQLWARN(2)	**Character string truncated**
SQLWARN(3)	**No longer in use**
SQLWARN(4)	**SELECT list not equal to INTO list**
SQLWARN(5)	**DELETE or UPDATE without WHERE clause**
SQLWARN(6)	**Reserved for future use**
SQLWARN(7)	**No longer in use**
SQLWARN(8)	**No longer in use**
SQLEXT	**Reserved for future use**

Note: In C, the indexing of array elements always starts at 0; for example, the third element of SQLERRD is sqlerrd[2].

Structure of the SQLCA

This section describes the structure of the SQLCA, its fields, and the values they can store.

SQLCAID

This string field is initialized to "SQLCA" to identify the SQL Communications Area.

SQLCABC

This integer field holds the length, in bytes, of the SQLCA structure.

SQLCODE

This integer field holds the status code of the most recently executed SQL statement. The status code, which indicates the outcome of the SQL operation, can be any of the following numbers:

- 0 — Means that ORACLE executed the statement without detecting an error or exception.
- > 0 — Means that ORACLE executed the statement but detected an exception. This occurs when ORACLE cannot find a row that meets your WHERE-clause search condition or when a SELECT INTO or FETCH returns no rows.

  When MODE= {ANSI | ANSI14 | ANSI13}, +100 is returned to SQLCODE after an INSERT of no rows. This can happen when a subquery returns no rows to process.
- < 0 — Means that ORACLE did not execute the statement because of a database, system, network, or application error. Such errors can be fatal. When they occur, the current transaction should, inmost cases, be rolled back.

  Negative return codes correspond to error codes listed in the *ORACLE7 Server Messages and Codes Manual.*

Table 7-1 shows how the value of SQLCODE depends on the MODE option

Table 7-1
SQLCODE Values

ORACLE Error or Warning	MODE	SQLCODE Scalars	SQLCODE Arrays
ORA-01403: no data found	ORACLE	+1403	+1 403
	ANSI13	+100	+100
	ANSI14	+100	n/a
ORA-01405: fetched column value is NULL (no indicator variable present)	ORACLE	0	0
	ANSI13	0	0
	ANSI14	–1405	n/a
ORA-01406: fetched column value was truncated	ORACLE	–1405	-1406
	ANSI13	0	0
	ANSI14	0	n/a

SQLERRM	This subrecord contains the following two fields:	
	SQLERRML	This integer field holds the length of the message text stored in SQLERRMC.
	SQLERRMC	This string field holds the message text corresponding to the error code stored in SQLCODE. This field can store up to 70 characters. To get the full text of messages longer than 70 characters, you must use the SQLGLM function (discussed later). Make sure SQLCODE is negative *before* you reference SQLERRMC. If you reference SQLERRMC when SQLCODE is zero, you get the message text associated with a prior SQL statement.
SQLERRP	This string field is reserved for future use.	
SQLERRD	This array of binary integers has six elements. Descriptions of the fields in SQLERRD follow:	
	SQLERRD(1)	This field is reserved for future use.
	SQLERRD(2)	This field is reserved for future use.
	SQLERRD(3)	This field holds the number of rows processed by the most recently executed SQL statement. However, if the SQL statement failed, the value of SQLERRD(3) is undefined, with one exception. If the error occurred during an array operation, processing stops at the row that caused the error, so SQLERRD(3) gives the number of rows processed successfully. The rows-processed count is zeroed after an OPEN statement and incremented after a FETCH statement. For the EXECUTE, INSERT, UPDATE, DELETE, and SELECT INTO statements, the count reflects the number of rows processed successfully. The count does *not* include rows processed by an update or delete cascade. For example, if 20 rows are deleted because they meet WHERE-clause criteria, and 5 more rows are deleted because they now (after the primary delete) violate column constraints, the count is 20 not 25.

SQLERRD(4) This field is reserved for future use.

SQLERRD(5) This field holds an offset that specifies the character position at which a parse error begins in the most recently executed SQL statement. The first character occupies position zero.

SQLERRD(6) This field is reserved for future use.

SQLWARN

This array of single characters has eight elements. They are used as warning flags. ORACLE sets a flag by assigning it a "W' (for warning) character value.

The flags warn of exceptional conditions. For example, a warning flag is set when ORACLE assigns a truncated column value to an output host variable.

Descriptions of the fields in SQLWARN follow:

SQLWARN(1) This flag is set if another warning flag is set.

SQLWARN(2) This flag is set if a truncated column value was assigned to an output host variable. This applies only to character data. ORACLE truncates certain numeric data without setting a warning or returning a negative SQLCODE.

To find out if a column value was truncated and by how much, check the indicator variable associated with the output host variable. The (positive) integer returned by an indicator variable is the original length of the column value. You can increase the length of the host variable accordingly.

SQLWARN(3) This flag is no longer in use.

SQLWARN(4) This flag is set if the number of columns in a query select list does not equal the number of host variables in the INTO clause of the SELECT or FETCH statement. The number of items returned is the lesser of the two.

SQLWARN(5)	This flag is set if every row in a table was processed by an UPDATE or DELETE statement without a WHERE clause. An update or deletion is called *unconditional* if no search condition restricts the number of rows processed. Such updates and deletions are unusual, so ORACLE sets this warning flag. That way, you can roll back the transaction if necessary.
SQLWARN(6)	This flag is set when an EXEC SQL CREATE {PROCEDURE I FUNCTION I PACKAGE I PACKAGE BODY} statement fails because of a PL/SQL compilation error.
SQLWARN(7)	This flag is no longer in use.
SQLWARN(8)	This flag is no longer in use.

SQLEXT — This string field is reserved for future use.

Declaring SQLCODE

When MODE= {ANSI I ANSI14}, you must declare a 4-byte integer variable named SQLCODE (SQLCOD in FORTRAN) inside or outside the Declare Section. An example follows:

```
-- declare host variables
EXEC SQL BEGIN DECLARE SECTION;
    emp_number     INTEGER ;
    emp_name       CHARACTER (10 ) ;
    dept_number    INTEGER ;
EXEC SQL END DECLARE SECTION;

-- declare status variable
    SQLCODE        INTEGER;
```

When MODE= {ANSI13 I ORACLE}, if you declare SQLCODE, it goes unused.

With host languages that allow local as well as global declarations, you can declare more than one SQLCODE. Access to a local SQLCODE is limited by its scope within your program.

After every SQL operation, ORACLE returns a status code to the SQLCODE currently in scope. So, your program can learn the outcome of the most recent SQL operation by checking SQLCODE explicitly, or implicitly with the WHENEVER statement.

When you declare SQLCODE instead of the SQLCA in a particular compilation unit, the precompiled allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA. If you declare the SQLCA *and* SQLCODE, ORACLE returns the same status code to both after every SQL operation.

The table below shows you how to name and type the status variable SQLCODE. Uppercase is required if your host language is case-sensitive.

Language	*Name*	*Datatype*
C	SQLCODE	4-byte int or long
COBOL	SQLCODE	PIC S9(9) COMP
FORTRAN	SQLCOD	INTEGER*4
Pascal	SQLCODE	INTEGER
PL/I	SQLCODE	FIXED BINARY (31)

Getting the Full Text of Error Messages

The SQLCA can accommodate error messages up to 70 characters long. To get the full text of longer (or nested) error messages, you need the SQLGLM function. If connected to ORACLE, you can call SQLGLM using the syntax

```
SQLGLM (message_buffer, buffer_size, message_ length) ;
```

where:

message_buffer Is the text buffer in which you want ORACLE to store the error message (ORACLE blank-pads to the end of this buffer).

buffer_size Is an integer variable that specifies the maximum size of the buffer in bytes.

message_length Is an integer variable in which ORACLE stores the actual length of the error message.

The maximum length of an ORACLE error message is 512 characters including the error code, nested messages, and message inserts such as table and column names. The maximum length of an error message returned by SQLGLM depends on the value you specify for *buffer_size*.

In the following example, you call SQLGLM to get an error message of up to 100 characters in length:

```
-- declare variables for function call
msg_buffer   CHARACTER(100) ;
buf_size     INTEGER ;
msg_length   INTEGER ;

set buf_size = 100;

EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
-- other  statements
...
sql_error:
    -- get full text of error message
    SQLGLM(msg_buffer,  buf_size,  msg_length);
    display contents of msg_buffer;
    ...
```

Notice that SQLGLM is called only when a SQL error has occurred. Always make sure SQLCODE is negative *before* calling SQLGLM. If you call SQLGLM when SQLCODE is zero, you get the message text associated with a prior SQL statement.

Using the WHENEVER Statement

By default, precompiled programs ignore ORACLE error and warning conditions and continue processing if possible. To do automatic condition checking and error handling, you need the WHENEVER statement.

With the WHENEVER statement you can specify actions to be taken when ORACLE detects an error, warning condition, or "not found" condition. These actions include continuing with the next statement, calling a routine, branching to a labeled statement, or stopping.

You code the WHENEVER statement using the following syntax:

```
EXEC SQL WHENEVER <condition> <action>;
```

You can have ORACLE automatically check the SQLCA for any of the following conditions.

SQLWARNING

SQLWARN(1) is set because ORACLE returned a warning (one of the warning flags, SQLWARN(2) through SQLWARN(8), is also set) or SQLCODE has a positive value other than +1403. For example, SQLWARN(1) is set when ORACLE assigns a truncated column value to an output host variable.

Declaring the SQLCA is optional when MODE= {ANSI | ANSI14}. To use WHENEVER SQLWARNING, however, you *must* declare the SQLCA. .

SQLERROR

SQLCODE has a negative value because ORACLE returned an error.

NOT FOUND

SQLCODE has a value of +1403 (+100 when MODE= {ANSI | ANSI14 | ANSI13}) because ORACLE could not find a row that meets your WHERE-clause search condition, or a SELECT INTO or FETCH returned no rows.

When MODE={ANSI | ANSI14 | ANSI13}, +100 is returned to SQLCODE after an INSERT of no rows.

When ORACLE detects one of the preceding conditions, you can have your program take any of the following actions.

CONTINUE

Your program continues to run with the next statement if possible. This is the default action, equivalent to not using the WHENEVER statement. You can use it to "turn off" condition checking.

DO routine_call

Your program transfers control to an internal routine. When the end of the routine is reached, control transfers to the statement that follows the failed SQL statement.

A *routine is any functional* program unit that can be invoked such as a C function, COBOL paragraph, FORTRAN subroutine, Pascal procedure, or PL/I internal procedure. In this context, separately compiled programs such as COBOL subroutines or external PL/I procedures are *not* routines.

The usual rules for entering and exiting a routine apply. However, passing parameters to the routine is *not* allowed. Furthermore, the routine must *not return* a value.

The parameter *routine_call* is a host language invocation, as in

```
EXEC SQL WHENEVER. . . DO function_name ( ) ;                  —— C
EXEC SQL WHENEVER. . . DO PERFORM paragraph_name END-EXEC. -- COBOL
EXEC SQL WHENEVER. . . DO CALL subroutine_name             -- FORTRAN
EXEC SQL WHENEVER. . . DO procedure_name;                  -- Pascal
EXEC SQL WHENEVER. . . DO CALL procedure_name;             -- PL/I
```

Notice that in C, the function name must be followed by an empty argument list.

GOTO label_name

Your program branches to a labeled statement.

STOP

Your program stops running and uncommitted work is rolled back.

Be careful. The STOP action displays no messages before logging off ORACLE. In Pascal, the STOP action is illegal because Pascal has no equivalent command.

Some Examples

If you want your program to

- go to *close_cursor* if a "no data found" condition occurs,
- continue with the next statement if a warning occurs, and
- go to *error_handler* if an error occurs

simply code the following WHENEVER statements before the first executable SQL statement

```
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO error_handler;
```

In the following example, you use WHENEVER...DO statements to handle specific errors:

```
...
EXEC SQL WHENEVER SQLERROR DO handle_insert_error;
EXEC SQL INSERT INTO = ( EMPNO , ENAME , DEPTNO )
    VALUES (: emp_number, : emp_name, : dept_number );

EXEC SQL WHENEVER SQLERROR DO handle_delete_error;
EXEC SQL DELETE FROM DEPT WHERE DEPTNO = : dept_number;
...
ROUTINE  handle_insert_error;
    BEGIN
        IF sqlca.sqlcode = -1 THEN   -- duplicate key value
            ...
        ELSEIF sqlca.sqlcode = -1401 THEN -- value too large
            ...
        ENDIF;
        ...
    END;

ROUTINE  handle_delete_error;
    BEGIN
        IF sqlca.sqlerrd(3) = 0 THEN -- no rows deleted
            ...
        ELSE
            ...
        ENDIF;
        ...
    END;
...
```

Notice how the procedures check variables in the SQLCA to determine a course of action.

Scope of WHENEVER

Because WHENEVER is a declarative statement, its scope is positional, not logical. That is, it tests all executable SQL statements that physically follow it in the source file, not in the flow of program logic. So, code the WHENEVER statement before the first executable SQL statement you want to test.

A WHENEVER statement stays in effect until superseded by another WHENEVER statement checking for the same condition.

In the example below, the first WHENEVER SQLERROR statement is superseded by a second, and so applies only to the CONNECT statement The second WHENEVER SQLERROR statement applies to both the UPDATE and DROP statements, despite the flow of control from *step1* to *step3.*

```
stepl:
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC SQL CONNECT :username IDENTIFIED BY : password;
    ...
    GOTO step3;

step2:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL UPDATE EMP SET SAL = SAL * 1.10;
    ...

step3:
    EXEC SQL DROP INDEX EMP_INDEX ;
    ...
```

Guidelines

The following guidelines will help you avoid some common pitfalls.

Placing the Statements

In general, code a WHENEVER statement before the first executable SQL statement in your program. This ensures that all ensuing errors are trapped because WHENEVER statements stay in effect to the end of a file.

Handling End-of-Data Conditions

Your program should be prepared to handle an end-of-data condition when using a cursor to fetch rows. If a FETCH returns no data, the program should branch to a labeled section of code where a CLOSE command is issued, as follows:

```
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
...
no_more:
    ...
    EXEC SQL CLOSE my_cursor;
    ...
```

Avoiding Infinite Loops

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes an executable SQL statement, your program might enter an infinite loop if the SQL statement fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the SQL statement, as shown in the following example:

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    ...
```

Without the WHENEVER SQLERROR CONTINUE statement, a ROLLBACK error would invoke the routine again, starting an infinite loop.

Careless use of WHENEVER can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets NOT FOUND because no rows meet the search condition

```
-- improper use of WHENEVER
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;

LOOP
    EXEC SQL FETCH emp_cursor INTO : emp_name, : salary;
    ...
ENDLOOP;

no_more:
    EXEC SQL DELETE FROM EMP WHERE EMPNO = : emp_number;
     ...
```

In the next example, you handle the NOT FOUND condition properly by resetting the GOTO target

```
-- proper use of WHENEVER
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;

LOOP
    EXEC SQL FETCH emp_cursor INTO : emp_name, : salary;
    ...
ENDLOOP;

no_more:
    EXEC SQL WHENEVER NOT FOUND GOTO no_match;
    EXEC SQL DELETE FROM EMP WHERE EMPNO = : emp_number;
    ...
no_match:
    ...
```

Maintaining Addressability

With host languages that allow local as well as global identifiers, make sure all SQL statements governed by a WHENEVER GOTO statement can branch to the GOTO label. The following code results in a compile-time error because *labelA* in FUNC1 is not within the scope of the INSERT statement in FUNC2:

```
FUNC 1
    BEGIN
    EXEC SQL WHENEVER SQLERROR GOTO labelA;
    EXEC SQL DELETE FROM EMP WHERE DEPTNO = : dept_number;
    ...
    labelA:
        ...
    END;

FUNC2
    BEGIN
    EXEC SQL INSERT INTO EMP (JOB) VALUES (: job_title) ;
    ...
    END;
```

The label to which a WHENEVER GOTO statement branches must be in the same precompilation file as the statement.

Returning after an Error

If your program must return after handling an error, use the DO *routine_call* action. Alternatively, you can test the value of SQLCODE, as shown in the following example:

```
...
EXEC SQL UPDATE EMP SET SAL = SAL * 1.10;

IF sqlca.sqlcode < 0 THEN
    -- handle error

EXEC SQL DROP INDEX EMP_INDEX ;
```

Just make sure no WHENEVER GOTO or WHENEVER STOP statement is active.

Using the ORACLE Communications Area (ORACA)

The SQLCA handles standard SQL communications; the ORACA handles ORACLE communications. When you need more information about runtime errors and status changes than the SQLCA provides, use the ORACA. It contains an extended set of diagnostic tools. However, use of the ORACA is optional because it adds to runtime overhead.

Besides helping you to diagnose problems, the ORACA lets you monitor your program's use of ORACLE resources such as the SQL Statement Executor and the cursor cache.

In host languages that allow local as well as global declarations, your program can have more than one ORACA. For example, it might have one global ORACA and several local ones. Access to a local ORACA is limited by its scope within the program. ORACLE returns information only to the "active" ORACA.

Declaring the ORACA

To declare the ORACA, you can hardcode it or simply copy it into your program with the INCLUDE statement, as follows:

```
EXEC SQL INCLUDE ORACA;
```

The ORACA must be declared *outside* the Declare Section.

When you precompiled your program, the INCLUDE ORACA statement is replaced by several program variable declarations. These declarations allow ORACLE to communicate with your program.

Enabling the ORACA

To enable the ORACA, you must specify the ORACA option, either on the command line with

```
ORACA=YES
```

or inline with

```
EXEC ORACLE OPTION ( ORACA=YES ) ;
```

Then, you must choose appropriate runtime options by setting flags in the ORACA.

What's in the ORACA? The ORACA contains option settings, system statistics, and extended diagnostics such as

- SQL statement text (you can specify when to save the text)
- name of the file in which an error occurred (useful when using subroutines)
- location of the error in a file
- cursor cache errors and statistics

Figure 7-2 shows all the variables in the ORACA. To see the ORACA structure and variable names in a particular host language, refer to your Supplement to the ORACLE Precompilers Guide.

Figure 7-2
The ORACA Variables

ORACA

ORACAID	**Character string "ORACA"**
ORACABC	**Length of ORACA data structure in bytes**
ORACCHF	**Cursor cache consistency flag**
ORADBGF	**Master debug flag**
ORAHCHF	**Heap consistency flag**
ORASTXTF	**Save-SQL-statement flag**
ORASTXT	**Subrecord for storing SQL statement-**
ORASTXTL	**Lenth of current SQL statement**
ORASTXTC	**Text of current SQL statement**
ORASFNM	**Subrecord for storing filename**
ORASFNML	**Length of filename**
ORASFNMC	**Name of file containing current SQL statement**
ORASLNR	**Line in file at or near current SQL statement**
ORAHOC	**Highest MAXOPENCURSORS requested**
ORAMOC	**Maximum open cursors required**
ORACOC	**Current number of cursors used**
ORANOR	**Number of cursor cache reassignments**
ORANPR	**Number of SQL statement parses**
ORANEX	**Number of SQL statement executions**

Choosing Runtime Options

The ORACA includes several option flags. Setting these flags by assigning them non-zero values allows you to

- save the text of SQL statements
- enable DEBUG operations
- check cursor cache consistency (the *cursor cache* is a continuously updated area of memory used for cursor management)
- check heap consistency (the *heap* is an area of memory reserved for dynamic variables)
- gather cursor statistics

The descriptions below will help you choose the options you need.

Structure of the ORACA

This section describes the structure of the ORACA, its fields, and the values they can store.

ORACAID

This string field is initialized to "ORACA" to identify the ORACLE Communications Area.

ORACABC

This integer field holds the length, in bytes, of the ORACA data structure.

ORACCHF

If the master DEBUG flag (ORADBGF) is set, this flag enables the gathering of cursor cache statistics and lets you check the cursor cache for consistency before every cursor operation.

The ORACLE runtime library does the consistency checking and might issue error messages, which are listed in Appendix D. They are returned to the SQLCA just like ORACLE error messages.

This flag has the following settings:

0	Disable cache consistency checking (the default).
1	Enable cache consistency checking.

ORADBGF

This master flag lets you choose all the DEBUG options. It has the following settings:

0	Disable all DEBUG operations (the default).
1	Enable all DEBUG operations.

ORAHCHF

If the master DEBUG flag (ORADBGF) is set, this flag tells the ORACLE runtime library to check the heap for consistency every time the precompiler dynamically allocates or frees memory. This is useful for detecting program bugs that upset memory.

This flag must be set before the CONNECT command is issued and, once set, cannot be cleared; subsequent change requests are ignored. It has the following settings:

0	Disable heap consistency checking (the default).
1	Enable heap consistency checking.

ORASTXTF

This flag lets you specify when the text of the current SQL statement is saved. It has the following settings:

0	Never save the SQL statement text (the default).
1	Save the SQL statement text on SQLERROR only.
2	Save the SQL statement text on SQLERROR or SQLWARNING.
3	Always save the SQL statement text.

The SQL statement text is saved in the ORACA subrecord named ORASTXT.

Diagnostics

The ORACA provides an enhanced set of diagnostics; the following variables help you to locate errors quickly.

ORASTXT

This subrecord helps you find faulty SQL statements. It lets you save the text of the last SQL statement parsed by ORACLE. It contains the following two fields:

ORASTXTL	This integer field holds the length of the current SQL statement.
ORASTXTC	This string field holds the text of the current SQL statement. At most, the first 70 characters of text are saved.

Statements parsed by the precompiled, such as CONNECT, FETCH, and COMMIT, are *not* saved in the ORACA.

ORASFNM

This subrecord identifies the file containing the current SQL statement and so helps you find errors when multiple files are precompiled for one application. It contains the following two fields:

ORASFNML	This integer field holds the length of the filename stored in ORASFNMC.
ORASFNMC	This string field holds the filename. At most, the first 70 characters are stored.

ORASLNR

This integer field identifies the line at (or near) which the current SQL statement can be found.

Cursor Cache Statistics

If the master DEBUG flag (ORADBGF) and the cursor cache flag (ORACCHF) are set, the variables below let you gather cursor cache statistics. They are automatically set by every COMMIT or ROLLBACK command your program issues.

Internally, there is a set of these variables for each CONNECTed database. The current values in the ORACA pertain to the database against which the last COMMIT or ROLLBACK was executed.

ORAHOC

This integer field records the highest value to which MAXOPENCURSORS was set during program execution.

ORAMOC

This integer field records the maximum number of open ORACLE cursors required by your program. This number can be higher than ORAHOC if MAXOPENCURSORS was set too low, which forced the precompiler to extend the cursor cache.

ORACOC

This integer field records the current number of open ORACLE cursors required by your program.

ORANOR

This integer field records the number of cursor cache reassignments required by your program. This number shows the degree of "thrashing" in the cursor cache and should be kept as low as possible.

ORANPR

This integer field records the number of SQL statement parses required by your program.

ORANEX

This integer field records the number of SQL statement executions required by your program. The ratio of this number to the ORANPR number should be kept as high as possible. In other words, avoid unnecessary reparsing. For help, see Appendix E.

An Example

The following program prompts for a department number, inserts the name and salary of each employee in that department into one of two tables, then displays diagnostic information from the ORACA:

```
EXEC SQL BEGIN DECLARE SECTION;
    username      CHARACTER (20) ;
    password      CHARACTER (20) ;
    emp_name      INTEGER;
    dept_number  INTEGER;
    salary        REAL ;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA ;
EXEC SQL INCLUDE ORACA ;

display  ' Username? ' ;
read username;
display  ' Password? ' ;
read password;

EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL CONNECT :username IDENTIFIED BY : password;

display  'Connected to ORACLE';

EXEC ORACLE OPTION (ORACA=YES);

-- set flags in the ORACA
set oraca.oradbgf = 1; -- enable debug operations
set oraca.oracchf = 1;  -- gather cursor cache statistics
set oraca.orastxtf = 3; -- always save the SQL statement

display  'Department number? ';
read dept_number;

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, SAL + NVL(COMM,0)
    FROM EMP
    WHERE DEPTNO =  :dept_number;

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO no_more;
```

```
LOOP
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    IF salary < 2500 THEN
        EXEC SQL INSERT INTO PAY1 VALUES (:emp_name, :salary);
    ELSE
        EXEC SQL INSERT INTO PAY2 VALUES (:emp_name, :salary);
    ENDIF;
ENDLOOP;

no_more:
    EXEC SQL CLOSE emp_cursor;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL COMMIT WORK RELEASE;
    display 'Last SQL statement: ', oraca.orastxt.orastxtc;
    display '... at or near line number: ', oraca.oraslnr;
    display
    display '          Cursor Cache Statistics';
    display ' ---------------------------------------- ' ;
    display 'Maximum value of MAXOPENCURSORS ', oraca.orahoc;
    display 'Maximum open cursors required:       ', oraca.oramoc;
    display 'Current number of open cursors: ', oraca.oracoc;
    display 'Number of cache reassignments:       ', oraca.oranor;
    display 'Number of SQL statement parses: ', oraca.oranpr;
    display 'Number of SQL statement executions: ', oraca.oranex;
    exit program;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    display 'Last SQL statement: ', oraca.orastxt.orastxtc;
    display '... at or near line number: ' , oraca.oraslnr;
    display
    display '          Cursor Cache Statistics';
    display ' ------- ------------- -------------------- ' ;
    display 'Maximum value of MAXOPENCURSORS ', oraca.orahoc;
    display 'Maximum open cursors required:       ', oraca.oramoc;
    display 'Current number of open cursors: ', oraca.oracoc;
    display 'Number of cache reassignments:       ', oraca.oranor;
    display 'Number of SQL statement parses: ', oraca.oranpr;
    display 'Number of SQL statement executions: ', oraca.oranex;
    exit program with an error;
```

CHAPTER

8

USING HOST ARRAYS

This chapter looks at using arrays to simplify coding and improve program performance. You learn how to manipulate ORACLE data using arrays, how to operate on all the elements of an array with a single SQL statement, and how to limit the number of array elements processed. The following questions are answered:

- What is a host array?
- Why use arrays?
- How are host arrays declared?
- How are arrays used in SQL statements?

What Isa Host Array?

An *array* is a collection of related data items, called elements, associated with a single variable name. When declared as a host variable, the array is called a *host array*. Likewise, an indicator variable declared as an array is called an *indicator array*. An indicator array can be associated with any host array.

Why Use Arrays?

Arrays can ease programming and offer improved performance.

When writing an application, you are usually faced with the problem of storing and manipulating large collections of data. Arrays simplify the task of naming and referencing the individual items in each collection.

Using arrays can boost the performance of your application. Arrays let you manipulate an entire collection of data items with a single SQL statement Thus, ORACLE communication overhead is reduced markedly, especially in a networked environment.

For example, suppose you want to insert information about 300 employees into the EMP table. Without arrays your program must do 300 individual INSERTs—one for each employee. With arrays, only one INSERT need be done.

Declaring Host Arrays

You declare host arrays, like simple host variables, in the Declare Section. You also *dimension* (set the size of) host arrays in the Declare Section. In the following example, you declare three host arrays and dimension them with 50 elements:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_name (50)     CHARACTER (20) ;
    emp_number (50)   INTEGER;
    salary (50)       REAL;
EXEC SQL END DECLARE SECTION;
```

Restrictions

You cannot declare host arrays of pointers.

Host arrays that might be referenced in a SQL statement are limited to one dimension. So, the two-dimensional array declared in the following example is *invalid:*

```
EXEC SQL BEGIN DECLARE SECTION;
    hi_lo_scores (25, 25)   INTEGER;    -- not allowed
EXEC SQL END DECLARE SECTION;
```

In Pro*C, although declared as two-dimensional, an array of character arrays is treated as a one-dimensional array of strings.

Dimensioning Arrays

The maximum dimension of a host array is 32,767 elements. If you use a host array that exceeds the maximum, you get a "parameter out of range" runtime error.

If you use multiple host arrays in a single SQL statement, their dimensions should be the same. Otherwise, an "array size mismatch' warning message is issued at precompiled time. If you ignore this warning, the precompiled uses the *smallest* dimension for the SQL operation.

Using Arrays in SQL Statements

The ORACLE Precompilers allow the use of host arrays in data manipulation statements. You can use host arrays as input variables in the INSERT, UPDATE, and DELETE statements and as output variables in the INTO clause of SELECT and FETCH statements.

Note: When MODE=ANSI14, array operations are *not* allowed. In other words, you can reference host arrays in an INSERT, UPDATE, DELETE, SELECT, or FETCH statement only when MODE= {ANSI | ANSI13 | ORACLE}.

The syntax used for host arrays and simple host variables is nearly the same. One difference is the optional FOR clause, which lets you control array processing. Also, there are restrictions on mixing host arrays and simple host variables in a SQL statement.

The following sections illustrate the use of host arrays in data manipulation statements.

Selecting into Arrays

You can use host arrays as output variables in the SELECT statement. If you know the maximum number of rows the SELECT will return, simply dimension the host arrays with that number of elements. In the following example, you select directly into three host arrays. Knowing the SELECT will return no more than 50 rows, you dimension the arrays with 50 elements:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_name (50)     CHARACTER (20);
    emp_number (50)   INTEGER;
    salary (50)       REAL;
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT ENAME, EMPNO, SAL
    INTO emp_name, :emp_number, : salary
    FROM EMP
    WHERE SAL > 1000;
```

In this example, the SELECT statement returns up to 50 rows. If there are fewer than 50 eligible rows or you want to retrieve only 50 rows, this method will suffice. However, if there are more than 50 eligible rows, you cannot retrieve all of them this way. If you reexecute the SELECT statement, it just returns the first 50 rows again, even if more are eligible. You must either dimension a larger array or declare a cursor for use with the FETCH statement.

If a SELECT INTO statement returns more rows than the number of elements you dimensioned, ORACLE issues the error message

```
ORA-02112:  PCC : SELECT . . . INTO returns too many rows
```

unless you specify SELECT_ERROR=NO. For more information about the SELECT_ERROR option, seethe section “Using the Precompiled Options” in Chapter 11.

Batch Fetches

If you do not know the maximum number of rows a SELECT will return, you can declare and open a cursor, then fetch from it in 'batches."

Batch fetches within a loop let you retrieve a large number of rows with ease. Each FETCH returns the next batch of rows from the current active set. In the following example, you fetch in 20-row batches:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_number (20)   INTEGER;
    salary (20)       REAL;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT EMPNO, SAL FROM EMP;

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO . . .

LOOP
    EXEC SQL FETCH emp_cursor
        INTO : emp_number, : salary;
    -- process batch of rows
ENDLOOP;
```

Number of Rows Fetched

Each FETCH returns, at most, the number of rows in the array dimension. Fewer rows are returned in the following cases:

- The end of the active set is reached. The "no data found" ORACLE error code is returned to SQLCODE in the SQLCA. For example, this happens if you fetch into an array of dimension 100 but only 20 rows are returned.
- Fewer than a full batch of rows remain to be fetched. For example, this happens if you fetch 70 rows into an array of dimension 20 because after the third FETCH, only 10 rows remain to be fetched.
- An error is detected while processing a row. The FETCH fails and the applicable ORACLE error code is returned to SQLCODE.

The cumulative number of rows returned can be found in the third element of SQLERRD in the SQLCA, called SQLERRD(3) in this guide. This applies to each open cursor. In the following example, notice how the status of each cursor is maintained separately:

```
EXEC SQL OPEN cursor1;
EXEC SQL OPEN cursor2 ;
EXEC SQL FETCH cursor1 INTO : array_of_20;
    -- now running total in SQLERRD( 3 ) is 20
EXEC SQL FETCH cursor2 INTO : array_of_30;
    -- now running total in SQLERRD(3) is 30, not 50
EXEC SQL FETCH cursor1 INTO :array_of_20;
    -- now running total in SQLERRD(3) is 40 (20 + 20)
EXEC SQL FETCH cursor2 INTO :array_of_30;
    -- now running total in SQLERRD(3) is 60 (30 + 30)
```

Restrictions

Using host arrays in the WHERE clause of a SELECT statement is *not* allowed except in a subquery. For an example, see the section "Using the WHERE Clause" later in this chapter.

Also, you cannot mix simple host variables with host arrays in the INTO clause of a SELECT or FETCH statement. If any of the host variables is an array, all must be arrays.

The following table shows which uses of host arrays are valid in a SELECT INTO statement

INTO Clause	*WHERE Clause*	*Valid?*
array	array	no
scalar	scalar	yes
array	scalar	yes
scalar	array	no

Fetching Nulls

When MODE= {ANSI13 | ORACLE}, if you SELECT or FETCH null column values into a host array not associated with an indicator array, no error is generated. So, when doing array SELECTS and FETCHes, always use indicator arrays. That way, you can test for nulls in the associated output host array.

When MODE=ANSI, if you SELECT or FETCH a null column value into a host array not associated with an indicator array, ORACLE stops processing, sets SQLERRD(3) to the number of rows processed, and issues the following error message:

```
ORA-01405: fetched column value is NULL
```

Fetching Truncated Values

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host array not associated with an indicator array, ORACLE stops processing, sets SQLERRD(3) to the number of rows processed, and issues the following error message:

```
ORA-01406: fetched column value was truncated
```

When MODE=ANSI13, ORACLE stops processing, sets SQLERRD(3) to the number of rows processed, but issues no error message.

In either case, you can check SQLERRD(3) for the number of rows processed before the truncation occurred. The rows-processed count includes the row that caused the truncation error.

When MODE=ANSI, truncation is not considered an error, so ORACLE continues processing.

Again, when doing array SELECTS and FETCHes, always use indicator arrays. That way, if ORACLE assigns one or more truncated column values to an output host array, you can find the original lengths of the column values in the associated indicator array.

Inserting with Arrays

You can use host arrays as input variables in an INSERT statement. Just make sure your program populates the arrays with data before executing the INSERT statement.

If some elements in the arrays are irrelevant, you can use the FOR clause to control the number of rows inserted. See the section "Using the FOR Clause" later in this chapter.

An example of inserting with host arrays follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_name (50)     CHARACTER (20);
    emp_number (50)   INTEGER;
    salary (50)       REAL;
EXEC SQL END DECLARE SECTION;

-- populate the host arrays

EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
    VALUES ( :emp_name,  :emp_number, : salary);
```

The cumulative number of rows inserted can be found in SQLERRD(3).

Although functionally equivalent to the following statement, the INSERT statement in the last example is much more efficient because it issues only one call to ORACLE:

```
FOR i = 1 TO array_dimension
    EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
       VALUES (:emp_name[il, :emp_number[i], :salary[i]);
ENDFOR;
```

In this imaginary example (imaginary because host variables *cannot* be subscripted in a SQL statement), you use a FOR loop to access all array elements in sequential order.

Restrictions

You cannot use an array of pointers in the VALUES clause of an INSERT statement; all array elements must be data items.

Mixing simple host variables with host arrays in the VALUES clause of an INSERT statement is *not* allowed. If any of the host variables is an array, all must be arrays.

Updating with Arrays

You can also use host arrays as input variables in an UPDATE statement, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_number (50)   INTEGER;
    salary (50)       REAL;
EXEC SQL END DECLARE SECTION;

-- populate the host arrays

EXEC SQL UPDATE EMP SET SAL = :salary
    WHERE EMPNO =  :emp_number;
```

The cumulative number of rows updated can be found in SQLERRD(3). The number does *not* include rows processed by an update cascade.

If some elements in the arrays are irrelevant, you can use the FOR clause to limit the number of rows updated.

The last example showed a typical update using a unique key *(emp_number)*. Each array element qualified just one row for updating. In the following example, each array element qualifies multiple rows:

```
EXEC SQL BEGIN DECLARE SECTION;
    job_title (10)    CHARACTER(10);
    commission (50)   REAL;
EXEC SQL END DECLARE SECTION;

-- populate the host arrays

EXEC SQL UPDATE EMP SET COMM = : commission
    WHERE JOB =  :job_title;
```

Restrictions

Mixing simple host variables with host arrays in the SET or WHERE clause of an UPDATE statement is *not* allowed. If any of the host variables is an array, all must be arrays. Furthermore, if you use a host array in the SET clause, you must use one in the WHERE clause. However, their dimensions and datatypes need not match.

You cannot use host arrays with the CURRENT OF clause in an UPDATE statement. For an alternative, see the section "Mimicking CURRENT OF" later in this chapter.

The following table shows which uses of host arrays are valid in an UPDATE statement

SET Clause	*WHERE Clause*	*Valid?*
array	array	yes
scalar	scalar	yes
array	scalar	no
scalar	array	no

Deleting with Arrays

You can also use host arrays as input variables in a DELETE statement. It is like executing the DELETE statement repeatedly using successive elements of the host array in the WHERE clause. Thus, each execution might delete zero, one, or more rows from the table.

An example of deleting with host arrays follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    emp_number (50)   INTEGER;
EXEC SQL END DECLARE SECTION;

-- populate the host array

EXEC SQL DELETE FROM EMP
    WHERE EMPNO = :emp_number;
```

The cumulative number of rows deleted can be found in SQLERRD(3). The number does ***not*** include rows processed by a delete cascade.

The last example showed a typical delete using a unique key *(emp_number)*. Each array element qualified just one row for deletion. In the following example, each array element qualifies multiple rows:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    job_title (10)   CHARACTER (10 ) ;
EXEC SQL END DECLARE SECTION;

-- populate the host array

EXEC SQL DELETE FROM EMP
    WHERE JOB =  :job_title;
```

Restrictions

Mixing simple host variables with host arrays in the WHERE clause of a DELETE statement is *not* allowed. If any of the host variables is an array, all must be arrays.

You cannot use host arrays with the CURRENT OF clause in a DELETE statement. For an alternative, see the section "Mimicking CURRENT OF" later in this chapter.

Using Indicator Arrays

You use indicator arrays to assign nulls to input host arrays and to detect null or truncated values in output host arrays. The following example shows how to insert with indicator arrays:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_number (50)    INTEGER;
    dept_number (50)   INTEGER;
    commission (50)    REAL;
    ind_comm (50)      SMALLINT;  -- indicator array
EXEC SQL END DECLARE SECTION;


-- populate the indicator array; to insert a null into
-- the COMM column, assign -1 to the appropriate e element in
-- the indicator array

EXEC SQL INSERT INTO EMP ( EMPNO, DEPTNO, COMM)
    VALUES (: emp_number, : dept_number, : commission: ind_comm);
```

The indicator array dimension cannot be smaller than the host array dimension.

Using the FOR Clause

You can use the optional FOR clause to set the number of array elements processed by any of the following SQL statements:

- DELETE
- EXECUTE
- FETCH
- INSERT
- OPEN
- UPDATE

The FOR clause is especially useful in UPDATE, INSERT, and DELETE statements. With these statements you might not want to use the entire array. The FOR clause lets you limit the elements used to just the number you need, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_name (100)   CHARACTER (20) ;
    salary (100)     REAL;
    rows_to_insert   INTEGER;
EXEC SQL END DECLARE SECTION;

-- populate the host arrays

set rows_to_insert = 25;    -- set FOR-clause variable

EXEC SQL FOR : rows_to_insert   -- will process only 25 rows
    INSERT INTO EMP (ENAME, SAL)
    VALUES ( :emp_name, : salary);
```

The FOR clause must use an integer host variable to count array elements. For example, the following statement is illegal:

```
EXEC SQL FOR 25  -- illegal
    INSERT INTO EMP ( ENAME , EMPNO , SAL )
    VALUES ( :emp_name, :emp_number, : salary) ;
```

The FOR-clause variable specifies the number of array elements to be processed. Make sure the number is not larger than the smallest array dimension. Also, the number must be positive. If it is negative or zero, no rows are processed and ORACLE issues an error message.

Restrictions

Two restrictions keep FOR clause semantics clear. You cannot use the FOR clause in a SELECT statement or with the CURRENT OF clause.

In a SELECT Statement

If you use the FOR clause in a SELECT statement, you get the following error message:

```
PCC-E-0056 :  FOR clause not allowed on SELECT statement at . . .
```

The FOR clause is not allowed in SELECT statements because its meaning is unclear. Does it mean "execute this SELECT statement *n* times"? Or, does it mean "execute this SELECT statement once, but return *n* rows"? The problem in the former case is that each execution might return multiple rows. In the latter case, it is better to declare a cursor and use the FOR clause in a FETCH statement, as follows:

```
EXEC SQL FOR : limit FETCH emp_cursor INTO . . .
```

With the CURRENT OF Clause

You can use the CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row returned by a FETCH statement, as the following example shows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, SAL FROM EMP WHERE EMPNO = : emp_number;
...
EXEC SQL OPEN emp_cursor;
...
EXEC SQL FETCH emp_cursor INTO : emp_name, : salary;
...
EXEC SQL UPDATE EMP SET SAL = : new_salary
    WHERE CURRENT OF emp_cursor;
```

However, you cannot use the FOR clause with the CURRENT OF clause. The following statements are invalid because the only logical value of *limit* is 1 (you can only update or delete the current row once):

```
EXEC SQL FOR : limit UPDATE EMP SET SAL = : new_salary
    WHERE CURRENT OF emp_cursor;
...
EXEC SQL FOR :limit DELETE FROM EMP
    WHERE CURRENT OF emp_cursor;
```

Using the WHERE Clause

ORACLE treats a SQL statement containing host arrays of dimension *n* like the same SQL statement executed *n* times with n different scalar variables (the individual array elements). The precompiled issues the following error message only when such treatment would be ambiguous:

```
PCC-S-0055: Array <name> not allowed as bind variable at . . .
```

For example, assuming the declarations

```
EXEC SQL BEGIN DECLARE SECTION;
    mgr_number (50)   INTEGER ;
    job_title (50)    CHARACTER (20) ;
EXEC SQL END DECLARE SECTION;
```

it would be ambiguous if the statement

```
EXEC SQL SELECT MGR INTO : mgr_number FROM EMP
    WHERE JOB =  :job_title;
```

were treated like the imaginary statement

```
FOR i = 1 TO 50
    SELECT MGR INTO :mgr_number [ i ] FROM EMP
        WHERE JOB = : job_title [i] ;
ENDFOR;
```

because multiple rows might meet the WHERE-clause search condition, but only one output variable is available to receive data. Therefore, an error message is issued.

On the other hand, it would not be ambiguous if the statement

```
EXEC SQL UPDATE EMP SET MGR = : mgr_number
    WHERE EMPNO IN ( SELECT EMPNO FROM EMP
                         WHERE JOB =  :job_title) ;
```

were treated like the imaginary statement

```
FOR i = 1 TO 50
    UPDATE EMP SET MGR = :mgr_number[i]
        WHERE EMPNO IN ( SELECT EMPNO FROM EMP
                             WHERE JOB =  :job_title [i] ) ;
ENDFOR;
```

because there is a *mgr_number* in the SET clause for each row matching *job_title* in the WHERE clause, even if each *job_title* matches multiple rows. All rows matching each *job_title* can be SET to the same *mgr_number*. Therefore, no error message is issued.

Mimicking CURRENT OF

You use the CURRENT OF *cursor* clause in a DELETE or UPDATE statement to refer to the latest row FETCHed from the cursor. (For more information, see the section "Using the CURRENT OF Clause" in Chapter 4.) However, you cannot use CURRENT OF with host arrays. Instead, select the ROWID of each row, then use that value to identify the current row during the update or delete. An example follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_name (25)    CHARACTER (20) ;
    job_title (25)   CHARACTER (15) ;
    old_title (25)   CHARACTER (15) ;
    row_id (25)      CHARACTER (18) ;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , JOB , ROWID FROM EMP ;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO . . .
LOOP
    EXEC SQL FETCH emp_cursor
        INTO : emp_name, : job_title, : row_id;
    ...
    EXEC SQL DELETE FROM EMP
        WHERE JOB = : old_title AND ROWID = : row_id;
    EXEC SQL COMMIT WORK;
ENDLOOP;
```

However, the fetched rows are *not* locked because no FOR UPDATE OF clause is used. (You cannot use FOR UPDATE OF without CURRENT OF.) So, you might get inconsistent results if another user changes a row after you read it but before you delete it.

Using SQLERRD(3)

For INSERT, UPDATE, DELETE, and SELECT INTO statements, SQLERRD(3) records the number of rows processed. For FETCH statements, it records the cumulative sum of rows processed.

When using host arrays with FETCH, to find the number of rows returned by the most recent iteration, subtract the current value of SQLERRD(3) from its previous value (stored in another variable). In the following example, you determine the number of rows returned by the most recent fetch:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_number (100)    INTEGER;
    emp_name (100)      CHARACTER (20) ;
EXEC SQL END DECLARE SECTION;

    rows_to_fetch       INTEGER;
    rows_before         INTEGER;
    rows_this_time      INTEGER;

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT EMPNO, ENAME
    FROM EMP
    WHERE DEPTNO = 30;

EXEC SQL OPEN emp_cursor ;

EXEC SQL WHENEVER NOT FOUND CONTINUE;

-- initialize loop variables
set rows_to_fetch = 20; -- number of rows in each "batch"
set rows_before = 0;       -- previous value of sqlerrd (3 )
set rows_this_time = 20;

WHILE rows_this_time = rows_to_fetch
    LOOP
        EXEC SQL FOR :rows_to_fetch
            FETCH emp_cursor
            INTO :emp_number, :emp_name;
        set rows_this_time = sqlca.sqlerrd(3) - rows_before;
        set rows_before = sqlca.sqlerrd(3) ;
    ENDLOOP;
ENDWHILE;
...
```

SQLERRD(3) is also useful when an error occurs during an array operation. Processing stops at the row that caused the error, so SQLERRD(3) gives the number of rows processed successfully.

CHAPTER

9

USING DYNAMIC SQL

This chapter shows you how to use dynamic SQL, an advanced programming technique that adds flexibility and functionality to your applications. After weighing the advantages and disadvantages of dynamic SQL, you learn four methods-from simple to complex—for writing programs that accept and process SQL statements “on the fly” at run time. You learn the requirements and limitations of each method and how to choose the right method for a given job.

What Is Dynamic SQL?

Most database applications do a specific job. For example, a simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables. In this case, you know the makeup of the UPDATE statement at precompiled time. That is, you know which tables might be changed, the constraints defined for each table and column, which columns might be updated, and the datatype of each column.

However, some applications must accept (or build) and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different SELECT statements for the various reports it generates. In this case, the statement's makeup is unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called *dynamic* SQL statements.

Unlike static SQL statements, dynamic SQL statements are not embedded in your source program. Instead, they are stored in character strings input to or built by the program at run time. They can be entered interactively or read from a file.

Advantages and Disadvantages of Dynamic SQL

Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs. Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.

For example, your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, dynamic SQL lets you write highly flexible applications.

However, some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

When to Use Dynamic SQL

In practice, static SQL will meet nearly all your programming needs. Use dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one of the following items is unknown at precompiled time:

- text of the SQL statement (commands, clauses, and so on)
- the number of host variables
- the datatypes of host variables
- references to database objects such as columns, indexes, sequences, tables, usernames, and views

Requirements for Dynamic SQL Statements

To represent a dynamic SQL statement, a character string must contain the text of a valid SQL statement, but not contain the EXEC SQL clause, host-language delimiters or statement terminator, or any of the following embedded SQL commands:

- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- INCLUDE
- OPEN
- PREPARE
- WHENEVER

Inmost cases, the character string can contain *dummy* host variables. They hold places in the SQL statement for actual host variables. Because dummy host variables are just placeholders, you do not declare them and can name them anything you like. For example, ORACLE makes no distinction between the following two strings:

```
' DELETE FROM EMP WHERE MGR =  : mgr_number AND JOB = : job_title'
' DELETE FROM EMP WHERE MGR = : m AND JOB = : j '
```

How Dynamic SQL Statements Are Processed

Typically, an application program prompts the user for the text of a SQL statement and the values of host variables used in the statement. Then ORACLE *parses* the SQL statement. That is, ORACLE examines the SQL statement to make sure it follows syntax rules and refers to valid database objects. Parsing also involves checking database access rights, reserving needed resources, and finding the optimal access path.

Next, ORACLE *binds* the host variables to the SQL statement. That is, ORACLE gets the addresses of the host variables so that it can read or write their values.

Then ORACLE executes the SQL statement. That is, ORACLE does what the SQL statement requested, such as deleting rows from a table.

The SQL statement can be executed repeatedly using new values for the host variables.

Methods for Using Dynamic SQL

This section introduces four methods you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, then offers guidelines for choosing the right method. Later sections show you how to use the methods. Also, you can find sample host-language programs in your *Supplement to the ORACLE Precompilers Guide.*

The four methods are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods ***1*** and 2, and so on. However, each method is most useful for handling a certain kind of SQL statement, as the following table shows:

Method	***Kind of SQL Statement***
1	nonquery without input host variables
2	nonquery with known number of input host variables
3	query with known number of select-list items and input host variables
4	query with unknown number of select-list items or input host variables

Note: The term select-list *item* includes column names and expressions such as SAL* 1.10 and MAX(SAL).

Method 1

This method lets your program accept or build a dynamic SQL statement, then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

```
' DELETE FROM EMP WHERE DEPTNO = 20 '
' GRANT SELECT ON EMP TO scott '
```

With Method 1, the SQL statement is parsed every time it is executed.

This method lets your program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands. The SQL statement must not be a query. The number of placeholders for input host variables and the datatypes of the input host variables must be known at precompiled time. For example, the following host strings fall into this category:

```
' INSERT INTO EMP ( ENAME, JOB ) VALUES (: emp_name, :job-title )'
' DELETE FROM EMP WHERE EMPNO = : emp_number'
```

With Method 2, the SQL statement is parsed just once, but can be executed many times with different values for the host variables. SQL data definition statements such as CREATE and GRANT are executed when they are PREPAREd.

Method 3

This method lets your program accept or build a dynamic query, then process it using the PREPARE command with the DECLARE, OPEN FETCH, and CLOSE cursor coremands. The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables must be known at precompiled time. For example, the following host strings qualify

```
'SELECT DEPTNO, MIN ( SAL ) , MAX ( SAL ) FROM EMP GROUP BY DEPTNO'
' SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = : dept_number'
```

Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors (discussed in the section "Using Method 4" later in this chapter). The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

```
' INSERT INTO EMP (<unknown>) VALUES (<unknown>)'
' SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables.

Guidelines

With all four methods, you must store the dynamic SQL statement in a character string, which must be a host variable or quoted literal. When you store the SQL statement in the string, omit the keywords EXEC SQL and the statement terminator.

With Methods 2 and 3, the number of placeholders for input host variables and the datatypes of the input host variables must be known at precompiled time.

Each succeeding method imposes fewer constraints on your application, but is more difficult to code. As a rule, use the simplest method you can. However, if a dynamic SQL statement will be executed repeatedly by Method 1, use Method 2 instead to avoid reparsing for each execution.

Method 4 provides maximum flexibility, but requires complex coding and a full understanding of dynamic SQL concepts. In general, use Method 4 only if you cannot use Methods 1, 2, or 3.

The decision logic in Figure 9-1 will help you choose the right method.

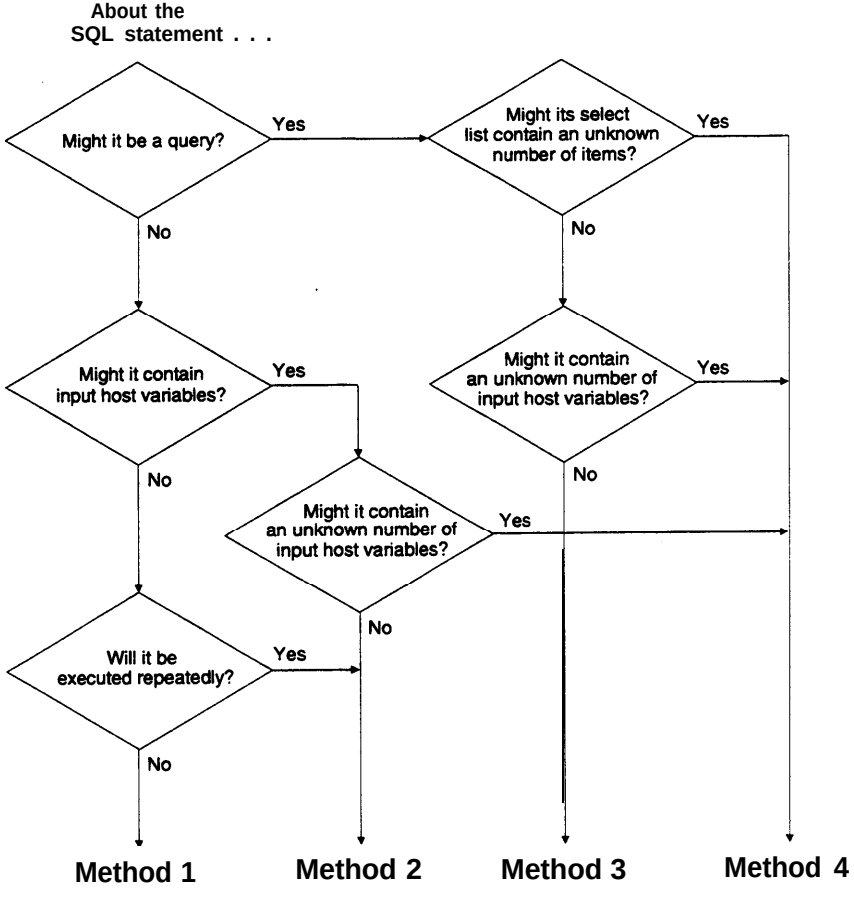
Avoiding Common Errors

If you use a character array to store the dynamic SQL statement, blank-pad the array before storing the SQL statement. That way, you clear extraneous characters. This is especially important when you reuse the array for different SQL statements. As a rule, always initialize (or reinitialize) the host string before storing the SQL statement.

Do not null-terminate the host string. ORACLE does not recognize the null terminator as an end-of-string sentinel. Instead, ORACLE treats it as part of the SQL statement.

If you use a VARCHAR variable to store the dynamic SQL statement, make sure the length of the VARCHAR is set (or reset) correctly before you execute the PREPARE or EXECUTE IMMEDIATE statement.

Figure 9-1
Choosing the Right Method



Using Method 1

The simplest kind of dynamic SQL statement results only in "success" or "failure" and uses no host variables. Some examples follow

```
' DELETE FROM table_name WHERE column_name = constant'
' CREATE TABLE table_name . . . '
' DROP INDEX index_name'
'UPDATE table_name SET column_name = constant'
'GRANT SELECT ON table_name TO username'
'REVOKE RESOURCE FROM username'
```

Method 1 parses, then immediately executes the SQL statement using the EXECUTE IMMEDIATE command. The command is followed by a character string (host variable or literal) containing the SQL statement to be executed, which cannot be a query.

The syntax of the EXECUTE IMMEDIATE statement follows:

```
EXEC SQL EXECUTE IMMEDIATE {  :host_string |string_literal };
```

In the following example, you use the host variable *sql_stmt* to store SQL statements input by the user:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    sql_stmt   CHARACTER (120) ;
EXEC SQL END DECLARE SECTION;
...
LOOP
    display 'Enter SQL statement: ';
    read sql_stmt;
    IF sql_stmt is empty THEN
        exit loop;
    ENDIF;
    -- sql_stmt now contains the text of a SQL statement
    EXEC SQL EXECUTE IMMEDIATE :sql_stmt;
ENDLOOP;
...
```

You can also use string literals, as the following example shows:

```
EXEC SQL EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER';
```

Because EXECUTE IMMEDIATE parses the input SQL statement before every execution, Method 1 is best for statements that are executed only once.

An Example

The following program prompts the user for a search condition to be used in the WHERE clause of an UPDATE statement, then executes the statement using Method 1:

```
EXEC SQL BEGIN DECLARE SECTION;
    username       CHARACTER (20) ;
    password       CHARACTER (20) ;
    update_stmt    CHARACTER (120) ;
EXEC SQL END DECLARE SECTION;

    search_cond    CHARACTER(40);

EXEC SQL INCLUDE SQLCA;

display 'Username? ';
read username;
display 'Password? ';
read password;

EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL CONNECT :username IDENTIFIED BY :password;

display 'Connected to ORACLE';

set update_stmt = 'UPDATE EMP SET COMM = 500 WHERE ';
display 'Enter a search condition for the following statement:';
display update_stmt;
read search_cond;
concatenate update_stmt, search_cond;

EXEC SQL EXECUTE IMMEDIATE :update_stmt;

EXEC SQL COMMIT WORK RELEASE;
exit program;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    display 'Processing error';
    exit program with an error;
```

Using Method 2

What Method 1 does in one step, Method 2 does in two. The dynamic SQL statement, which cannot be a query, is first PREPAREd (named and parsed), then EXECUTEd.

With Method 2, the SQL statement can contain placeholders for input host variables and indicator variables. You can PREPARE the SQL statement once, then EXECUTE it repeatedly using different values of the host variables. Furthermore, you need *not* rePREPARE the SQL statement after a COMMIT or ROLLBACK (unless you log off and reconnect).

Note that you can use EXECUTE for nonqueries with Method 4.

The syntax of the PREPARE statement follows:

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };
```

PREPARE parses the SQL statement and gives it a name.

The *statement_name* is an identifier used by the precompiled, *not* a host or program variable, and should not be declared in the Declare Section. It simply designates the PREPAREd statement you want to EXECUTE.

The syntax of the EXECUTE statement is

```
EXEC SQL EXECUTE statement_name [USING host_variable_list ] ;
```

where *host_variable_list* stands for the following syntax:

```
:host_variable1 [ : indicator1l [, host_variable2 [:indicator2] , . ..1
```

EXECUTE executes the parsed SQL statement, using the values supplied for each input host variable.

In the following example, the input SQL statement contains the placeholder *n:*

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    emp_number      INTEGER ;
    delete_stmt     CHARACTER(120)  ;
EXEC SQL END DECLARE SECTION;

    search_cond     CHARACTER (40 ) ;
...
```

```
set delete_stmt = 'DELETE FROM EMP WHERE EMPNO = :n AND ';

display 'Complete the following statement's search condition:';
display delete_stmt;
read search_cond;

concatenate delete_stmt, search_cond;

EXEC SQL PREPARE sql_stmt FROM :delete_stmt;

LOOP
    display 'Enter employee number: ';
    read emp_number;
    IF emp_number = 0 THEN
        exit loop;
    EXEC SQL EXECUTE sql_stmt USING :emp_number;
ENDLOOP;
...
```

With Method 2, you must know the data types of input host variables at precompile time. In the last example, *emp_number* was declared as type INTEGER. It could also have been declared as type REAL or CHARACTER, because ORACLE supports all these datatype conversions to NUMBER.

The USING Clause

When the SQL statement is EXECUTEd, input host variables in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

Every placeholder in the PREPAREd dynamic SQL statement must correspond to a host variable in the USING clause. So, if the same placeholder appears two or more times in the PREPAREd statement, each appearance must correspond to a host variable in the USING clause.

The names of the placeholders need not match the names of the host variables. However, the order of the placeholders in the PREPAREd dynamic SQL statement must match the order of corresponding host variables in the USING clause.

If one of the host variables in the USING clause is an array, all must be arrays.

To specify nulls, you can associate indicator variables with host variables in the USING clause. For more information, see the section "Using Indicator Variables" in Chapter 4.

An Example

The following program prompts the user for a search condition to be used in the WHERE clause of an UPDATE statement, then prepares and executes the statement using Method 2. Notice that the SET clause ‘ of the UPDATE statement contains a placeholder *(c)*.

```
EXEC SQL BEGIN DECLARE SECTION;
    username       CHARACTER (20) ;
    password       CHARACTER (20) ;
    commission     REAL ;
    update_stmt    CHARACTER (120) ;
EXEC SQL END DECLARE SECTION;

    search_cond    CHARACTER (40) ;

EXEC SQL INCLUDE SQLCA;

display ‘ Username? ‘ ;
read username;
display ‘ Password? ‘ ;
read password;

EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL CONNECT :username IDENTIFIED BY : password;

display ‘Connected to ORACLE’;

set update_stmt = ‘UPDATE EMP SET COMM = :c WHERE ':
display ‘Enter a search condition for the following statement:’;
display update_stmt;
read search_cond;
concatenate update_stmt, search_cond;

EXEC SQL PREPARE eql_stmt FROM :update_stmt;

display ‘Commission? ‘;
read commission;

EXEC SQL EXECUTE sql_stmt USING :commission;

EXEC SQL COMMIT WORK RELEASE;
exit program;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    display ‘Processing error’;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit program with an error;
```

Using Method 3

Method 3 is similar to Method 2 but combines the PREPARE statement with the statements needed to define and manipulate a cursor. This allows your program to accept and process queries. In fact, if the dynamic SQL statement is a query, you *must* use Method 3 or 4.

For Method 3, the number of columns in the query select list and the number of placeholders for input host variables must be known at precompiled time. However, the names of database objects such as tables and columns need not be specified until run time. Clauses that limit, group, and sort query results (such as WHERE, GROUP BY, and ORDER BY) can also be specified at run time.

With Method 3, you use the following sequence of embedded SQL statements:

```
PREPARE statement_name FROM {  :host_string |string_literal } ;
DECLARE cursor_name CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list ] ;
FETCH cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

Now let's look at what each statement does.

PREPARE

PREPARE parses the dynamic SQL statement and gives it a name. In the following example, PREPARE parses the query stored in the character string *select_stmt* and gives it the name *sql_stmt:*

```
set select_stmt =  ' SELECT MGR, JOB FROM EMP WHERE SAL < : salary' ;
EXEC SQL PREPARE sql_stmt FROM : select_stmt;
```

Commonly, the query WHERE clause is input from a terminal at run time or is generated by the application.

The identifier *sql_stmt* is *not* a host or program variable, but must be unique. It designates a particular dynamic SQL statement.

DECLARE

DECLARE defines a cursor by giving it a name and associating it with a specific query. Continuing our example, DECLARE defines a cursor named *emp_cursor* and associates it with *sql_stmt, as* follows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

The identifiers *sql_stmt* and *emp_cursor are not* host or program variables, but must be unique. If you declare two cursors using the same statement name, the precompiled considers the two cursor names synonymous.

For example, if you execute the statements

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
EXEC SQL DECLARE dept_cursor FOR sql_stmt;
```

when you OPEN *emp_cursor*, you will process the dynamic SQL statement stored in *delete_stmt*, not the one stored in *select_stmt*.

OPEN

OPEN allocates an ORACLE cursor, binds input host variables, and executes the query, identifying its active set. OPEN also positions the cursor on the first row in the active set and zeroes the rows-processed count kept by the third element of SQLERRD in the SQLCA. Input host variables in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

In our example, OPEN allocates *emp_cursor* and assigns the host variable salary to the WHERE clause, as follows:

```
EXEC SQL OPEN emp_cursor USING : salary;
```

FETCH

FETCH returns a row from the active set, assigns column values in the select list to corresponding host variables in the INTO clause, and advances the cursor to the next row. If there are no more rows, FETCH returns the "no data found" ORACLE error code to SQLCODE in the SQLCA.

In our example, FETCH returns a row from the active set and assigns the values of columns MGR and JOB to host variables *mgr_number* and *job_title*, as follows:

```
EXEC SQL FETCH emp_cursor INTO : mgr_number, : job_title;
```

CLOSE

CLOSE disables the cursor. Once you CLOSE a cursor, you can no longer FETCH from it.

In our example, CLOSE disables *emp_cursor*, as follows:

```
EXEC SQL CLOSE emp_cursor;
```

An Example

The following program prompts the user for a search condition to be used in the WHERE clause of a query, then prepares and executes the query using Method 3.

```
EXEC SQL BEGIN DECLARE SECTION;
    username       CHARACTER(20);
    password       CHARACTER(20);
    dept_number    INTEGER;
    emp_name       CHARACTER(10);
    salary         REAL ;
    select_stmt    CHARACTER(120);
EXEC SQL END DECLARE SECTION;

    search_cond    CHARACTER(40);

EXEC SQL INCLUDE SQLCA;

display 'Username? ';
read username;
display 'Password? ';
read password;

EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL CONNECT :username IDENTIFIED BY :password;

display 'Connected to ORACLE';

set select_stmt = 'SELECT ENAME,SAL FROM EMP WHERE ';
display 'Enter a search condition for the following statement:';
display select_stmt;
read search_cond;
concatenate select_stmt, search_cond;

EXEC SQL PREPARE sql_stmt FROM :select_stmt;

EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO no_more;

display 'Employee    Salary' ;
display ' -------- ------ ' ;

LOOP
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    display emp_name, salary;
ENDLOOP;

no_more:
    EXEC SQL CLOSE emp_cursor;
    EXEC SQL COMMIT WORK RELEASE;
    exit program;

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit program with an error;
```

Using Method 4

The implementation of Method 4 is very language-dependent. Therefore, this section only gives an overview. For details, see your *Supplement to the ORACLE Precompilers Guide.*

There is a kind of dynamic SQL statement that your program cannot process using Method 3. When the number of select-list items or placeholders for input host variables is unknown until run time, your program must use a descriptor. A *descriptor* is an area of memory used by your program and ORACLE to hold a complete description of the variables in a dynamic SQL statement.

Recall that for a multirow query, you FETCH selected column values INTO a list of declared output host variables. If the select list is unknown, the host-variable list cannot be established at precompiled time by the INTO clause. For example, you know the following query returns two column values:

```
SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = : dept_number;
```

However, if you let the user define the select list, you might not know how many column values the query will return.

Need for the SQLDA

To process this kind of dynamic query, your program must issue the DESCRIBE SELECT LIST command and declare a data structure called the SQL Descriptor Area (SQLDA). Because it holds descriptions of columns in the query select list, this structure is also called a *select descriptor.*

Likewise, if a dynamic SQL statement contains an unknown number of placeholders for input host variables, the host-variable list cannot be established at precompiled time by the USING clause.

To process the dynamic SQL statement, your program must issue the DESCRIBE VARIABLES command and declare another kind of SQLDA called a bind *descriptor* to hold descriptions of the placeholders for input host variables. (Input host variables are also called *bind variables.)*

If your program has more than one active SQL statement (it might have OPENed two or more cursors, for example), each statement must have its own SQLDA(s). However, non-concurrent cursors can reuse SQLDAs. There is no set limit on the number of SQLDAs in a program.

The DESCRIBE Statement

DESCRIBE initializes a descriptor to hold descriptions of select-list items or input host variables.

If you supply a select descriptor, the DESCRIBE SELECT LIST statement examines each select-list item in a PREPAREd dynamic query to determine its name, datatype, constraints, length, scale, and precision. It then stores this information in the select descriptor.

If you supply a bind descriptor, the DESCRIBE BIND VARIABLES statement examines each placeholder in a PREPAREd dynamic SQL statement to determine its name, length, and the datatype of its associated input host variable. It then stores this information in the bind descriptor for your use. For example, you might use placeholder names to prompt the user for the values of input host variables.

What Is a SQLDA?

A SQLDA is a host-program data structure that holds descriptions of select-list items or input host variables.

SQLDA variables are *not* defined in the Declare Section.

Though SQLDAs differ among host languages, a generic select SQLDA contains the following information about a query select list:

- maximum number of columns that can be DESCRIBEd
- actual number of columns found by DESCRIBE
- addresses of buffers to store column values
- lengths of column values
- datatypes of column values
- addresses of indicator-variable values
- addresses of buffers to store column names
- sizes of buffers to store column names
- current lengths of column names

A generic bind SQLDA contains the following information about the input host variables in a SQL statement

- maximum number of placeholders that can be DESCRIBEd
- actual number of placeholders found by DESCRIBE
- addresses of input host variables
- lengths of input host variables
- datatypes of input host variables
- addresses of indicator variables
- addresses of buffers to store placeholder names
- sizes of buffers to store placeholder names

- current lengths of placeholder names
- addresses of buffers to store indicator-variable names
- sizes of buffers to store indicator-variable names
- current lengths of indicator-variable names

To see the SQLDA structure and variable names in a particular host language, see your *Supplement to the ORACLE Precompilers Guide.*

Implementing Method 4

With Method 4, you generally use the following sequence of embedded SQL statements:

```
EXEC SQL PREPARE statement_name
    FROM { :host_string |string_ literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
    INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
    [USING DESCRIPTOR bind_descriptor_name ] ;
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
    INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
    USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

However, select and bind destiptors need not work in tandem. So, if the number of columns in a query select list is known, but the number of placeholders for input host variables is unknown, you can use the Method 4 OPEN statement with the following Method 3 FETCH statement

```
EXEC SQL FETCH emp_cursor INTO host_variable_list;
```

Conversely, if the number of placeholders for input host variables is known, but the number of columns in the select list is unknown, you can use the Method 3 OPEN statement

```
EXEC SQL OPEN cursor_name [USING host_variable_list] ;
```

with the Method 4 FETCH statement.

Note that EXECUTE can be used for nonqueries with Method 4.

To see how these statements allow your program to process dynamic SQL statements using descriptors, refer to your *Supplement to the ORACLE Precompilers Guide.*

Using the DECLARE STATEMENT Statement

With Methods 2,3, and 4, you might need to use the statement

```
EXEC SQL [AT db_name ] DECLARE statement_name STATEMENT;
```

where *db_name* and *statement_name* are identifiers used by the precompiled, *not* host or program variables.

DECLARE STATEMENT declares the name of a dynamic SQL statement so that the statement can be referenced by PREPARE, EXECUTE, DECLARE CURSOR, and DESCRIBE. It is required if you want to execute the dynamic SQL statement at a non-default database. An example using Method 2 follows:

```
EXEC SQL AT remote_db DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM : sql_string;
EXEC SQL EXECUTE sql_stmt;
```

In the example, *remote_db* tells ORACLE whereto EXECUTE the SQL statement.

With Methods 3 and 4, DECLARE STATEMENT is also required if the DECLARE CURSOR statement precedes the PREPARE statement, as shown in the following example:

```
EXEC SQL DECLARE sql_stmt STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM : sql_string;
```

The usual sequence of statements is

```
EXEC SQL PREPARE sql_stmt FROM : sqi_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

Using Host Arrays

The use of host arrays in static SQL and dynamic SQL is similar. For example, to use input host arrays with dynamic SQL Method 2, simply use the syntax

```
EXEC SQL EXECUTE Statement_name USING host_array_list;
```

where *host_array_list* contains one or more host arrays.

Similarly, to use input host arrays with Method 3, use the following syntax:

```
OPEN cursor_name USING host_array_list;
```

To use output host arrays with Method 3, use the following syntax:

```
FETCH cursor_name INTO host_array_list;
```

With Method 4, you must use the optional FOR clause to tell ORACLE the size of your input or output host array. To see how this is done, refer to your *Supplement to the ORACLE Precompilers Guide.*

Using PL/SQL

The ORACLE Precompilers treat a PL/SQL block like a single SQL statement. So, like a SQL statement, a PL/SQL block can be stored in a string host variable or literal. When you store the PL/SQL block in the string, omit the keywords EXEC SQL EXECUTE, the keyword END-EXEC, and the statement terminator.

However, there are two differences in the way the precompiled handles SQL and PL/SQL:

- The precompiled treats all PL/SQL host variables as *input* host variables whether they serve as input or output host variables (or both) inside the PL/SQL block.
- You cannot FETCH from a PL/SQL block because it might contain any number of SQL statements.

With Method 1

If the PL/SQL block contains no host variables, you can use Method 1 to EXECUTE the PL/SQL string in the usual way.

With Method 2

If the PL/SQL block contains a known number of input and output host variables, you can use Method 2 to PREPARE and EXECUTE the PL/SQL string in the usual way.

You must put *all* host variables in the USING clause. When the PL/SQL string is EXECUTEd, host variables in the USING clause replace corresponding placeholders in the PREPAREd string. Though the precompiled treats all PL/SQL host variables as input host variables, values are assigned correctly. Input (program) values are assigned to input host variables, and output (column) values are assigned to output host variables.

Every placeholder in the PREPAREd PL/SQL string must correspond to a host variable in the USING clause. So, if the same placeholder appears two or more times in the PREPAREd string, each appearance must correspond to a host variable in the USING clause.

With Method 3

Methods 2 and 3 are the same except that Method 3 allows FETCHing. Since you cannot FETCH from a PL/SQL block, just use Method 2 instead.

With Method 4

If the PL/SQL block contains an unknown number of input or output host variables, you must use Method 4.

To use Method 4, you setup one bind descriptor for all the input and output host variables. Executing DESCRIBE BIND VARIABLES stores information about input *and* output host variables in the bind descriptor. Because the precompiled treats all PL/SQL host variables as input host variables, executing DESCRIBE SELECT LIST has no effect.

The use of bind descriptors with Method 4 is detailed in your *Supplement to the ORACLE Precompilers Guide.*

Caution

Do not use ANSI-style comments (– . ..) in a PL/SQL block that will be processed dynamically because end-of-line characters are ignored. AS a result, ANSI-style comments extend to the end of the block, not just to the end of a line. Instead, use C-style comments (/* . . . */).

CHAPTER

10

WRITING SQL*FORMS USER EXITS

This chapter focuses on writing user exits for your SQL*Forms applications. You learn how host-language subroutines can do certain jobs more quickly and easily than SQL*Forms. The following topics are covered:

- common uses for user exits
- writing a user exit
- passing values between SQL*Forms and a user exit
- implementing a user exit
- calling a user exit
- guidelines for user exits

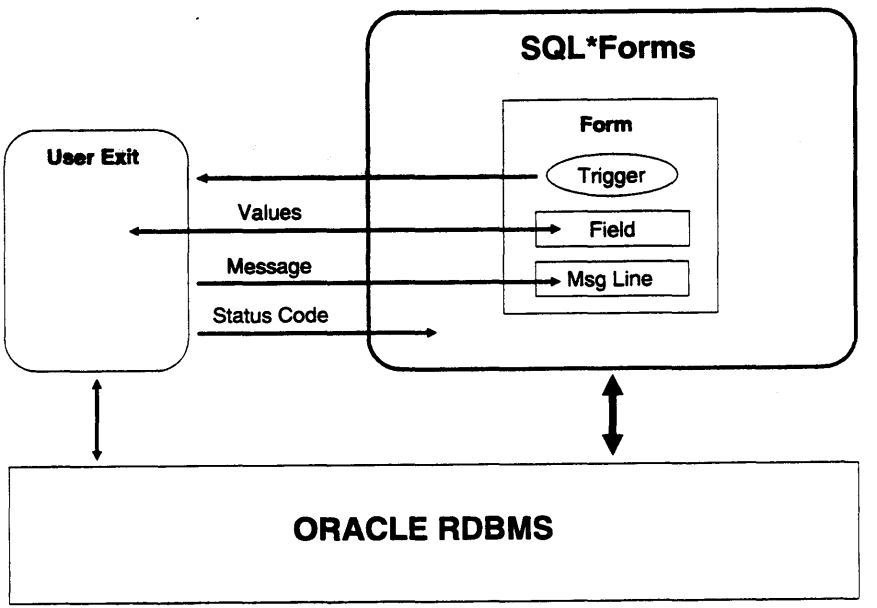
This chapter is supplemental. For more information about user exits, refer to the *SQL*Forms Designer's Reference* and the ORACLE installation or user's guide for your system.

What Isa User Exit?

A *user exit* is a host-language subroutine written by you and called by SQL*Forms to do special-purpose processing. You can embed SQL commands and PL/SQL blocks in your user exit, then precompiled it as you would a host program.

When called by a SQL*Forms trigger, the user exit runs, then returns a status code to SQL*Forms (refer to Figure 10-1). Your user exit can display messages on the SQL*Forms status line, get and put field *values*, manipulate ORACLE data, do high-speed computations and table lookups—even logon to different databases.

Figure 10-1
User Exit Called by SQL*Forms



Why Write a User Exit?

SQL*Forms Version 3 allows you to use PL/SQL blocks in triggers. So, in most cases, instead of calling a user exit, you can use the procedural power of PL/SQL. If the need arises, you can call user exits from a PL/SQL block with the USER_EXIT function.

User exits are harder to write and implement than SQL, PL/SQL, or SQL*Forms commands. So, you will probably use them only to do processing that is beyond the scope of SQL, PL/SQL, and SQL*Forms. Some common uses follow:

- operations more quickly or easily done in third generation languages like C and FORTRAN (numerical integration, for instance)
- controlling realtime devices or processes (issuing a sequence of instructions to a printer or graphics device, for example)
- data manipulations that need extended procedural capabilities (recursive sorting, for example)
- special file I/O operations

Developing a User Exit

This section outlines the way to develop a SQL*Forms user exit; later sections go into more detail.

To incorporate a user exit into a form, you take the following steps:

1. Write the user exit in a supported host language.
2. Precompiled the source code.
3. Compile the modified source code.
4. Use the GENXTB utility to add an entry to the LAP program table IAPXTB in the module IAPXIT. (LAP is the component of SQL*Forms that runs a form.)
5. Create a new IAP by linking the standard IAP modules, the modified IAPXIT module, and the new user exit module.
6. In the form, define a trigger to call the user exit.
7. Instruct operators to use the new IAP when running the form. This is unnecessary if the new LAP replaces the standard one. For details, see the ORACLE installation or user's guide for your system.

Writing a User Exit

You can use the following kinds of statements to write your SQL*Forms user exit

- host-language
- EXEC SQL
- EXEC ORACLE
- EXEC IAF GET
- EXEC IAF PUT

This section focuses on the EXEC IAF GET and PUT statements, which let you pass values between SQL*Forms and a user exit.

Requirements for Variables

The variables used in EXEC IAF statements must correspond to field names used in the form definition. If a field reference is ambiguous because you did not specify a block name, EXEC IAF defaults to the *context block*— the block that calls the user exit. An invalid or ambiguous reference to a form field generates an error.

Host variables must be named in the user exit Declare Section and must be prefixed with a colon (:) in EXEC IAF statements.

Note: Indicator variables are not allowed in EXEC IAF GET and PUT statements.

The IAF GET Statement

This statement allows your user exit to "get" values from fields on a form and assign them to host variables. The user exit can then use the values in calculations, data manipulations, updates, and so on. The syntax of the GET statement follows:

```
EXEC IAF GET field_name1, field_name2, . . .
    INTO :host_variable1, :host_variable2, ...;
```

where *field_name* can be any of the following SQL*Forms variables:

- field
- block.field
- system variable
- global variable
- host variable (prefixed with a colon) containing the value of a field, block.field, system variable, or global variable

If *field_name* is not qualified, the field must be in the context block.

Using IAF GET

The following example shows how a user exit GETs a field value and assigns it to a host variable

```
EXEC IAF GET employee.job INTO : new_job;
```

All field values are character strings. If it can, GET converts a field value to the datatype of the corresponding host variable. If an illegal or unsupported datatype conversion is attempted, an error is generated.

In the last example, a constant is used to specify *block.field.* You can also use a host string to specify block and field names, as follows:

```
set blkfld = 'employee. job' ;
EXEC IAF GET : blkfld INTO : new_job;
```

Unless the field is in the context block, the host string must contain the full *block.field* reference with intervening period. For example, the following usage is *invalid:*

```
set blk = ' employee' ;
set fld = 'job';
EXEC IAF GET :blk. : fld INTO :new_job;
```

You can mix explicit and stored field names in a GET statement field list, but not in a single field reference. For example, the following usage is *invalid:*

```
set fld = 'job';
EXEC IAF GET employee. : fld INTO : new_job;
```

The IAF PUT Statement

This statement allows your user exit to "put" the values of constants and host variables into fields on a form. Thus, the user exit can display on the SQL*Forms screen any value or message you like. The syntax of the PUT statement follows:

```
EXEC IAF PUT field_name1, field_name2, . . .
    VALUES (:host_variable1, :host-variable2, . . . ) ;
```

where field_name can be any of the following SQL*Forms variables:

- field
- block.field
- system variable
- global variable
- host variable (prefixed with a colon) containing the value of a field, block.field, system variable, or global variable

Using IAF PUT

The following example shows how a user exit PUTS the values of a numeric constant, string constant, and host variable into fields on a form:

```
EXEC IAF PUT employee. number, employee. name, employee. job
    VALUES (7934, 'MILLER', :new_job) ;
```

Like GET, PUT lets you use a host string to specify block and field names, as follows:

```
set blkfld = 'employee. job' ;
EXEC IAF PUT : blkfld VALUES (: new_job) ;
```

On character-mode terminals, a value PUT into a field is displayed when the user exit returns, rather than when the assignment is made, provided the field is on the current display page. On block-mode terminals, the value is displayed the next time a field is read from the device.

If a user exit changes the value of a field several times, only the last change takes effect.

Calling a User Exit

You call a user exit from a SQL*Forms trigger using a packaged procedure named USER_EXIT (supplied with SQL*Forms). The syntax you use is

```
USER_EXIT (user_exit_string [, error_string] ) ;
```

where *user_exit_string* contains the name of the user exit plus optional parameters and *error_string* contains an error message issued by SQL*Forms if the user exit fails. For example, the following trigger command calls a user exit named LOOKUP:

```
USER_EXIT ( ' LOOKUP ' ) ;
```

Notice that the user exit string is enclosed by single (not double) quotes.

Passing Parameters to a User Exit

When you call a user exit, SQL*Forms passes it the following parameters automatically:

Command Line	Is the user exit string.
Command Line Length	Is the length (in characters) of the user exit string.
Error Message	Is the error string (failure message) if one is defined.
Error Message Length	Is the length of the error string.
In-Query	Isa Boolean value indicating whether the exit was called in normal or query mode.

However, the user exit string allows you to pass additional parameters to the user exit. For example, the following trigger coremand passes two parameters and an error message to the user exit LOOKUP:

```
USER_EXIT (‘LOOKUP 2025 A’ , ‘Lookup failed’ ) ;
```

You can use this feature to pass field names to the user exit, as the following example shows:

```
USER _EXIT ('CONCAT firstname, lastname, address’ ) ;
```

However, it is up to the user exit, not SQL*Forms, to parse the user exit string.

Returning Values to a Form

When a user exit returns control to SQL*Forms, it must also return a code indicating whether it succeeded, failed, or suffered a fatal error. The return code is an integer constant defined by SQL*Forms (see the next section). The three results have the following meanings:

success	The user exit encountered no errors. SQL*Forms proceeds to the success label or the next step, unless the Reverse Return Code switch is set by the calling trigger step.

failure	The user exit detected an error, such as an invalid value in a field. An optional message passed by the exit appears on the message line at the bottom of the SQL*Forms screen and on the Display Error screen. SQL*Forms responds as it does to a SQL statement that affects no rows.
fatal error	The user exit detected a condition that makes further processing impossible, such as an execution error in a SQL statement. An optional error message passed by the exit appears on the SQL*Forms Display Error screen. SQL*Forms responds as it does to a fatal SQL error.

If a user exit changes the value of a field, then returns a *failure* or *fatal error* code, SQL*Forms does *not* discard the change. Nor does SQL*Forms discard changes when the Reverse Return Code switch is set and a success code is returned.

The IAP Constants

SQL*Forms defines three symbolic constants for use as return codes. Depending on the host language, they are prefixed with IAP or SQL. For example, they might be IAPSUCC, IAPFAIL, and IAPFTL.

Using the SQLIEM Function

By calling the function SQLIEM, your user exit can specify an error message that SQL*Forms will display on the message line if the trigger step fails or on the Display Error screen if the step causes a fatal error. The specified message replaces any message defined for the step. The syntax of the SQLIEM function call is

```
SQLIEM ( error_message, message_ length ) ;
```

where *error_message* and *message_length* are character and integer variables, respectively. The ORACLE Precompilers generate the appropriate external function declaration for you. You pass both parameters by reference; that is, you pass their addresses, not their values.

SQLIEM is a SQL*Forms function; it cannot be called from other ORACLE tools such as SQL*ReportWriter.

Using WHENEVER

You can use the WHENEVER statement in an exit to detect invalid datatype conversions (SQLERROR), truncated values PUT into form fields (SQLWARNING), and queries that return no rows (NOT FOUND).

An Example

The following example shows how a typical user exit is coded. Notice that, like a host program, the user exit has a Declare Section and a SQLCA.

```
-- subroutine  MYEXIT

EXEC SQL BEGIN DECLARE SECTION;
    field1       CHARACTER (20);
    field2       CHARACTER (20);
    value1       CHARACTER (20);
    value2       CHARACTER (20);
    result_val   CHARACTER (20);
EXEC SQL END DECLARE SECTION;

    errmsg    CHARACTER (80);
    errlen    INTEGER;

EXEC SQL INCLUDE SQLCA;

EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

-- get field values from form
EXEC IAF GET :field1, :field2 INTO :value1, :value2;

-- manipulate values to obtain result_val

-- put result_val into form field result
EXEC IAF PUT result VALUES (:result_val);
return(IAPSUCC);    -- trigger step succeeded

sqlerror:
    set errmsg = CONCAT('MYEXIT: ', sqlca.sqlerrm.sqlerrmc)  ;
    set errlen = LENGTH(errmsg) ;
    sqliem(errmsg, errlen);  -- pass error message to SQL*Forms
    return(IAPFAIL) ;  -- trigger step failed
```

Precompiling and Compiling a User Exit

User exits are precompiled like stand-alone host programs. Refer to Chapter 11, "Running the ORACLE Precompilers."

For instructions on compiling a user exit, see the ORACLE installation or user's guide for your system.

Using the GENXTB Utility

The LAP program table IAPXTB in module IAPXIT contains an entry for each user exit linked into IAP. IAPXTB tells IAP the name, location, and host language of each user exit. When you add anew user exit to IAP, you must add a corresponding entry to IAPXTB.

IAPXTB is derived from a database table, also named IAPXTB. You can modify the database table by running the GENXTB form on the operating system command line, as follows:

```
RUNFORM  GENXTB  username/password
```

A form is displayed that allows you to enter the following information for each user exit you define:

- exit name (see the section "Guidelines for User Exits" later in this chapter)
- host-language code (C, COB, FOR, PAS, or PLI)
- date created
- date last modified
- comments

After modifying the IAPXTB database table, use the GENXTB utility to read the table and create an Asembler or C source program that defines the module IAPXIT and the IAPXTB program table it contains. The source language used depends on your operating system. The syntax you use to run the GENXTB utility is

```
GENXTB username/password out file
```

where *outfile* is the name you give the Assembler or C source program that GENXTB creates.

Linking a User Exit into SQL*Forms

Before running a form that calls a user exit, you must link the user exit into IAP, the SQL*Forms component that runs a form. The user exit can be linked into your standard version of IAP or into a special version for those forms that call the exit.

To produce a new executable copy of IAP, link your user exit object module, the standard IAP modules, the IAPXIT module, and any modules needed from the ORACLE and host-language link libraries.

The details of linking are system-dependent. Check the ORACLE installation or user's guide for your system.

Guidelines

The guidelines in this section will help you avoid some common pitfalls.

Naming the Exit

The name of your user exit cannot be an ORACLE reserved word. Also avoid using names that conflict with the names of SQL*Forms commands, function codes, and externally defined names used by SQL*Forms.

SQL*Forms converts the name of a user exit to upper case before searching for the exit. Therefore, the exit name must be in upper case in your source code if your host language is case-sensitive.

The name of the user exit entry point in the source code becomes the name of the user exit itself. The exit name must be a valid filename for your host language and operating system.

Connecting to ORACLE

User exits communicate with ORACLE via the connection made by SQL*Forms. However, a user exit can establish additional connections to any database via SQL*Net. For more information, see the section "Concurrent Logons" in Chapter 3.

Issuing I/O Calls

SQL*Forms I/O routines might conflict with host-language printer I/O routines. If they do, your user exit will be unable to issue printer I/O calls. (This restriction does not apply to user exits written in C.) File I/O is supported but screen I/O is not.

Using Host Variables

Restrictions on the use of host variables in a stand-alone program also apply to user exits. Host variables must be named in the user exit Declare Section and must be prefixed with a colon in EXEC SQL and EXEC IAF statements. However, the use of host arrays is not allowed in EXEC IAF statements.

Updating Tables

Generally, a user exit should not UPDATE database tables associated with a form For example, suppose an operator updates a record in the SQL*Forms work space, then a user exit UPDATEs the corresponding row in the associated database table. When the transaction is COMMITted, the record in the SQL*Forms workspace is applied to the table, overwriting the user exit UPDATE.

Issuing Commands

Avoid issuing a COMMIT or ROLLBACK command from your user exit because ORACLE will commit or roll back work begun by the SQL*Forms operator, not just work done by the user exit. Instead, issue the COMMIT or ROLLBACK from the SQL*Forms trigger. This also applies to data definition commands (such as ALTER, CREATE, and GRANT) because they issue an implicit COMMIT before and after executing.

CHAPTER

11

RUNNING THE ORACLE PRECOMPILERS

This chapter details the requirements for running the ORACLE Precompilers. You learn what occurs during precompilation, how to issue the precompile command, how to specify the many useful precompiled options, and how to do conditional and separate precompilations.

What Occurs during Precompilation?

During precompilation, the ORACLE Precompilers generate host-language code sequences that replace the SQL statements embedded in your host program. A precompiled can issue error messages, which are listed in Appendix D.

Issuing the Precompiled Command

To run an ORACLE Precompiled, you issue one of the following language-specific commands:

Host Language	*Recompiler Command*
C	PROC
COBOL	PROCOB
FORTRAN	PROFOR
Pascal	PROPAS
PL/I	PROPLI

Required Arguments

The only argument required by the precompiled command is

```
INAME=filename
```

where *filename* identifies the precompiled input file. The precompiled assumes the standard input file extension (see table below). So, you need not use a file extension when specifying INAME unless the extension is nonstandard.

Host Language	*Standard File Extension*
C	PC
COBOL	PCO
FORTRAN	PFO
Pascal	PPA
PL/I	PPL

Revised HOST Option

With earlier ORACLE Precompilers, you used the HOST option to specify the host language of the input file or you used the standard input file extension when specifying INAME. The HOST parameter or standard file extension was needed because the precompiled accepted input files written in any licensed host language. Also, on some ports, the HOST option was used to specify different compilers.

However, the Version 1.5 ORACLE Precompilers are separate executable programs, each designed for a specific host language. As a result, the HOST option is no longer needed for specifying host languages, but it remains for port-specific and COBOL-specific purposes.

Optional Arguments

Many useful options are available at precompiled time. They let you control how resources are used, how errors are reported, how input and output is formatted, and how cursors are managed. To specify a precompiled option, you use the following syntax:

```
option_name=value
```

Leave no space around the equal sign because spaces delimit individual options.

Table 11-1 is a quick reference to the precompiled options. It summarizes the section “Using the Precompiled Options” later in this chapter.

Table 11-1
Precompiler Options

Syntax	Default	Specifies . . .
ASACC=YESINO	NO	carriage control for listing
CODE=ANSI_CIKR_C	KR_C	how C function prototypes are generated
COMMON_NAME=block_name *		name of FORTRAN common blocks
DBMS=NATIVEIV6IV7	NATIVE	version-specific behavior of ORACLE
DEFINE=symbol		symbol used in conditional precompilation
ERRORS=YESINO*	YES	whether errors are sent to the terminal
FIPS=YESINO	NO	whether ANSI/ISO extensions are flagged
FORMAT=ANSIITERMINAL	ANSI	format of COBOL or FORTRAN input line
HOLD_CURSOR=YESINO*	NO	how cursor cache handles SQL statements
HOST=CICOB74ICOBOLI		host language of input file
FORTRANIPASCALIPLI		name of input file
INAME=path and filename		directory path for INCLUDEd files
INCLUDE=path*	80	record length of input file
IRECLEN=integer	NO	whether C #line directives are generated
LINES=YESINO	QUOTE	delimiter for COBOL strings
LITDELIM=APOSTIQUOTE*		name of listing file
LNAME=path and filename	132	record length of listing file
LRECLEN=integer	LONG	type of listing
LTYPE=LONGISHORTINONE		maximum length of strings
MAXLITERAL=integer*	10	maximum number of cursors cached
MAXOPENCURSORS=integer*		
MODE=ANSIIISOIANSI14IISO14	ORACLE	compliance with ANSI/ISO standard
ANSI13IISO13IORACLE		name of ouput file
ONAME=path and filename	NO	whether the ORACA is used
ORACA=YESINO*	60	record length of output file
ORECLEN=integer	66	lines per page in listing
PAGELEN=integer	NO	how cursor cache handles SQL statements
RELEASE_CURSOR=YESINO*	YES	how SELECT errors are handled
SELECT_ERROR=YESINO*		
SQLCHECK=SEMANTICSIFULLI	SYNTAX	extent of syntactic and/or semantic checking
SYNTAXILIMITEDINONE*		valid ORACLE username and password
USERID=usemame/password	YES	cross-reference section in listing
XREF=YESINO*		
* Can be entered inline		

Another handy reference to the precompiled options is available online. To see the online display, just enter the precompiled command without options at your operating system prompt.

The display gives the name, syntax, default value, and purpose of each option. Options marked with an asterisk (*) can be specified inline as well as on the command line.

Scope of Options

The options specified for a given precompilation unit affect only that unit; they have no effect on other units. For example, if you specify HOLD_CURSOR and RELEASE_CURSOR for unit A but not for unit B, SQL statements in unit A run with the specified HOLD_CURsOR and RELEASE_CURSOR values, but SQL statements in unit B run with the default values.

Entering Options

All the precompiled options can be entered on the command line; many can also be entered inline.

On the Command Line

You enter precompiled options on the command line using the following syntax:

```
... [option_name=value1 [option_name=value] . . .
```

Separate each option with one or more spaces. For example, you might enter the following

```
... ERRORS=no  LTYPE=short
```

Inline

You enter options inline by coding EXEC ORACLE statements, using the following syntax:

```
EXEC ORACLE OPTION (option_name=value ) ;
```

For example, you might code the following

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES ) ;
```

An option entered inline overrides the same option entered on the command line.

Uses for EXEC ORACLE

The EXEC ORACLE feature is especially useful for changing option values during precompilation. For example, you might want to change the values of HOLD_CURSOR and RELEASE_CURSOR on a statement-by-statement basis. Appendix E shows you how to optimize runtime performance using Mine options.

Specifying options inline is also helpful if your operating system limits the number of characters you can enter on the command line. You can even store inline options in a separate file, then INCLUDE them at the appropriate place in your program.

Scope of EXEC ORACLE

An EXEC ORACLE statement stays in effect until superseded by another EXEC ORACLE statement specifying the same option. In the following example, HOLD_CURSOR=NO stays in effect until superseded by HOLD_CURSOR=YES:

```
EXEC SQL BEGIN DECLARE SECTION;
    emp_name     CHARACTER (20);
    emp_number   INTEGER;
    salary       REAL;
    dept_number  INTEGER:
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER NOT FOUND GOTO no_more;

EXEC ORACLE OPTION ( HOLD_CURSOR=NO ) ;

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT EMPNO, DEPTNO FROM EMP;

EXEC SQL OPEN emp_cursor;

display‘ Employee Number Dept';
display '--------------- ----';

LOOP
    EXEC SQL FETCH emp_cursor INTO : emp_number, : dept_number;
    display  emp_number,  dept_number;
ENDLOOP;

no_more:
    EXEC SQL WHENEVER NOT FOUND CONTINUE;

    LOOP
        display ‘ Employee number? ‘ ;
        read emp_number;
        IF emp_number = 0 THEN
            exit  loop;
        EXEC ORACLE OPTION (HOLD_CURSOR=YES ) ;
        EXEC SQL SELECT ENAME, SAL
            INTO : emp_name, : salary
            FROM EMP
            WHERE EMPNO =  :emp_number;
        display  ‘Salary for ‘ , emp_name, ‘ is ', salary;
    ENDLOOP;
...
```

Using the Precompiled Options

This section is organized for easy reference. It lists the precompiler options alphabetically and gives each option's purpose, syntax, default value, and usage notes.

ASACC

Purpose Specifies whether the listing file follows the ASA convention of using the first column in each line for carriage control.

Syntax ASACC=YES | NO

Default NO

Usage Notes Can be entered only on the command line.

CODE

Purpose Specifies the format of C function prototypes generated by the Pro*C Precompiled. (A *function prototype* declares a function and the datatypes of its arguments.) The precompiled generates function prototypes for SQL library routines so that your C compiler can resolve external references. The CODE option lets you control the prototyping.

Syntax CODE=ANSI_C | KR_C

Default KR_C

Usage Notes Can be entered inline or on the command line.

ANSI C standard X3.159-1989 provides for function prototyping. When CODE=ANSI_C, the Pro*C Precompiled generates full function prototypes, which conform to the ANSI C standard. An example follows:

```
extern void sqlora (long *, void *) ;
```

The precompiled can also generate other ANSI-approved constructs such as the const type qualifier. But, before specifying CODE=ANSI_C, make sure you understand all the ANSI requirements, including the rules for argument promotion. Also make sure to invoke an ANSI-compliant C compiler by using the appropriate command-line switches, which are system-specific.

When CODE=KR_C (the default), the precompiler generates code compatible with the earlier C specified in Kernighan and Ritchie's book *The C Programming Language* (first edition). Consequently, the precompiled comments out the argument lists of generated function prototypes, as shown in the following example:

```
extern void sqlora ( /*_ long *, void * _*/) ;
```

So, specify CODE=KR_C if your C compiler is not ANSI-compliant.

COMMON_NAME

Purpose Specifies a prefix used to name internal FORTRAN COMMON blocks. Your host program does not access the COMMON blocks directly. But, they allow two or more program units in the same precompilation unit to contain SQL statements.

Syntax COMMON_NAME=block_name

Default First 5 characters in name of input file

Usage Notes Can be entered inline or on the command line.

The Pro*FORTRAN Precompiled uses a special program unit called a *block data subprogram* to establish COMMON blocks for all the SQL variables in an input file. The block data subprogram defines two COMMON blocks, one for CHARACTER variables, the other for non-CHARACTER variables.

To name the COMMON blocks, the precompiled uses the name of the input file and the suffixes C and I. At most, the first 5 characters of the filename are used. For example, if the name of the input file is ACCTSPAY, the precompiled names the COMMON blocks ACCTSC and ACCTSI.

However, the precompiled can give COMMON blocks defined in different output files the same name, as the following schematic shows

```
ACCTSPAY.PFO.==> ACCTSC, ACCTSI in ACCTSPAY.FOR
ACCTSREC.PFO.==> ACCTSC, ACCTSI in ACCTSREC.FOR
```

If you were to link ACCTSPAY and ACCTSREC into an executable program, the linker would see two, not four, COMMON blocks.

To solve the problem, you can rename the input files. Or, you can override the default COMMON block names by specifying COMMON_NAME in-line or on the command line. For example, if you specify COMMON_NAME=PAY, the precompiled names its COMMON blocks PAYC and PAYI. At most, the first 5 characters in *block_name* are used.

If you specify COMMON_NAME inline, its EXEC ORACLE OPTION statement must precede the host-language PROGRAM, SUBROUTINE, or FUNCTION statement.

You might want to override the default COMMON block names if they conflict with your user-defined COMMON block names. However, it is better programming practice to rename the user-defined COMMON blocks.

DBMS

Purpose Specifies whether ORACLE follows the semantic and syntactic rules of ORACLE Version 6, ORACLE7, or the native version of ORACLE (that is, the version resident on your system, which must be Version 6 or later).

Syntax DBMS=NATIVE | V6 | V7

Default NATIVE

Usage Notes Can be entered only on the command line.

The DBMS option lets you control the version-specific behavior of ORACLE. When DBMS=NATIVE (the default), ORACLE follows the semantic and syntactic rules of the native version of ORACLE. However, not knowing the version, the precompiled cannot check for incompatibility errors such as specifying DBMS=V6 with MODE=ANSI.

When DBMS={V6 | V7}, ORACLE follows the rules of ORACLE Version 6 or ORACLE7, respectively. A summary of the differences between DBMS=V6 and DBMS=V7 follows:

- When DBMS=V6, ORACLE treats string literals like variable-length character values. However, when DBMS=V7, ORACLE treats string literals like fixed-length character values, and CHAR semantics change slightly to comply with the current ANSI/ISO SQL standard. For more information, see the section “VARCHAR2 versus CHAR” in Chapter 3.
- When DBMS=V6, ORACLE treats local CHAR variables in a PL/SQL block like variable-length character values. When DBMS=V7, however, ORACLE treats the CHAR variables like ANSI-compliant, fixed-length character values.
- When DBMS=V6, ORACLE treats the return value of the function USER like a variable-length character value. However, when DBMS=V7, ORACLE treats the return value of USER like an ANSI-compliant, fixed-length character value.

- When DBMS=V6, if you process a multirow query that calls a SQL group function such as AVG or COUNT, the function is called at OPEN time. When DBMS=V7, however, the function is called at FETCH time. At OPEN time or FETCH time, if the function call fails, ORACLE issues an error message immediately. Thus, the DBMS value affects error reporting slightly.
- When DBMS=V6 and RELEASE_CURSOR=YES, after ORACLE executes a SQL statement and its cursor is closed, its parsed representation is lost. So, before the SQL statement can be reexecuted, it must be reparsed. However, when DBMS=V7 and RELEASE_CURSOR=YES, the reparse might require no processing because ORACLE7 caches the parsed representations of SQL statements and PL/SQL blocks in its *Shared SQL Cache.* Even if its cursor is closed, the parsed representation remains available until it is aged out of the **cache.**
- When DBMS=V6, RELEASE_CURSOR=NO, and HOLD_CURSOR=YES, after ORACLE executes a SQL statement, its parsed representation remains available. However, when DBMS=V7, RELEASE_CURSOR=NO, and HOLD_CURSOR=YES, the parsed representation remains available only until it is aged out of the Shared SQL Cache. Normally, this is not a problem, but you might get unexpected results if the definition of a referenced object changes before the SQL statement is reparsed.

As the following table shows, some DBMS and MODE option values are incompatible or unrecommended:

DBMS	*MODE*	*Compatible?*	*Recommended?*
V6	ANSI	No	—
V6	ANSI14	Yes	No
V6	ANSI13	Yes	No
V6	ORACLE	Yes	Yes
V7	ANSI	Yes	Yes
V7	ANSI14	No	—
V7	ANSI13	No	—
V7	ORACLE	Yes	Yes

DEFINE

Purpose Specifies a user-defined symbol that is used to include or exclude portions of source code during a conditional precompilation. For more information, see the section "Doing Conditional Precompilations" later in this chapter.

Syntax DEFINE=symbol

Default None

Usage Notes Can be entered inline or on the command line.

If you enter DEFINE inline, the EXEC ORACLE statement takes the following form

```
EXEC ORACLE DEFINE Symbol ;
```

ERRORS

Purpose Specifies whether precompiled error messages are sent to the terminal and listing file or only to the listing file.

Syntax ERRORS=YES | NO

Default YES

Usage Notes Can be entered inline or on the command line.

When ERRORS=YES, error messages are sent to the terminal and listing file.

When ERRORS=NO, error messages are sent only to the listing file.

FIPS

Purpose Specifies whether extensions to ANSI SQL are flagged (by the FIPS Flagger). An extension is any SQL element that violates ANSI format or syntax rules, except privilege enforcement rules.

Syntax FIPS=YES | NO

Default NO

Usage Notes Can be entered inline or on the command line.

When FIPS=YES, the FIPS Flagger issues warning (not error) messages if you use an ORACLE extension to ANSI SQL or use an ANSI SQL feature in a nonconforming manner. The following extensions to ANSI SQL are flagged at precompiled time:

- array interface including the FOR clause
- SQLCA, ORACA, and SQLDA data structures
- dynamic SQL including the DESCRIBE statement
- embedded PL/SQL blocks
- automatic datatype conversion
- DATE, COMP-3 (Pro*COBOL only), NUMBER, RAW, LONGRAW, VARRAW, ROWID, and VARCHAR datatypes
- pointer host variables (Pro*C and Pro*Pascal only)
- ORACLE OPTION statement for specifying runtime options
- IAF statements in user exits
- CONNECT statement
- TYPE and VAR datatype equivalencing statements
- AT <db_name> clause
- DECLARE...DATABASE, ...STATEMENT, and ...TABLE statements
- SQLWARNING condition in WHENEVER statement
- DO, DO BREAK (Pro*C only), and STOP actions in WHENEVER statement
- COMMENT and FORCE TRANSACTION clauses in COMMIT statement
- FORCE TRANSACTION and TO SAVEPOINT clauses in ROLLBACK statement
- RELEASE parameter in COMMIT and ROLLBACK statements
- optional colon-prefixing of WHENEVER...GOTO labels and of host variables in the INTO clause

FORMAT

Purpose Specifies the format of COBOL or FORTRAN input lines.

Syntax FORMAT=ANSI | TERMINAL

Default ANSI

Usage Notes Can be entered only on the command line.

The format of input lines is system-dependent. Check the ORACLE installation or user's guide for your system.

When FORMAT=ANSI, the format of input lines conforms as much as possible to the current ANSI standard.

HOLD_CURSOR

Purpose Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

Syntax HOLD_CURSOR=YES | NO

Default NO

Usage Notes Can be entered inline or on the command line.

You can use HOLD_CURSOR to improve the performance of your program. For more information, refer to Appendix E.

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an ORACLE private SQL area, which stores information needed to process the statement. HOLD_CURSOR controls what happens to the link between the cursor and cursor cache.

When HOLD_CURSOR=NO, after ORACLE executes the SQL statement and the cursor is closed, the precompiled marks the link as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks.

When HOLD_CURSOR=YES, the link is maintained; the precompiled does not reuse it. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an ORACLE private SQL area.

For inline use with implicit cursors, set HOLD_CURSOR before executing the SQL statement. For inline use with explicit cursors, set HOLD_CURSOR before OPENing the cursor.

Note: RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For a table showing how these two options interact, refer to Appendix E.

HOST

Purpose Specifies the host language of the input file.

Syntax HOST=C | COB74 | COBOL | FORTRAN | PASCAL | PLI

Default Language-dependent

Usage Notes Can be entered only on the command line.

If you use a non-standard input file extension when specifying INAME, you must also specify HOST. Also, on some ports, HOST is needed to specify a compiler.

COB74 refers to the 1974 version of ANSI-approved COBOL.

INAME

Purpose Specifies the name of the input file.

Syntax INAME=path and filename

Default None

Usage Notes Can be entered only on the command line.

If you use a non-standard input file extension when specifying INAME, you must also specify HOST.

INCLUDE

Purpose Specifies a directory path for EXEC SQL INCLUDE files. It only applies to operating systems that use directories.

Syntax INCLUDE=path

Default Current directory

Usage Notes Can be entered inline or on the command line.

Typically, you use INCLUDE to specify a directory path for the SQLCA and ORACA files. The precompiled searches first in the current directory, then in the directory specified by INCLUDE, and finally in a directory for standard INCLUDE files. Hence, you need not specify a directory path for standard files such as the SQLCA and ORACA.

You must still use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. You can specify more than one path on the command line, as follows:

```
... INCLUDE=path1 INCLUDE=path2 . . .
```

The precompiled searches first in the current directory, then in the directory named by *path1*, then in the directory named by *path2*, and finally in the directory for standard INCLUDE files.

Remember, the precompiled looks for a file in the current directory first-even if you specify a directory path. So, if the file you want to INCLUDE resides in another directory, make sure no file with the same name resides in the current directory.

The syntax for specifying a directory path is system-specific. Check the ORACLE installation or user's guide for your system.

IRECLEN

Purpose Specifies the record length of the input file.

Syntax IRECLEN=integer

Default 80

Usage Notes Can be entered only on the command line.

The value you specify for IRECLEN should not exceed the value of ORECLEN. The maximum value allowed is system-dependent.

LINES

Purpose Specifies whether the Pro*C Precompiled adds #line preprocessor directives to its output file.

Syntax LINES=YES | NO

Default NO

Usage Notes Can be entered only on the command line.

The LINES option helps with debugging. When LINES=YES, the Pro*C Precompiled adds #line preprocessor directives to its output file.

Normally, your C compiler increments its line count after each input line is processed. The #line directives force the compiler to reset its input line counter so that lines of precompiled-generated code are not counted. Moreover, when the name of the input file changes, the next #line directive specifies the new filename.

The C compiler uses the line numbers and filenames to show the location of errors. Thus, error messages issued by the compiler always refer to your original source files, not the modified source file.

When LINES=NO (the default), the precompiled adds no #line directives to its output file.

LITDELIM

Purpose Specifies the delimiter for COBOL string literals.

Syntax LITDELIM=APOST | QUOTE

Default QUOTE

Usage Notes Can be entered inline or on the command line.

When LITDELIM=APOST, an apostrophe is used. When LITDELIM=QUOTE, a quotation mark is used.

LNAME

Purpose Specifies anon-default name for the listing file.

Syntax LNAME=path and filename

Default *input*.LIS, where *input* is the base name of the input file.

Usage Notes Can be entered only on the command line.

By default, the listing file is written to the current directory.

LRECLEN

Purpose Specifies the record length of the listing file.

Syntax LRECLEN=integer

Default 132

Usage Notes Can be entered only on the command line.

The value of LRECLEN can range from 80 through 255. If you specify a value below the range, 80 is used instead. If you specify a value above the range, 255 is used instead. LRECLEN should exceed IRECLEN by at least 8 to allow for the insertion of line numbers.

LTYPE

Purpose Specifies the listing type.

Syntax LTYPE=LONG | SHORT | NONE

Default LONG

Usage Notes Can be entered only on the command line.

When LTYPE=LONG, input lines appear in the listing file. When LTYPE=SHORT, input lines do *not* appear in the listing file. When LTYPE=NONE, no listing file is created.

MAXLITERAL

Purpose Specifies the maximum length of string literals generated by the precompiled so that compiler limits are not exceeded. For example, if your compiler cannot handle string literals longer than 256 characters, you can specify MAXLITERAL=256 on the command line.

Syntax MAXLITERAL=integer

Default Language-dependent

Language	*Default Value*
C	1000
COBOL	256
FORTRAN	1000
Pascal	1000
PLI	256

Usage Notes Can be entered Mine or on the command line.

The maximum value of MAXLITERAL is compiler-dependent. The default value is language-dependent, but you might have to specify a lower value. For example, the default value for C is 1000. However, some C compilers cannot handle string literals longer than 512 characters, so you would specify MAXLITERAL=512.

Strings that exceed the length specified by MAXLITERAL are divided during precompilation, then recombined (concatenated) at run time.

You can enter MAXLITERAL inline, as the following example shows:

```
EXEC ORACLE OPTION (MAXLITERAL=512 ) ;
```

Your program can set MAXLITERAL just once, and the EXEC ORACLE statement must precede the first EXEC SQL statement. Otherwise, the precompiled issues a warning message, ignores the extra or misplaced EXEC ORACLE statement, and continues processing.

With previous ORACLE Precompilers, your program could set MAXLITERAL more than once, although there was no reason to do so. It is unlikely that your old programs set MAXLITERAL more than once, but you might need to relocate the EXEC ORACLE statement.

MAXOPENCURSORS

Purpose Specifies the number of concurrently open cursors that the precompiled tries to keep cached.

Syntax MAXOPENCURSORS=integer

Default 10

Usage Notes Can be entered inline or on the command line.

You can use MAXOPENCURSORS to improve the performance of your program. For more information, see Appendix E.

The maximum number of open cursors per user process (which defaults to 50 but can range from 5 to 255) is set by the ORACLE initialization parameter OPEN_CURSORS. You can override this parameter by using MAXOPENCURSORS to specify a lower (but not higher) value. To avoid a "maximum open cursors exceeded" ORACLE error, MAXOPENCURSORS must be lower than OPEN_CURSORS by at least 6.

MAXOPENCURSORS specifies the *initial* size of the cursor cache. If a new cursor is needed and there are no free cache entries, ORACLE tries to reuse an entry. Its success depends on the values of HOLD_CURSOR and RELEASE_CURSOR and, for explicit cursors, on the status of the cursor itself. ORACLE allocates an additional cache entry if it cannot find one to reuse. If necessary, ORACLE keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN_CURSORS.

As your program's need for concurrently open cursors grows, you might want to respecify MAXOPENCURSORS to match the need. A value of 45 to 50 is not uncommon, but remember that each cursor requires another private SQL area in the user process memory space. The default value of 10 is adequate for most programs.

MODE

Purpose Specifies whether your program observes ORACLE practices or complies with the current ANSI SQL standard.

Syntax MODE=ANSI | ISO | ANSI14 | ISO14 | ANSI13 | ISO13 | ORACLE

Default ORACLE

Usage Notes Can be entered only on the command line.

The following pairs of MODE values are equivalent ANSI and ISO, ANSI14 and ISO14, ANSI13 and ISO13.

When MODE=ORACLE (the default), your embedded SQL program observes ORACLE practices. When MODE= {ANSI14 | ANSI13}, your program complies closely with the current ANSI SQL standard. When MODE=ANSI, your program complies fully with the ANSI standard and the following changes go into effect

- CHAR column values, USER pseudocolumn values, character host values, and quoted literals are treated like ANSI fixed-length character strings. Also, ANSI-compliant blank-padding semantics are used when you assign, compare, INSERT, UPDATE, SELECT, or FETCH such values. (When MODE= {ANSI14 | ANSI13 | ORACLE}, such values are treated like VARCHAR2 variable-length character strings and non-blank-padding semantics are used.)
- Issuing a COMMIT or ROLLBACK closes all explicit cursors. (When MODE={ANSI13 | ORACLE}, COMMIT or ROLLBACK closes only cursors referenced in a CURRENT OF clause.)
- You cannot OPEN an already open cursor or CLOSE an already closed cursor. (When MODE=ORACLE, you can reOPEN an open cursor to avoid reparking.)
- You cannot SELECT or FETCH nulls into a host variable not associated with an indicator variable. (When MODE= {ANSI13 | ORACLE}, you need not supply an indicator variable.)
- You must declare a 4-byte integer variable named SQLCODE (SQLCOD in FORTRAN) inside or outside the Declare Section.
- Declaring the SQLCA is optional. You need not hardcode the SQLCA or copy it into your program with the INCLUDE statement. (When MODE= {ANSI13 | ORACLE}, declaring the SQLCA is required.)

- No error message is issued if ORACLE assigns a truncated column value to an output host variable.
- The "no data found" ORACLE error code returned to SQLCODE becomes +100 instead of +1403. The error message text does not change.
- In SQL data manipulation statements, every host variable must be prefixed with a colon. (When MODE= {ANSI13 | ORACLE}, the colon prefix is optional in the INTO clause of a SELECT or FETCH statement.)

As the table below shows, some of these changes were not in effect under Versions 1.3 and 1.4 of the ORACLE Precompilers. So, if you want to trade some ANSI compatibility for more flexibility, specify MODE= {ANSI14 | ANSI13}. Note, however, that when MODE=ANSI14, array operations are *not* allowed. You can reference host arrays in a data manipulation statement only when MODE= {ANSI | ANSI13 | ORACLE}.

Change	***V1.3***	***V1.4***	***V1.5***
CHAR values treated like ANSI strings	no	no	yes
array operations allowed	yes	no	yes
COMMIT, ROLLBACK closes explicit cursors	no	yes	yes
illegal to OPEN an already open cursor	yes	yes	yes
must use indicator variable when fetching nulls	no	yes	yes
must declare SQLCODE	no	yes	yes
declaring the SQLCA is optional	no	yes	yes
no error message if output value is truncated	yes	yes	yes
"no data found" ORACLE error code is +100	yes	yes	yes
colon prefix required in the INTO clause	no	yes	yes

To ensure downward compatibility with applications written for the Version 1.3 and Version 1.4 ORACLE Precompilers, specify MODE=ANSI13 and MODE=ANSI14, respectively.

Some MODE and DBMS option values are incompatible or unrecommended. For more information, see the section "DBMS earlier in this chapter.

ONAME

Purpose Specifies the name of the output file.

Syntax ONAME=path and filename

Default System-dependent

Usage Notes Can be entered only on the command line.

By default, the output file is written to the current directory.

ORACA

Purpose Specifies whether a program can use the ORACLE Communications Area (ORACA).

Syntax ORACA=YES | NO

Default NO

Usage Notes Can be entered inline or on the command line.

When ORACA=YES, you must place the INCLUDE ORACA statement in your program.

ORECLEN

Purpose Specifies the record length of the output file.

Syntax ORECLEN=integer

Default 80

Usage Notes Can be entered only on the command line.

The value you specify for ORECLEN should equal or exceed the value of IRECLEN. The maximum value allowed is system-dependent.

PAGELEN

Purpose Specifies the number of lines per physical page of the listing file.

Syntax PAGELEN=integer

Default 66

Usage Notes Can be entered only on the command line.

The maximum value allowed is system-dependent.

RELEASE_CURSOR

Purpose Specifies how the cursors for SQL statements and lPL/SQL blocks are handled in the cursor cache.

Syntax RELEASE_CURSOR=YES I NO

Default N O

Usage Notes Can be entered inline or on the command line.

You can use RELEASE_CURSOR to improve the performance of your program. For more information, see Appendix E.

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an ORACLE private SQL area, which stores information needed to process the statement. RELEASE_CURSOR controls what happens to the link between the cursor cache and private SQL area.

When RELEASE_CURSOR=YES, after ORACLE executes the SQL statement and the cursor is closed, the precompiled immediately removes the link. This frees memory allocated to the private SQL area and releases parse locks. To make sure that associated resources are freed when you CLOSE a cursor, you must specify RELEASE_CURSOR=YES.

When RELEASE_CURSOR=NO and HOLD_CURSOR=YES, the link is maintained. The precompiled does not reuse the link unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an ORACLE private SQL area.

For inline use with implicit cursors, set RELEASE_CURSOR before executing the SQL statement. For inline use with explicit cursors, set RELEASE_CURSOR before OPENing the cursor.

Note: RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For a table showing how these two options interact, refer to Appendix E.

SELECT_ERROR

Purpose Specifies whether your program generates an error when a single-row SELECT statement returns more than one row or more rows than a host array can accommodate.

Syntax SELECT_ERROR=YES | NO

Default YES

Usage Notes Can be entered inline or on the command line.

When SELECT_ERROR=YES, an error is generated when a single-row SELECT returns too many rows or when an array SELECT returns more rows than the host array can accommodate. The result of the SELECT is indeterminate.

When SELECT_ERROR=NO, no error is generated when a single-row SELECT returns too many rows or when an array SELECT returns more rows than the host array can accommodate.

Whether you specify YES or NO, a random row is selected from the table. The only way to ensure a specific ordering of rows is to use the ORDER BY clause in your SELECT statement. When SELECT_ERROR=NO and you use ORDER BY, ORACLE returns the first row, or the first *n* rows when you are SELECTing into an array. When SELECT_ERROR=YES, whether or not you use ORDER BY, an error is generated when too many rows are returned.

SQLCHECK

Purpose Specifies the type and extent of syntactic and semantic checking.

Syntax SQLCHECK=SEMANTICS | FULL | SYNTAX | LIMITED | NONE

Default SYNTAX

Usage Notes Can be entered inline or on the command line.

The ORACLE Precompilers can help you debug a program by checking the syntax and semantics of embedded SQL statements and PL/SQL blocks. You control the level of checking by entering the SQLCHECK option inline and/or on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line. For example, if you specify SQLCHECK={SYNTAX | LIMITED} on the command line, you cannot specify SQLCHECK={SEMANTICS | FULL} inline.

When SQLCHECK={SEMANTICS | FULL}, the precompiled checks the syntax and semantics of

- data manipulation statements such as INSERT and UPDATE
- PL/SQL blocks
- host variable datatypes

as well as the syntax of data definition statements such as CREATE and ALTER. However, only syntactic checking is done on data manipulation statements that use the AT *db_name* clause.

When SQLCHECK={SEMANTICS | FULL}, the precompiled gets information needed for a semantic check by using embedded DECLARE TABLE statements or if you specify the USERID option on the command line, by connecting to ORACLE and accessing the data dictionary. You need not connect to ORACLE if every table referenced in a data manipulation statement or PL/SQL block is defined in a DECLARE TABLE statement.

If you connect to ORACLE, but some needed information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. A DECLARE TABLE definition overrides a data dictionary definition if they conflict.

If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK={SEMANITCS | FULL}.

When SQLCHECK={SYNTAX | LIMITED}, the precompiled checks the syntax of

- data manipulation statements
- data definition statements
- host variable datatypes

No semantic check is done. DECLARE TABLE statements are ignored, and PL/SQL blocks are not allowed.

Note: The values LIMITED and SYNTAX are equivalent; that is, they specify the same action. However, under Version 1.3 of the ORACLE Precompilers, the value LIMITED specifies a different action. With Version 1.3, when SQLCHECK=LIMITED, the precompiled checks the syntax and semantics of data manipulation statements and PL/SQL blocks. Also, to do the semantic check, the precompiled connects to ORACLE, so you must specify the USERID option.

When SQLCHECK=NONE, the precompiled does no semantic checking and minimal syntactic checking, DECLARE TABLE statements are ignored, and PL/SQL blocks are not allowed.

Specify SQLCHECK=NONE only if

- your program references tables not yet created and is missing DECLARE TABLE statements for them
- stricter datatype checking is undesirable

For more information about the SQLCHECK option, see Appendix F.

USERID

Purpose Specifies an ORACLE username and password.

Syntax USERID=username/password

Default None

Usage Notes Can be entered only on the command line.

Do not specify this option when using the automatic logon feature, which accepts your ORACLE username prefixed with OPS$.

When SQLCHECK=SEMANTICS, if you want the precompiled to get needed information by connecting to ORACLE and accessing the data dictionary, you must also specify USERID.

XREF

Purpose Specifies whether a cross-reference section is included in the listing file.

Syntax XREF=YES | NO

Default YES

Usage Notes Can be entered inline or on the command line.

When XREF=YES, cross references are included for host variables, cursor names, and statement names. The cross references show where each object is defined and referenced in your program.

When XREF=NO, the cross-reference section is not included.

Obsolete Options

With earlier ORACLE Precompilers, the AREASIZE option specified the size of the initial private SQL area opened for ORACLE cursors. You could respecify AREASIZE for each cursor or set of cursors used by your program.

The REBIND option specified how often host variables in SQL statements were bound. You could respecify REBIND for each SQL statement or set of SQL statements in your program.

The REENTRANT option specified whether reentrant code was generated. (A *reentrant* program or subroutine can be reentered before it has finished executing. Thus, it can be used simultaneously by two or more processes.) On some systems, you had to specify REENTRANT=YES.

With the Version 1.5 ORACLE Precompilers, private SQL areas are automatically resized, host variables are rebound only when necessary, and reentrant code is generated automatically for systems that require it. These advances make the AREASIZE, REBIND, and REENTRANT options obsolete. You no longer have to worry about using these options correctly. In fact, if you specify AREASIZE, REBIND, or REENTRANT, you get the following warning message:

```
PCC-I-0093 : Invalid or obsolete option, ignored
```

Doing Conditional Precompilations

Conditional precompilation includes (or excludes) sections of code in your host program based on certain conditions. For example, you might want to include one section of code when precompiling under UNIX and another section when precompiling under VMS. Conditional precompilation lets you write programs that can run in different environments.

Conditional sections of code are marked by statements that define the environment and actions to take. You can code host-language statements as well as EXEC SQL statements in these sections. The following statements let you exercise conditional control over precompilation:

```
EXEC ORACLE DEFINE symbol;    -- define a symbol
EXEC ORACLE IFDEF symbol; -- if symbol is defined
EXEC ORACLE IFNDEF symbol; -- if symbol is not defined
EXEC ORACLE ELSE;             -- otherwise
EXEC ORACLE ENDIF ;           -- end this control block
```

An Example

In the following example, the SELECT statement is precompiled only when the symbol *site2* is defined

```
EXEC ORACLE IFDEF site2;
    EXEC SQL SELECT DNAME
        INTO : dept_name
        FROM DEPT
        WHERE DEPTNO = : dept_number;
EXEC ORACLE ENDIF ;
```

Blocks of conditions can be nested as shown in the following example:

```
EXEC ORACLE IFDEF outer;
    EXEC ORACLE IFDEF inner;
    ...
    EXEC ORACLE ENDIF ;
EXEC ORACLE ENDIF ;
```

You can “comment out” host-language or embedded SQL code by placing it between IFDEF and ENDIF and *not* defining the symbol.

Defining Symbols

You can define a symbol in two ways. Either include the statement

```
EXEC ORACLE DEFINE symbol ;
```

in your host program or define the symbol on the command line using the syntax

```
... INAME=filename. . . DEFINE=symbol
```

where *symbol* is not case-sensitive.

Some port-specific symbols are predefined for you when the ORACLE Precompilers are installed on your system. Predefine operating system symbols include CMS, MVS, MSDOS, UNIX, and VMS. Predefine hardware symbols include IBM and DEC. For example, on a VMS system the symbols DEC and VMS are predefined.

Doing Separate Precompilations

With the ORACLE Precompilers, you can precompiled several host program modules separately, then link them into one executable program. The individual program modules need not be written in the same language.

Guidelines

The following guidelines will help you avoid some common problems.

Referencing Cursors

Cursor operations cannot span precompilation units (files). That is, you cannot DECLARE a cursor in one file and OPEN or FETCH from it in another file. So, when doing a separate precompilation, make sure all definitions and references to a given cursor are in one file.

Specifying MAXOPENCURSORS

When you precompiled the program module that CONNECTS to ORACLE, specify a value for MAXOPENCURSORS that is high enough for any of the program modules. If you use it for another program module, MAXOPENCURSORS is ignored. Only the value in effect for the CONNECT is used at run time.

Using a Single SQLCA

If you want to use just one SQLCA, you must declare it as global in one of the program modules and as external in the other modules. For example, in C, this is done by using the extern storage class, which tells the precompiled to look for the SQLCA in another program module. Unless you declare the SQLCA as external, each program module will use its own local SQLCA.

Compiling and Linking

To get an executable program, you must compile the modified source file(s) produced by the precompiled, then link the resulting object module with any modules needed from the ORACLE and OCI runtime libraries.

The linker resolves symbolic references in the object modules. If these references conflict, the link fails. This can happen when you try to link third party software into a precompiled program. Not all third-party software is compatible with ORACLE. So, linking your program *shared* might cause an obscure problem. In some cases, linking *stand-alone* or *two-task* might solve the problem.

Compiling and linking are system-dependent. For instructions, see the ORACLE installation or user's guide for your system.

APPENDIX

A

NEW FEATURES

This appendix looks at the array of new features offered by Versions 1.4 and 1.5 of the ORACLE Precompilers. Designed to meet the practical needs of professional software developers, these features will help you build effective, reliable applications.

New Features in Version 1.4

The Version 1.4 ORACLE Precompilers offer many improvements and new features including

- better performance
- close ANSI compliance
- datatype equivalencing
- an enhanced WHENEVER statement
- anew SQLCA debugging aid
- improved precompiled options
- a simpler precompiled cornmand

Better Performance

This section highlights new features designed to cut processing overhead.

Separate Executable

Previous ORACLE Precompilers accepted input files written in any licensed host language. So, they did a lot of multiway conditional processing. However, the Version 1.4 ORACLE Precompilers are separate executable programs, each designed for a specific host language. This streamlined design is simpler and saves precompilation time.

Terser, Faster Generated Code

The Version 1.4 ORACLE Precompilers generate terser, faster code than their predecessors. Moreover, the generated code calls an entirely rewritten set of SQL library routines adapted to a new ORACLE interface. As a result, your embedded SQL programs are easier to debug, use less memory, and run faster.

Bundled Database Calls

Previous ORACLE Precompilers generated several database calls per embedded SQL statement. However, the Version 1.4 ORACLE Precompilers generate only one (bundled) database call per embedded SQL statement. This speeds up communication with ORACLE, especially in a networked environment.

Improved Handling of Host Variables and Cursors

With previous ORACLE Precompilers, the AREASIZE option specified the size of the initial context area opened for ORACLE cursors. The REBIND option specified how often host variables in SQL statements were bound. With the Version 1.4 ORACLE Precompilers, context areas are automatically resized, and host variables are rebound only when necessary. These advances make the AREASIZE and REBIND options obsolete.

Close ANSI Compliance

The Version 1.4 ORACLE Precompilers comply closely with the current ANSI SQL standard. The changes described in this section make your applications more ANSI-compliant. Some changes are always in effect; others go into effect when MODE=ANSI14.

Changes Always in Effect

Introduced with Version 1.4 of the ORACLE Precompilers, the changes discussed in this subsection are *not* governed by the MODE option.

Optional FOR UPDATE OF Clause

The FOR UPDATE OF clause is optional (no longer required) when you DECLARE a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement.

New Keyword INDICATOR

To improve readability, you can precede any indicator variable with the optional keyword INDICATOR. You must still prefix the indicator variable with a colon. The correct syntax is

```
:host_variable INDICATOR :indicator_variable
```

Words No Longer Reserved

The words EXEC, IAF, ORACLE, and SQL are no longer reserved by the ORACLE Precompilers; they are just keywords and so can be redefined. That is, you can use them as host identifiers.

Changes in Effect When MODE=ANSI14

ANSI compliance is governed by the MODE option. When MODE=ANSI14, your program complies closely with the ANSI SQL standard and the following changes go into effect

- Array operations are not allowed.
- Issuing a COMMIT or ROLLBACK command closes all explicit cursors.
- You cannot OPEN an already open cursor or CLOSE an already closed cursor.
- You cannot SELECT or FETCH nulls into a host variable unless you append an indicator variable to it.
- You must declare a 4-byte integer variable named SQLCODE (SQLCOD in FORTRAN) inside or outside the Declare Section.
- Declaring the SQLCA is optional. You need not hardcode the SQLCA or copy it into your program with the INCLUDE statement.
- No error message is issued if ORACLE assigns a truncated column value to an output host variable.
- The “no data found’ ORACLE error code returned to SQLCODE becomes +100 instead of +1403. The error message text does not change.
- In SQL data manipulation statements, every host variable must be prefixed with a colon.

Datatype Equivalencing

The Version 1.4 ORACLE Precompilers add flexibility to your applications by letting you equivalence datatypes. You can equivalence host language datatypes to ORACLE external datatypes on a variable-by-variable basis. With Pro*C and Pro*Pascal, you can also equivalence user-defined types to ORACLE external datatypes.

Enhanced WHENEVER Statement

The WHENEVER statement lets you specify actions to be taken when ORACLE detects an error or warning condition. With previous ORACLE Precompilers, you had three choices: GOTO to a labeled statement, CONTINUE with the next statement, or STOP. With the Version 1.4 ORACLE Precompilers, you have a useful fourth choice: DO (call) a routine.

If a SQL statement fails, you can have your program transfer control to a routine. When the end of the routine is reached, control transfers to the statement that follows the failed SQL statement.

A New Debugging Aid

Starting with Version 1.4 of the ORACLE Precompilers, the SQLCA stores added runtime information about the outcome of SQL operations. Specifically, SQLERRD(5) stores a parse error offset to help you debug SQL statements.

New FOR Clause Restrictions

Starting with Version 1.4 of the ORACLE Precompilers, you can no longer use the FOR clause in SELECT statements because its meaning is unclear. Also, you can no longer use the FOR clause with the CURRENT OF clause because the only logical value of the FOR-clause variable is 1 (you can only update or delete the current row once).

Improved Precompiled Options

The SQLCHECK, INCLUDE, and MAXLITERAL precompiled options have been fine-tuned.

SQLCHECK Option

The Version 1.4 ORACLE Precompilers can help you debug a program by checking the syntax and semantics of embedded SQL statements and PL/SQL blocks. You control the extent of checking by entering the precompiled option SQLCHECK on the command line. You can specify the following values for SQLCHECK. SEMANTICS, SYNTAX, or NONE.

INCLUDE Option

With previous ORACLE Precompilers, you used the INCLUDE option to specify a directory path for EXEC SQL INCLUDE files. Typically, INCLUDE was used to specify a directory path for the SQLCA and ORACA files. The precompiled searched first in the current directory, then in the directory specified by INCLUDE.

The Version 1.4 ORACLE Precompilers search first in the current directory, then in the directory specified by INCLUDE, and finally in a directory for standard INCLUDE files. Hence, you need not specify a directory path for standard files such as the SQLCA and ORACA files.

MAXLITERAL Option

With previous ORACLE Precompilers, your program could set MAXLITERAL more than once, although there was no reason to do so. With the Version 1.4 ORACLE Precompilers, your program can set MAXLITERAL just once, and the EXEC ORACLE statement must precede the first EXEC SQL statement. Otherwise, the precompiled issues a warning message, ignores the extra or misplaced EXEC ORACLE statement, and continues processing.

It is unlikely that your old programs set MAXLITERAL more than once, but you might need to relocate the EXEC ORACLE statement.

Simpler Precompiled Command

To run a Version 1.4 ORACLE Precompiled, you issue a language-specific coremand. The only argument required by the precompiled command is

```
INAME=filename
```

The precompiled assumes the standard input file extension. So, you need not use a file extension when specifying INAME unless the extension is nonstandard.

Revised HOST Option

With previous ORACLE Precompilers, the HOST parameter was needed because the precompiled accepted input files written in any licensed host language. Also, on some ports, the HOST option was used to specify different compilers.

The Version 1.4 ORACLE Precompilers are separate executable programs, each designed for a specific host language. As a result, the HOST option is no longer needed for specifying the host language, but it remains for port-specific and COBOL-specific purposes.

Obsolete Options

With the Version 1.4 ORACLE Precompilers, context areas are automatically resized, host variables are rebound only when necessary, and reentrant code is generated automatically for systems that require it. These advances make the AREASIZE, REBIND, and REENTRANT options obsolete. You no longer have to worry about using these options correctly.

Pro*C Improvements

This section tells you about several improvements introduced with Version 1.4 of the Pro*C Precompiled.

Function Prototyping

The Version 1.4 Pro*C Precompiler generates function prototypes for SQL library routines so that your C compiler can resolve external references.

Flexible Declaration of Host Variables

The changes described in this section make your host program more flexible and more compliant with the ANSI C standard.

Support for Storage-class Specifiers

With the Version 1.4 Pro*C Recompiler, you can use the following storage-class specifiers to define host variables in the Declare Section

- auto
- extern
- static

Support for Type Modifiers

Now you can also use the following type modifiers in the Declare section

- const
- volatile

Enhanced WHENEVER Statement

In Pro*C, besides letting you call a function, the WHENEVER...DO statement lets you simulate a break statement, as follows:

```
EXEC SQL WHENEVER <condition> DO break;
```

You can use the DO break action to exit a loop prematurely or to keep execution from falling through to the next case in a switch statement.

New LINES Option

Introduced with Version 1.4 of the Pro*C Precompiled, the LINES option helps with debugging. When LINES=YES, the precompiled adds C #line directives to its output file.

Unrestricted SQL Line Continuation

With the Version 1.4 Pro*C Precompiled, you can continue string literals in a SQL statement from one line to the next.

Pro*COBOL Improvements

This section tells you about two improvements introduced with Version 1.4 of the Pro*COBOL Precompiled.

Revised HOST Option

With previous Pro*COBOL Precompilers, you used the HOST option to specify the host language of the input file or you used the standard input file extension (PCO) when specifying INAME.

With the Version 1.4 Pro*COBOL Precompiled, the HOST option is no longer needed for specifying the host language. However, if the host language dialect is COBOL-74, you must specify HOST=COB74 because the HOST option defaults to HOST=COBOL. The input file extension defaults to PCO. So, you must give the file extension only if it is nonstandard.

New DISPLAY SIGN LEADING SEPARATE Datatype

To make your applications more ANSI-compliant, Pro*COBOL now supports the DISPLAY SIGN LEADING SEPARATE datatype. This datatype typically requires *n + 1* bytes of storage for PIC S9 *(n)* and *n + d + 1* bytes of storage for PIC S9*(n)* V9 *(d)*.

Pro*FORTRAN Improvements

This section tells you about three improvements introduced with Version 1.4 of the Pro*FORTRAN Precompiled.

Unrestricted Precompilation

Previous Pro*FORTRAN Precompilers imposed the following restriction: in a given precompilation unit, only one program unit could contain SQL statements. The Version 1.4 Pro*FORTRAN Precompiled lifts that restriction. For example, now you can DECLARE a cursor in one program unit, OPEN it in another, FETCH from it in yet another, and CLOSE it in still another.

New COMMON_NAME Option

The Version 1.4 Pro*FORTRAN Precompiled uses a block data subprogram to establish COMMON blocks for all the SQL variables in an input file. The COMMON_NAME option specifies a prefix used to name the internal FORTRAN COMMON blocks. Your host program does not access the COMMON blocks directly, but they allow two or more program units in the same precompilation unit to contain SQL statements.

Obsolete Options

By default, previous Pro*FORTRAN Precompilers generated the following sequence of GOTO labels:

90000,90001,...,99999

You had to avoid using labels in this range or use the BEGLABEL and ENDLABEL options to specify another range for generated labels.

The Version 1.4 Pro*FORTRAN Precompiled generates no GOTO labels. This makes the BEGLABEL and ENDLABEL options obsolete. You no longer have to worry about using these options correctly.

Pro*PL/I Improvement

This section tells you about an improvement introduced with Version 1.4 of the Pro*PL/I Precompiled.

Unrestricted SQL Line Continuation

With the Version 1.4 Pro*PL/I Precompiled, you can continue string literals in a SQL statement from one line to the next.

New Features in Version 1.5

The Version 1.5 ORACLE Precompilers offer several improvements and new features including

- full ANSI compliance
- calls to stored PL/SQL procedures
- the use of host arrays within PL/SQL blocks
- full support for ORACLE7 SQL

Full ANSI Compliance

The Version 1.5 ORACLE Precompilers comply *fully* with SQL standards established by the American National Standards Institute (ANSI), the International Standards Organization (ISO), and the U.S. National Institute of Standards and Technology (NIST).

The ORACLE Precompilers conform to Level 2 of ANSI standard . X3.135-1989 and ISO standard 9075-1989 and provide the FIPS Flagger required by NIST standard FIPS PUB 127-1. The FIPS Flagger identifies nonconforming SQL elements and so helps you develop portable applications.

New PIPS Option

Anew option named FIPS enables the FIPS Flagger. You can enter the FIPS option inline or on the command line. When FIPS=YES, warning (not error) messages are issued if you use an ORACLE extension to ANSI SQL or use an ANSI SQL feature in a nonconforming manner.

Revised MODE Option

ANSI compliance is governed by the MODE option. When MODE=ANSI, your program complies fully with the ANSI SQL standard and the following changes go into effect

- CHAR column values, USER pseudocolumn values, character host values, and quoted literals are treated like ANSI fixed-length character strings. Also, ANSI-compliant blank-padding semantics are used when you assign, compare, INSERT, UPDATE, SELECT, or FETCH such values.
- Issuing a COMMIT or ROLLBACK command closes all explicit cursors.
- You cannot OPEN an already open cursor or CLOSE an already closed cursor.
- You cannot SELECT or FETCH nulls into a host variable not associated with an indicator variable.
- YOU must declare a 4-byte integer variable named SQLCODE (SQLCOD in FORTRAN) inside or outside the Declare Section.
- Declaring the SQLCA is optional. You need not hardcode the SQLCA or copy it into your program with the INCLUDE statement.
- No error message is issued if ORACLE assigns a truncated column value to an output host variable.
- The "no data found" ORACLE error code returned to SQLCODE becomes +100 instead of +1403. The error message text does not change.
- In SQL data manipulation statements, every host variable must be prefixed with a colon.

Calls to Stored Procedures

Unlike anonymous blocks, PL/SQL procedures can be compiled separately, stored in an ORACLE database, and invoked.

You create a stored procedure using an ORACLE tool such as SQL*Plus. Once compiled and stored in the ORACLE data dictionary, the procedure is a named database object, which can be reexecuted without being recompiled. You can invoke (call) the stored procedure from your host program using an anonymous PL/SQL block. Moreover, you can call remote procedures via database links.

Host Arrays within PL/SQL Blocks

The Version 1.5 ORACLE Precompilers let you pass input host arrays and indicator arrays to a PL/SQL block. Like a locally declared PL/SQL table, they can be indexed by a PL/SQL variable of type BINARY_INTEGER or by a host variable compatible with that type. Normally, the entire host array is passed to PL/SQL, but you can use the new ARRAYLEN statement to specify a smaller array dimension. Furthermore, you can use a procedure call to assign all the values in a host array to rows in a PL/SQL table.

Full Support for ORACLE7 SQL

The Version 1.5 ORACLE Precompilers support all the ORACLE7 SQL commands, functions, operators, and datatypes. For example, now you can

- UPDATE rows in remote tables using database links
- manage distributed transactions using the FORCE TRANSACTION clause in COMMIT and ROLLBACK statements
- dynamically manage database privileges using the SET ROLE statement
- use the new transcendental functions SIN, COS, TAN, SINH, COSH, TANH, EXP, LN, and LOG
- use the new UNION ALL set operator
- use the extended ORACLE internal datatypes VARCHAR2, RAW, LONG, and LONG RAW
- use the new ORACLE external datatypes CHAR, LONG VARCHAR, and LONG VARRAW
- use optimizer hints to influence decisions made by the ORACLE optimizer

Pro*C Improvements

This section tells you about two improvements introduced with Version 1.5 of the Pro*C Precompiled.

New CODE Option

Like its predecessor, the Version 1.5 Pro*C Precompiler generates function prototypes for SQL library routines so that your C compiler can resolve external references. The new CODE option lets you control the prototyping.

You can enter the CODE option inline or on the command line. When CODE=ANSI_C, the precompiled generates full function prototypes, which conform to the current ANSI C standard (X3.159-1989). When CODE=KR_C (the default), precompiled generates code compatible with the earlier C specified in Kernighan and Ritchie's book *The C Programming Language* (first edition). Consequently, the precompiled comments out the argument lists of generated function prototypes.

New CHARZ Datatype

The Version 1.5 Pro*C Precompiled supports a new ORACLE external datatype called CHARZ. CHARZ variables store ≤ 255-byte, fixed-length, null-terminated character strings. ANSI-compliant blank-padding semantics are used when you assign, compare, insert, or select CHARZ values.

APPENDIX

B

QUICK REFERENCE TO EMBEDDED SQL

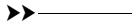
Rather than trying to memorize the syntax for SQL constructs you do not use every day, simply refer to this appendix. It focuses on the differences between embedded and interactive SQL, giving you the purpose of each embedded SQL statement, its syntax diagram, keyword and parameter descriptions, brief usage notes, and one or more programming examples. For detailed usage notes, see the ORACLE7 Server *SQL Language Reference Manual.*

Easy-to-understand *railroad diagrams are* used to illustrate embedded SQL syntax. They are line-and-arrow drawings that depict valid syntax (and resemble a railroad yard seen from above). If you have never used them, do not worry. The next section "How to Read Railroad Diagrams" tells you all you need to know.

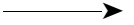
How to Read Railroad Diagrams

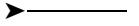
Once you understand the logical flow of a railroad diagram, it becomes a helpful guide. You can verify or construct any embedded SQL statement by tracing through its railroad diagram.

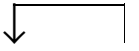
Railroad diagrams use lines and arrows to show how commands, parameters, and other language elements are sequenced to form statements. Trace each diagram from left to right, in the direction shown by the arrows. The following symbols will guide you:


Marks the beginning of the diagram.

Marks the end of the diagram.


Shows that the diagram continues on a line below.

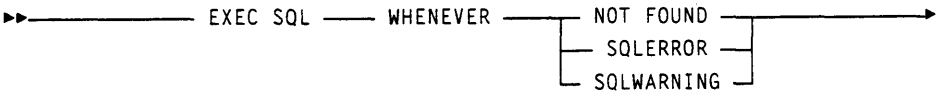

Shows that the diagram is continued from a line above.


Represents a loop.

Commands and other keywords appear in UPPERCASE. Parameters appear in lower case. Operators, delimiters, and terminators appear as usual. Following the conventions defined in the Preface, a semicolon terminates statements.

If the railroad diagram has more than one path, you can choose any path to travel.

If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list. In the following example, you can travel down the vertical line as far as you like, then continue along any horizontal line:



According to the diagram, all of the following statements are valid:

```
EXEC SQL WHENEVER NOT FOUND . . .
EXEC SQL WHENEVER SQLERROR . . .
EXEC SQL WHENEVER SQLWARNING . . .
```

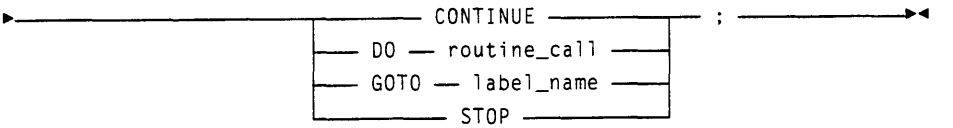
Required Keywords and Parameters

Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *cursor_name is* a required parameter:

If there is a cursor named *emp_cursor*, then, according to the diagram, the following statement is valid:

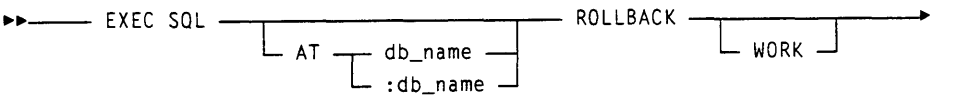
```
EXEC SQL CLOSE emp_cursor;
```

If any of the keywords *or* parameters in a vertical list appears on the main path, one of them is required. That is, you must choose one of the keywords or parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the four actions:



Optional Keywords and Parameters

If keywords and parameters appear in a vertical list below the main path, they are optional. That is, you need not choose one of them. In the following example, instead of traveling down a vertical line, you can continue along the main path:

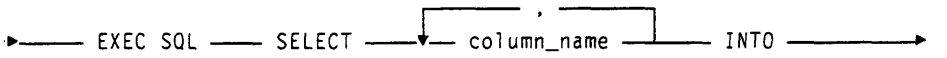


If there is a database named *oracle2*, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL ROLLBACK;
EXEC SQL ROLLBACK WORK;
EXEC SQL AT oracle2 ROLLBACK;
```

Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, *column_name* is inside a loop. So, after choosing one column name, you can go back repeatedly to choose another.

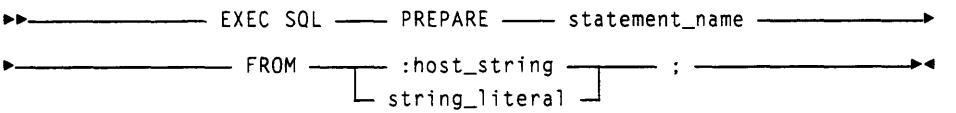


If DEBIT, CREDIT, and BALANCE are column names, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL SELECT DEBIT INTO . . .
EXEC SQL SELECT CREDIT, BALANCE INTO . . .
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO . . .
```

Multipart Diagrams

Read a multipart diagram as if all the main paths were joined end-to-end. The following example is a two-part diagram:



According to the diagram, the following statement is valid:

```
EXEC SQL PREPARE sql_statement FROM : sql_string;
```

Database Objects

The names of ORACLE objects, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if an ORACLE identifier is enclosed by quotation marks (“), it can contain any combination of legal characters, including spaces but excluding quotation marks.

ORACLE identifiers are not case-sensitive except when enclosed by quotation marks.

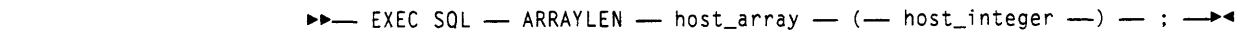
ARRAYLEN

Purpose

The ARRAYLEN statement limits the number of host array elements passed to a PL/SQL block for processing. By default, when binding a host array, the ORACLE Precompilers use its declared dimension. However, you might not want to process the entire array. In that case, you can use the ARRAYLEN statement.

Syntax

The following railroad diagram shows how to construct an ARRAYLEN statement



where *host_integer* is an integer host variable, *not* a literal or expression.

Usage Notes

The ARRAYLEN statement lets you downsize host arrays passed to a PL/SQL block. You override the default (declared) array dimension by specifying a smaller dimension. ARRAYLEN associates the host array with an integer host variable, which stores the smaller dimension.

The ARRAYLEN statement must appear in the Declare Section along with, but somewhere after, the declarations of *host_array* and *host_integer*. You cannot specify an offset into *host array*. However, you might be able to use host-language features for that purpose.

Example

The following example illustrates the use of ARRAYLEN:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    emp_name (150)   CHARACTER (10) ;
    limit            INTEGER ;
    EXEC SQL ARRAYLEN emp_name ( limit ) ;
EXEC SQL END DECLARE SECTION;
...
set limit = 25; -- set smaller array dimension
EXEC SQL EXECUTE
    DECLARE
        ...
    BEGIN
        -- process downsized host array
    END;
END- EXEC;
```

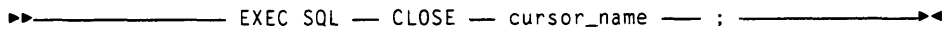
CLOSE

Purpose

The CLOSE statement disables a cursor, frees the resources acquired by OPENing the cursor, and releases parse locks.

Syntax

The following railroad diagram shows how to construct a CLOSE statement:



where *cursor_name* refers to a previously OPENed cursor.

Usage Notes

Attempting to CLOSE an undeclared or unOPENed cursor generates an error.

You need not CLOSE a cursor to reOPEN it when MODE=ORACLE.

When MODE={ANSI13 | ORACLE}, issuing a COMMIT or ROLLBACK closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by COMMIT or ROLLBACK and if open, remain open. However, when MODE= {ANSI | ANSI14}, issuing a COMMIT or ROLLBACK closes all explicit cursors.

The HOLD_CURSOR and RELEASE_CURSOR options alter the effect of CLOSE. To make sure that associated resources are freed when you CLOSE a cursor, you must specify RELEASE_CURSOR=YES. For more information, see the section "Using the Precompiled Options" in Chapter 11.

Example

The following example illustrates the use of CLOSE:

```
EXEC SQL CLOSE emp_cursor;
```

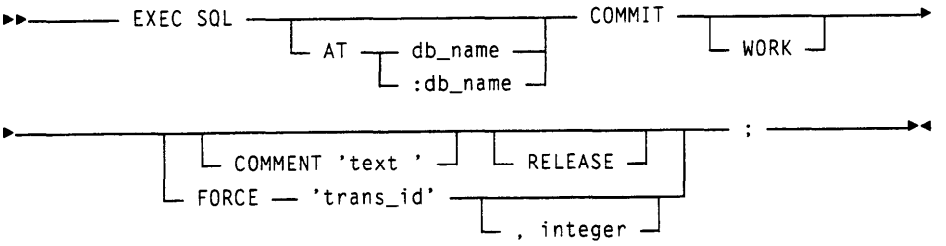
COMMIT

Purpose

The COMMIT statement ends the current transaction and makes changes to the database permanent.

Syntax

The following railroad diagram shows how to construct a COMMIT statement



where *db_name* identifies a non-default connection, WORK provides ANSI compatibility, COMMENT specifies a comment to be associated with the current transaction, RELEASE frees all resources and logs off the database, FORCE commits an in-doubt distributed transaction, *trans_id* identifies an in-doubt transaction, and *integer is* a system change number.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable.

Always explicitly COMMIT or ROLLBACK the last transaction in your program, specifying the RELEASE option to disconnect from ORACLE. If the program terminates abnormally, ORACLE automatically rolls back changes.

When MODE= {ANSI13 I ORACLE}, issuing a COMMIT closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by COMMIT and if open, remain open. However, when MODE={ANSI I ANSI14}, issuing a COMMIT closes all explicit cursors. COMMIT has no effect on host variables or the flow of control in a program.

You use FORCE to commit an in-doubt distributed transaction. The transaction must be identified by a quoted literal containing the transaction ID, which can be found in the data dictionary view DBA_2PC_PENDING. FORCE commits only the specified transaction and does not affect your current transaction.

Optionally, you can use *integer* to assign a system change number (SCN) to the transaction. If you omit *integer*, the current SCN is assigned to the transaction.

If ever a distributed transaction is in doubt, ORACLE stores the text specified by COMMENT in the data dictionary view DBA_2PC_PENDING along with the transaction ID. The text must be a quoted literal ≤ 50 characters in length.

Example

The following examples illustrate the use of COMMIT:

```
EXEC SQL COMMIT WORK RELEASE ;
...
EXEC SQL COMMIT COMMENT ‘ In-doubt trans; notify Order Entry’ ;
...
EXEC SQL AT oracle2 COMMIT RELEASE;
...
EXEC SQL COMMIT FORCE ’22.31.83 ‘ ;
```

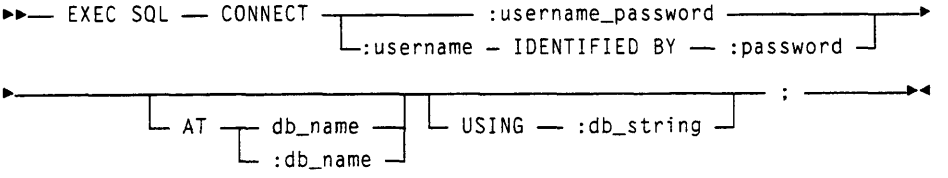
CONNECT

Purpose

The CONNECT statement logs onto an ORACLE database.

Syntax

The following railroad diagram shows how to construct a CONNECT statement



where *username* and *password* make up a valid ORACLE userid, *username_password* contains *username/password*, *db_name* identifies a non-default connection, and *db_string* is a SQL*Net string used to connect to a non-default database.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable. If *db_name* is not a host variable, you must declare *db_name* in a DECLARE DATABASE statement before executing a CONNECT . . . AT *db_name* statement.

Your program can have multiple CONNECT statements, but can only CONNECT once (implicitly specifying the default connection) without the AT clause. If you use the USING clause without the AT clause, the database referenced in the USING clause becomes the default connection.

Examples

The following example illustrates the use of CONNECT:

```
EXEC SQL CONNECT :username IDENTIFIED BY : password;
```

You can also use

```
EXEC SQL CONNECT :userid;
```

where *userid* contains *username/password.* Or, provided it is a valid ORACLE userid, you can automatically logon to ORACLE with

```
OPS$username
```

where *username* is your current operating system user or task name. You simply pass to ORACLE a one-character string containing a slash, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    userid      CHARACTER ( 1 ) ;
EXEC SQL END DECLARE SECTION;

set userid = ' /' ;

EXEC SQL CONNECT : userid;
```

This automatically connects you as user OPS$ *username.*

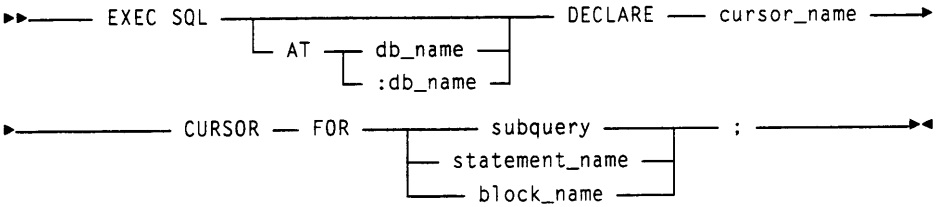
DECLARE CURSOR

Purpose

The DECLARE CURSOR statement defines a cursor by giving it a name and associating it with a specific query.

Syntax

The following railroad diagram shows how to construct a DECLARE CURSOR statement:



where *cursor_name* identifies a cursor, *db_name* identifies a non-default connection, *subquery* is a SELECT statement with no INTO clause, *statement_name* identifies a PREPAREd SQL statement, and *block_name* identifies a PREPAREd PL/SQL block.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable. If *db_name* is a host variable, its declaration must be within the scope of all SQL statements that refer to the DECLAREd cursor. For example, if you OPEN the cursor in one subprogram, then FETCH from it in another subprogram, you must declare *db_name* globally.

The *cursor_name, statement_name,* and *block_name* are undeclared identifiers used by the precompiled, *not* host or program variables. The *statement_name* is used with dynamic SQL Methods 3 and 4. The *block_name* is used with dynamic SQL Method 4.

You must DECLARE a cursor before referencing it in other SQL statements. The scope of a cursor declaration is global within the current precompilation unit. So, every DECLARE CURSOR statement must be unique. You cannot DECLARE two cursors with the same name in one precompilation unit.

Cursor names cannot be hyphenated. They can be any length, but only the first 31 characters are significant. For ANSI compatibility, use cursor names no longer than 18 characters.

You can reference the cursor in an UPDATE or DELETE statement using CURRENT OF *cursor* provided the cursor is open and positioned on a row.

If a PL/SQL block contains an unknown number of input or output host variables, you must use Method 4 to process the PL/SQL string. To use Method 4, you setup one bind descriptor for all the input and output host variables.

Executing DESCRIBE BIND VARIABLES stores information about input and output host variables in the bind descriptor. The precompiled treats all PL/SQL host variables as input host variables, so executing DESCRIBE SELECT LIST has no effect. The use of bind descriptors with Method 4 is detailed in your *Supplement to the ORACLE Precompilers Guide.*

Example

The following example illustrates the use of DECLARE CURSOR

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , EMPNO , JOB , SAL FROM EMP
    WHERE DEPTNO = : dept_number
    FOR UPDATE OF SAL;
```

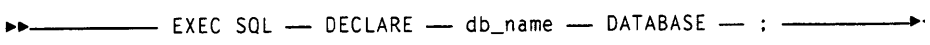
DECLARE DATABASE

Purpose

The DECLARE DATABASE statement declares the name of a non-default database for use in the AT clause of SQL statements.

Syntax

The following railroad diagram shows how to construct a DECLARE DATABASE statement



where *db_name* identifies a non-default database.

Usage Notes

The db_name is an undeclared identifier used by the precompiled, *not* a host or program variable.

You declare the non-default database so that other SQL statements can refer to that database in the AT clause. However, the DECLARE DATABASE statement is not needed if all database names in AT clauses are host variables.

Example

The following example illustrates the use of DECLARE DATABASE:

```
EXEC SQL DECLARE oracle3 DATABASE;
```

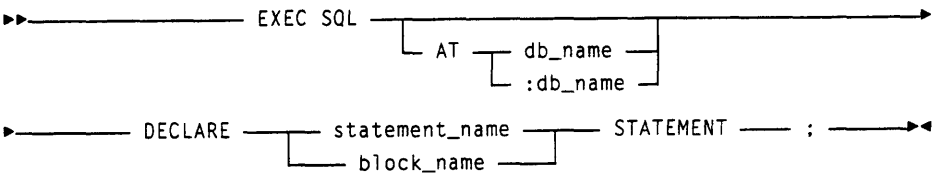
DECLARE STATEMENT

Purpose

The DECLARE STATEMENT statement declares the name of a dynamic SQL statement that will be PREPAREd and/or EXECUTEd.

Syntax

The following railroad diagram shows how to construct a DECLARE STATEMENT statement



where *db_name* identifies a non-default connection, *statement_name* identifies a PREPAREd SQL statement, and *block_name* identifies a PREPAREd PL/SQL block.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable. The *db_name* and *statement_name* are undeclared identifiers used by the precompiler, *not* host or program variables.

You must use DECLARE STATEMENT with dynamic SQL Methods 2, 3, and 4 if you want to execute the SQL statement at a remote database.

With Methods 3 and 4, DECLARE STATEMENT is also required if the DECLARE CURSOR statement physically (not logically) precedes the PREPARE statement.

Like that of a cursor declaration, the scope of a statement declaration is global within a precompilation unit.

Example

The following examples illustrate the use of DECLARE STATEMENT. In the first example, Method 2 is used to execute a dynamic SQL statement at a remote database:

```
EXEC SQL AT oracle2 DECLARE sql_statement STATEMENT;
EXEC SQL PREPARE sql_statement FROM : sql_string;
EXEC SQL EXECUTE sql_statement;
```

In the next example, DECLARE STATEMENT is required because DECLARE CURSOR physically precedes PREPARE:

```
EXEC SQL DECLARE sql_statement STATEMENT;
call  prepare_sql_statement;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_statement;
...
ROUTINE  prepare_sql_statement
BEGIN
    EXEC SQL PREPARE sql_statement FROM :sql_string;
END;
```

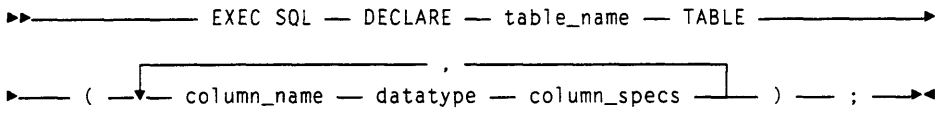
DECLARE TABLE

Purpose

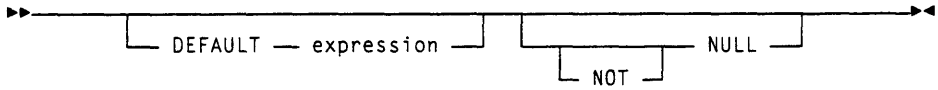
The DECLARE TABLE statement defines the structure of a table or view, including each column's datatype, optional default value, and optional NULL or NOT NULL specification.

Syntax

The following railroad diagram shows how to construct a DECLARE TABLE statement:



where *column_specs* stands for the following syntax:



Usage Notes

A DECLARE TABLE statement serves only as program documentation unless SQLCHECK=SEMANTICS. For more information about the SQLCHECK option, see the section "Using the Precompiled Options" in Chapter Il.

You can use the optional keywords WITH DEFAULT to maintain compatibility with IBM's DB2.

If the database name in the AT clause of a SQL statement is a host variable, all database tables referenced by the SQL statement must be defined in DECLARE TABLE statements. Otherwise, the precompiled issues a warning.

Example

The following example illustrates the use of DECLARE TABLE:

```
EXEC SQL DECLARE PARTS TABLE
     ( PARTNO NUMBER NOT NULL,
      BIN     NUMBER,
      QTY     NUMBER ) ;
```

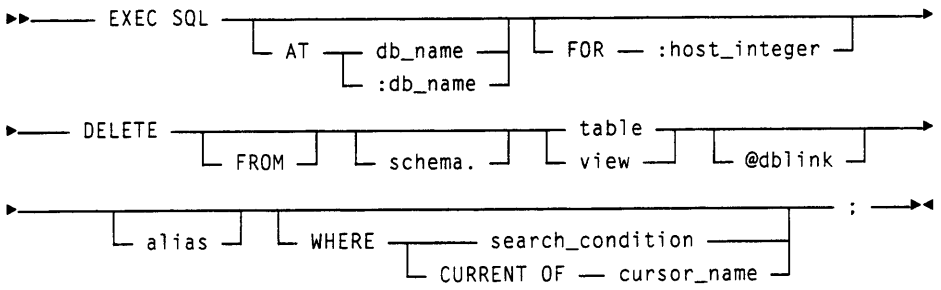
DELETE

Purpose

The DELETE statement removes unwanted rows from a table.

Syntax

The following railroad diagram shows how to construct a DELETE statement



where *db_name* identifies a non-default connection, FOR *:host_integer specifies the* maximum number of host array elements processed, *table_name* identifies the table from which rows will be deleted, *alias* is another name assigned to the table, *search condition* is a Boolean expression containing references to host variables or host arrays, and *cursor_name* refers to a cursor defined in a previous DECLARE CURSOR statement.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable. If *db_name* is a host variable, all database tables referenced by the DELETE statement must be defined in DECLARE TABLE statements.

When MODE=ANSI14, array DELETEs are *not* allowed; that is, you can reference host arrays in a DELETE statement only when MODE={ANSI | ANSI13 | ORACLE}. If one of the host variables in the WHERE clause is an array, all must be arrays. The host arrays can have different dimensions, in which case the number of array elements processed is determined by comparing the dimension of the *smallest* host array in the SQL statement with the optional FOR-clause variable. The lesser value is used. If the host variables in the WHERE clause are arrays, the DELETE statement is executed once for each set of array elements.

The cumulative number of rows deleted is returned to the third element of SQLERRD in the SQLCA. The number does *not* include rows processed by a delete cascade.

If no rows meet the *search_condition*, none are deleted, and the "no data found" ORACLE error code is returned to SQLCODE in the SQLCA.

If you omit the WHERE clause, all rows of the table are deleted and the fifth element of SQLWARN in the SQLCA is set.

several restrictions apply to the CURRENT OF clause; see the section "WHERE Clause" at the end of this chapter.

Examples

The following examples illustrate the use of DELETE:

```
EXEC SQL DELETE FROM EMP
    WHERE DEPTNO =  : dept_number AND JOB = : job_title;
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT EMPNO , COMM FROM EMP
        WHERE DEPTNO =  :dept_number
        FOR UPDATE OF COMM ;

EXEC SQL OPEN emp_cursor;

EXEC SQL FETCH emp_cursor INTO : emp_number, : commission;

EXEC SQL DELETE FROM EMP
    WHERE CURRENT OF emp_cursor;
```

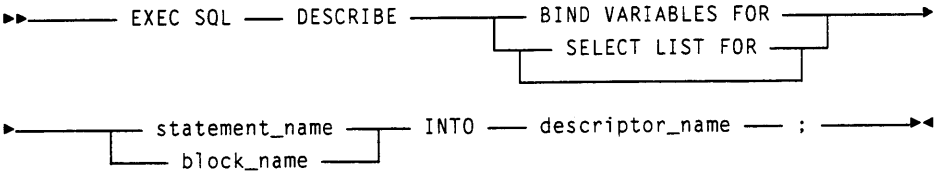
DESCRIBE

Purpose

The DESCRIBE statement initializes a descriptor to hold descriptions of the select-list items or input host variables in a l?REl?AREd dynamic SQL statement.

Syntax

The following railroad diagram shows how to construct a DESCRIBE statement



where BIND VARIABLES FOR specifies a list of input host variables, SELECT LIST FOR specifies a list of output host variables, *statement_name* identifies a PREPAREd SQL statement, *block_name* identifies a PREPAREd PL/SQL block, and *desciptor_name* identifies a descriptor.

Usage Notes

The statement_name and *block_name* are undeclared identifiers used by the precompiled, *not* host or program variables.

DESCRIBE is used with dynamic SQL Method 4. You must execute DESCRIBE before calling host functions that manipulate bind or select descriptors. You cannot use the same descriptor for input and output host variables simultaneously.

The number of variables found by DESCRIBE equals the total number of placeholders in the PREPAREd string, not the number of uniquely named placeholders.

Explicit use of the SELECT LIST FOR clause is optional.

Example

The following example illustrates the use of DESCRIBE:

```
EXEC SQL PREPARE sql_statement FROM : sql_string;
EXEC SQL DECLARE emp_cursor FOR . . .
EXEC SQL DESCRIBE BIND VARIABLES FOR sql_statement
    INTO bind_descriptor;
EXEC SQL OPEN emp_cursor USING . . .
EXEC SQL DESCRIBE SELECT LIST FOR sql_statement
    INTO select_descriptor;
```

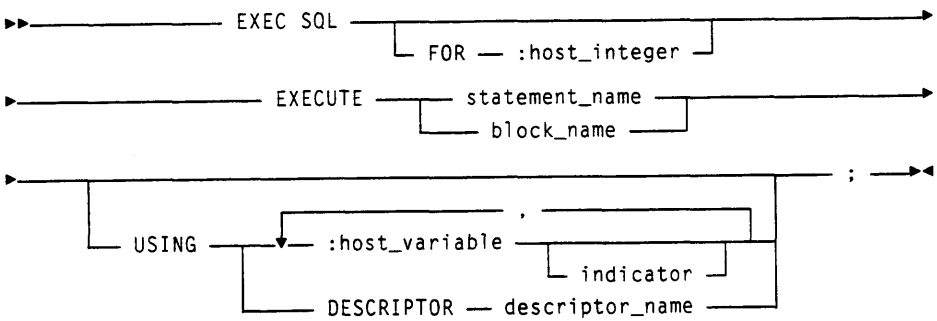
EXECUTE

Purpose

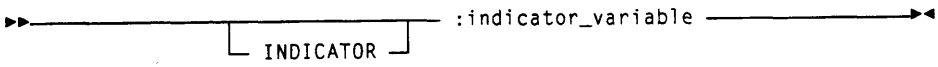
The EXECUTE statement executes a previously PREPAREd dynamic SQL statement or PL/SQL block.

Syntax

The following railroad diagram shows how to construct an EXECUTE statement



where FOR *:host_integer* specifies the maximum number of host array elements processed, *statement_name* identifies a PREPAREd SQL statement, *block_name* identifies a PREPAREd PL/SQL block, USING specifies a descriptor or a list of host variables substituted for placeholders in a PREPAREd string, *descriptor_name* identifies a descriptor referenced in a previous DESCRIBE statement, and *indicator stands* for the following syntax:



Usage Notes

The *statement_name* and *block_name* are undeclared identifiers used by the precompiled, *not* host or program variables. The *statement_name* must not identify a SELECT statement.

The EXECUTE statement is used mainly with dynamic SQL Method 2 but can also be used for nonqueries with Method 4.

Every placeholder in the PREPAREd string must correspond to a host variable in the USING clause. So, if the same placeholder appears two or more times in the PREPAREd string, each appearance must correspond to a host variable in the USING clause.

The names of the placeholders need not match the names of the host variables. However, the order of the placeholders in the PREPAREd string must match the order of the host variables in the USING clause. If one of the host variables in the USING clause is an array, all must be arrays.

If a PL/SQL block contains a known number of input and output host variables, you can use Method 2 to PREPARE and EXECUTE the PL/SQL string. The precompiled treats all PL/SQL host variables as input host variables whether they serve as input or output host variables (or both) inside the PL/SQL block. So, you must put all host variables in the USING clause.

Example

The following example illustrates the use of EXECUTE:

```
EXEC SQL PREPARE sql_statement FROM : sql_string;
EXEC SQL EXECUTE sql_statement USING : emp_number;
```

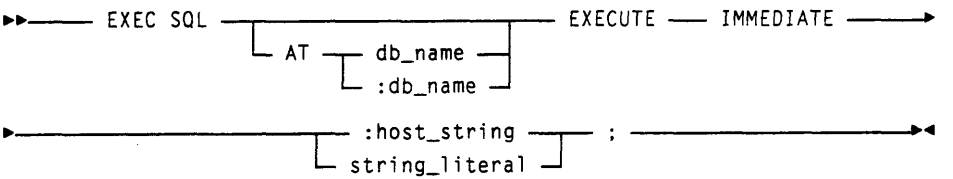
EXECUTE IMMEDIATE

Purpose

The EXECUTE IMMEDIATE statement prepares and executes dynamic SQL statements (except queries) and PL/SQL blocks, provided they contain no host variables.

Syntax

The following railroad diagram shows how to construct an EXECUTE IMMEDIATE statement



where *db_name* identifies a non-default connection, *host_string* is a host variable that contains the text of a SQL statement or PL/SQL block, and *string_literal* is a string literal that contains the text of a SQL statement or PL/SQL block.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable.

EXECUTE IMMEDIATE is used with dynamic SQL Method 1. When an EXECUTE IMMEDIATE statement is executed, the specified SQL statement is parsed and checked for errors. If the statement is invalid, it is not executed, and the appropriate ORACLE error code is returned to SQLCODE in the SQLCA.

EXECUTE IMMEDIATE is most useful for SQL statements executed only once. For statements executed repeatedly, use dynamic SQL Method 2 because a PREPAREd statement is not reparsed every time it is reexecuted.

If a PL/SQL block contains no host variables, you can use Method 1 to EXECUTE the PL/SQL string.

When you store a SQL statement in *host_string* or *string_literal,* omit the keywords EXEC SQL and the statement terminator. Likewise, when you store a PL/SQL block in *host_string* or *string_literal,* omit the keywords EXEC SQL EXECUTE, the keyword END-EXEC, and the statement terminator.

Examples

The following examples illustrate the use of EXECUTE IMMEDIATE:

```
set sql_string = ' DELETE FROM EMP WHERE EMPNO = 7839 ' ;
EXEC SQL EXECUTE IMMEDIATE : sql_string;
```

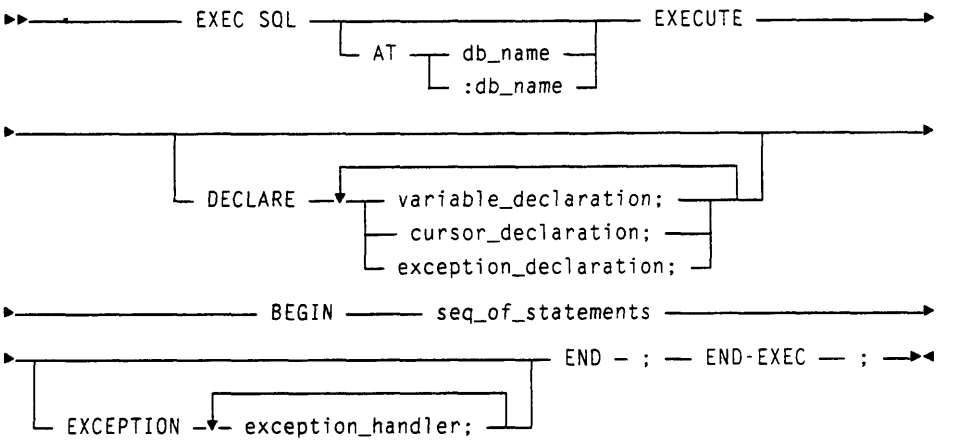
EXECUTE plsql_block

Purpose

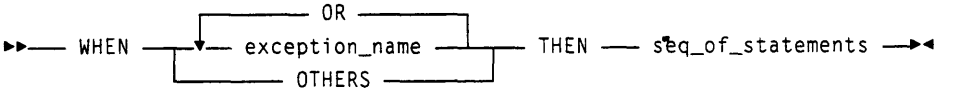
The EXECUTE plsql_block statement executes an embedded PL/SQL block.

Syntax

The following railroad diagram shows how to construct an EXECUTE *plsql_block* statement



where *db_name* identifies a non-default connection, *variable_declaration* declares PL/SQL constants and variables, *cursor_declaration* declares PL/SQL cursors, *exception_declaration* declares PL/SQL exceptions (error conditions), *seq_of_statements* is a sequence of PL/SQL statements, and *exception_handler* stands for the following syntax



where *exception_name* identifies a PL/SQL predefined or user-defined exception.

Usage Notes

The *db.name* can be an undeclared identifier or a host variable.

DECLARE marks the start of local declarations. Only the current block and its sub-blocks can reference locally declared objects.

The EXCEPTION_INIT pragma (compiler directive) can also be used in the declarative part of a block. For details, see the *PL/SQL User's Guide and Reference.*

A cursor declaration names a cursor and associates it with a query. The cursor name is not a PL/SQL variable. You cannot assign values to a cursor name or use it as a value in a PL/SQL expression. However, cursors and variables follow the same scoping rules.

Exceptions are unlike variables and constants in that they cannot be passed as arguments to functions or procedures.

An exception handler executes a sequence of statements in response to a raised exception. Some exceptions are predefined and need not be declared. For a list of these exceptions, see the *PL/SQL User's Guide and Reference.*

The scope rules for exception names and variables are the same. So, an exception declared in a sub-block overrides an exception declared with the same name in an enclosing block, and an enclosing block cannot reference exceptions declared in a sub-block.

Embedding PL/SQL Blocks

The ORACLE Precompilers treat a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in a host program that you can place a SQL statement.

Inside a PL/SQL block, host variables are treated as global to the entire block and can be used anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon.

When entering a PL/SQL block, ORACLE automatically checks the length fields of VARCHAR host variables. So, you must set the length fields before the block is entered. For input host variables, set the length field to the actual length of the value stored in the string field. For output host variables, set the length field to the maximum length allowed by the string field.

In a PL/SQL block, you cannot refer to an indicator variable by itself; it must be appended to its associated host variable. Also, if you refer to a host variable with its indicator variable, you must always refer to it that way in the same block.

When entering a block, if an indicator variable has a value of -1, PL/SQL automatically assigns a null to the host variable. When exiting the block, if a host variable is null, PL/SQL automatically assigns a value of-1 to the indicator variable.

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string.

Example

The following example illustrates the use of EXECUTE *plsql_block:*

```
EXEC SQL EXECUTE
    BEGIN
        SELECT ENAME , JOB , SAL
            INTO : emp_name: ind_name, : job_title, : salary
            FROM EMP
            WHERE EMPNO =  : emp_number;

        IF :emp_name:ind_name IS NULL THEN
            RAISE name_missing;
        END IF;
        ...
    END ;
END- EXEC;
```

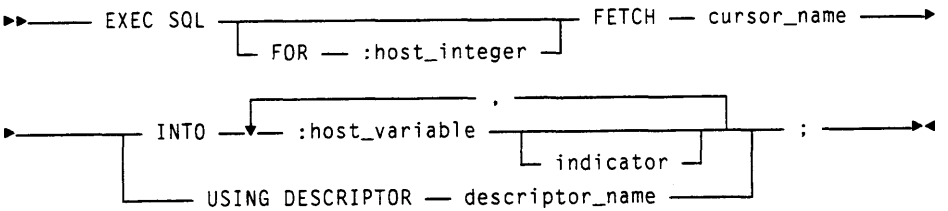
FETCH

Purpose

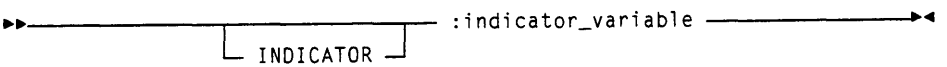
The FETCH statement retrieves a row from the database, assigns column values to host variables, and advances the cursor to the next qualified row.

Syntax

The following railroad diagram shows how to construct a FETCH statement



where FOR *:host_integer* specifies the maximum number of host array elements processed, *cursor_name* refers to a cursor defined in a previous DECLARE CURSOR statement, INTO *:host_variable* identifies one or more output host variables (each optionally associated with an indicator variable), *descriptor_name* identifies a descriptor referenced in a previous DESCRIBE statement, and *indicator* stands for the following syntax:



Usage Notes

The *cursor_name* and *descriptor_name* are undeclared identifiers used by the precompiled, *not* host or program variables.

The cursor must be DECLAREd and OPENed before a FETCH is executed.

The associated SELECT statement cannot have an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement.

The FETCH statement reads the rows of the active set and specifies the output host variables. The number of rows retrieved is determined by the dimensions of the output host variables in the FETCH statement and the value of the optional FOR variable. The cursor is positioned on the next row in the table (if any), and values are assigned to host variables in the INTO clause of the FETCH statement or the USING clause of the OPEN statement.

If you want to change the active set, you must assign new values to the input host variables in the query associated with the cursor, then reOPEN the cursor. When MODE= {ANSI | ANSI14 | ANSI13}, you must CLOSE the cursor before reOPENing it.

You can FETCH from the same cursor using different sets of output host variables. However, corresponding host variables in the INTO clause of each FETCH statement must have the same datatype.

If the cursor is currently positioned after the last row of the target table, the "no data found" ORACLE error code is returned to SQLCODE in the SQLCA.

When MODE=ANSI14, array FETCHes are *not* allowed; that is, you can reference host arrays in a FETCH statement only when MODE= {ANSI | ANSI13 | ORACLE}. If one of the host variables in the INTO clause is an array, all must be arrays.

Example

The following example illustrates the use of FETCH:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT JOB , SAL FROM EMP WHERE DEPTNO = 30 ;
...
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO. . .
LOOP
    EXEC SQL FETCH emp_cursor INTO : job_title1, : salary;
    EXEC SQL FETCH emp_cursor INTO : job_title2, : salary;
    ...
ENDLOOP;
...
```

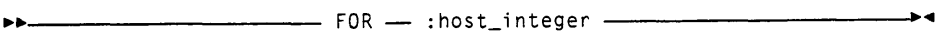
FOR

Purpose

The FOR clause limits the number of repetitions for array processing in an UPDATE, DELETE, INSERT, OPEN, FETCH, or EXECUTE statement.

Syntax

The following railroad diagram shows how to construct a FOR clause:



where FOR :*host_integer* specifies the maximum number of host array elements processed.

Usage Notes

You cannot use the FOR clause in a SELECT statement or with the CURRENT OF clause. For an explanation, see the section "Using the FOR Clause" in Chapter 8.

The host arrays can have different dimensions, in which case the number of array elements processed is determined by comparing the dimension of the *smallest* host array in the SQL statement with the FOR-clause variable. The lesser value is used.

If the value of the FOR-clause variable is less than or equal to zero, no rows are processed.

The FOR clause is optional. If you omit the FOR clause, the dimension of the smallest array in the SQL statement determines the number of array elements processed.

Example

The following example illustrates the use of the FOR clause:

```
set limit = 10;
EXEC SQL FOR :limit DELETE FROM EMP
    WHERE ENAME = : ename_array AND JOB = : job_array;
```

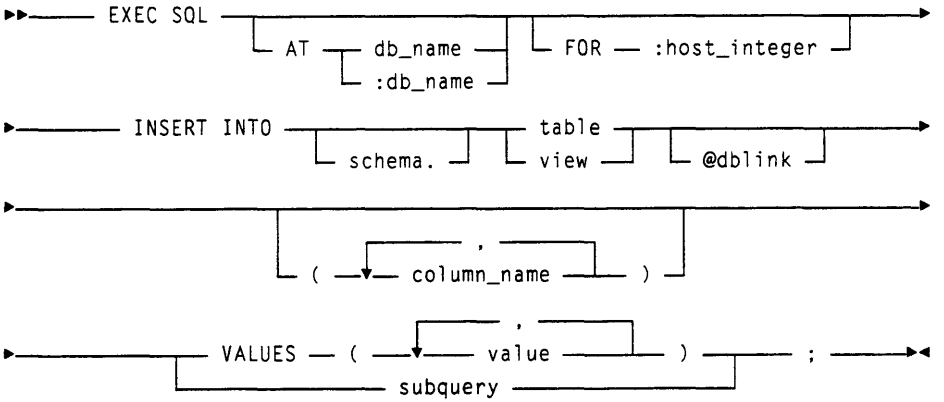
INSERT

Purpose

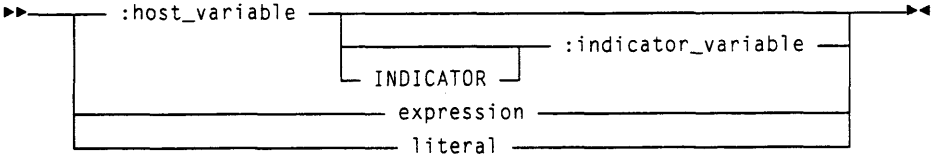
The INSERT statement adds new rows to a table or view.

Syntax

The following railroad diagram shows how to construct an INSERT statement



where *db_name* identifies a non-default connection, FOR *:host_integer* specifies the maximum number of host array elements processed, INTO *table_name* identifies the table to which rows will be added, *column_name* identifies a column for which a value is provided, VALUES introduces a row of values to be inserted, *subquery* is a SELECT statement with no INTO clause, and *value* stands for the following syntax:



Usage Notes

The *db_name* can be an undeclared identifier or a host variable. If *db_name* is a host variable, all database tables referenced by the INSERT statement must be defined in DECLARE TABLE statements.

Each of the specified columns must belong to the table named in the INTO clause. The number of values in the VALUES clause (or number of items in the subquery select list) must equal the number of names in the column list. However, you can omit the column list if the VALUES clause contains a value for each column in the table.

When MODE=ANSI14, array INSERTS are *not* allowed; that is, you can reference host arrays in a INSERT statement only when MODE={ANSI | ANSI13 | ORACLE}. If one of the host variables in the VALUES or WHERE clause is an array, all must be arrays. The host arrays can have different dimensions, in which case the number of array elements processed is determined by comparing the dimension of the *smallest* host array in the SQL statement with the optional FOR-clause variable. The lesser value is used. If the host variables in the WHERE clause are arrays, the INSERT statement is executed once for each set of array elements.

Examples

The following examples illustrate the use of INSERT

```
EXEC SQL INSERT INTO EMP ( ENAME , EMPNO , DEPTNO )
    VALUES (: emp_name, : emp_number, : dept_number ) ;
...
EXEC SQL INSERT INTO EMP2 ( ENAME , EMPNO , DEPTNO )
    SELECT ENAME , EMPNO , DEPTNO FROM EMP
        WHERE JOB =  :job_title;
```

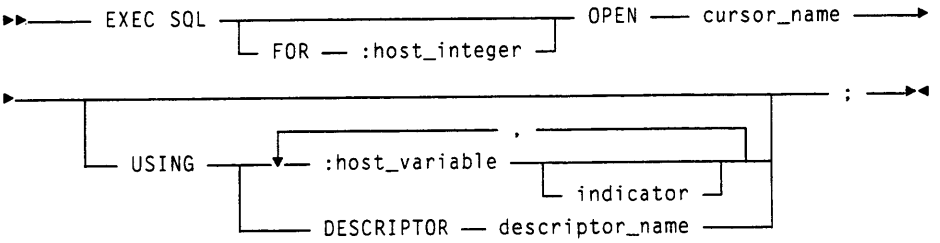
OPEN

Purpose

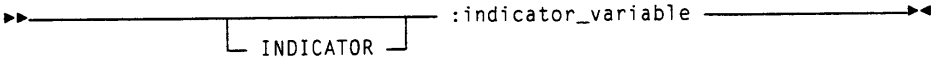
The OPEN statement opens a previously DECLAREd cursor, evaluates the query associated with the cursor, and substitutes in the WHERE clause of the query any host variable names supplied by the USING clause.

Syntax

The following railroad diagram shows how to construct an OPEN statement



where FOR *:host_integer* specifies the maximum number of host array elements processed, *cursor_name* refers to a cursor defined in a previous DECLARE CURSOR statement, USING specifies a descriptor or a list of host variables substituted for placeholders in a PREPAREd string, *descriptor_name* identifies a descriptor referenced in a previous DESCRIBE statement, and *indicator* stands for the following syntax:



Usage Notes

OPEN positions the cursor just before the first row of the active set. No rows are actually retrieved at this point; that is done by the FETCH statement.

Once you OPEN a cursor, its input host variables are not reexamined unless you reOPEN it. To change the active set, you must reOPEN the cursor with new values for the input host variables.

You need not CLOSE a cursor before reOPENing it when MODE=ORACLE.

The USING clause is used with dynamic SQL Methods 3 and 4. Typically, variable names in the USING clause differ from placeholder names in the PREPAREd string. Host variables are substituted for placeholders based on position.

Example The following example illustrates the use of OPEN

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, EMPNO, JOB, SAL FROM EMP
    WHERE DEPTNO =  :dept_number;
EXEC SQL OPEN emp_cursor;
```

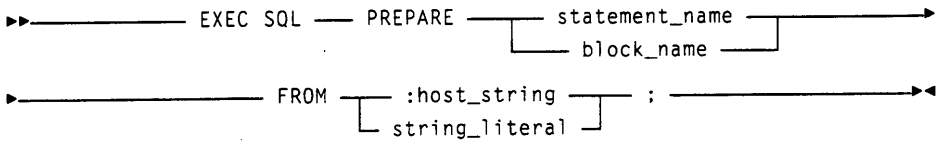
PREPARE

Purpose

The PREPARE statement parses the text of a dynamic SQL statement stored in a string host variable or string literal.

Syntax

The following railroad diagram shows how to construct a PREPARE statement



where *statement_name* identifies a PREPAREd SQL statement, *block_name* identifies a PREPAREd PL/SQL block, *host_string* is a host variable that contains the text of a SQL statement or PL/SQL block, and *string_literal* is a string literal that contains the text of a SQL statement or PL/SQL block.

Usage Notes

The *statement_name* and *block_name* are undeclared identifiers used by the precompiled, *not* host or program variables. If *statement_name* (or *block_name)* has already been assigned to another PREPAREd statement, that assignment is superseded.

The PREPARE statement is used with dynamic SQL Methods 2,3, and 4. When you store a SQL statement in *host_string* or *string_literal*, omit the keywords EXEC SQL and the statement terminator. Likewise, when you store a PL/SQL block in *host_string* or *string_literal*, omit the keywords EXEC SQL EXECUTE, the keyword END-EXEC, and the statement terminator.

The variables referenced in *host_string* are just placeholders. The actual host variable names are assigned in the OPEN USING clause or in the FETCH INTO clause. The substitution is based on position and has nothing to do with the placeholder names used. The number of variables in the USING clause and the number of placeholders in the PREPAREd statement must be the same.

A SQL statement is parsed only once, but can be executed many times. SQL data definition statements are executed when they are PREPAREd.

Example

The following example illustrates the use of PREPARE:

```
EXEC SQL PREPARE sql_statement FROM : sql_string;
EXEC SQL EXECUTE sql_statement;
```

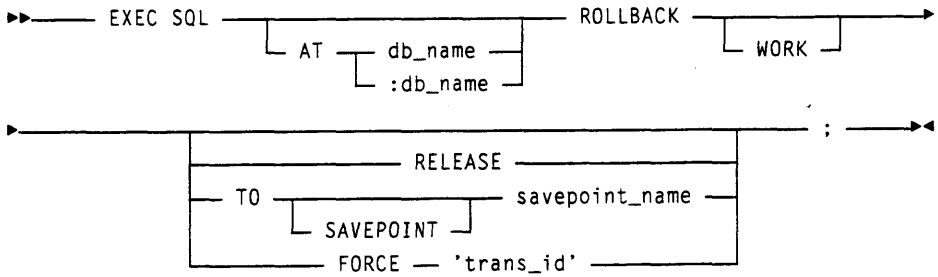
ROLLBACK

Purpose

The ROLLBACK statement ends the current transaction, undoes changes to the database, and releases all locks except parse locks held by cached cursors.

Syntax

The following railroad diagram shows how to construct a ROLLBACK statement:



where *db_name* identifies a non-default connection, WORK provides ANSI compatibility, *savepoint_name* identifies a marked savepoint, TO rolls back the current transaction to a previously marked savepoint, RELEASE frees all resources and logs off the database, FORCE rolls back an in-doubt distributed transaction, and *trans_id* identifies an in-doubt transaction.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable.

Always explicitly COMMIT or ROLLBACK the last transaction in your program, specifying the RELEASE option to disconnect from ORACLE. If the program terminates abnormally, ORACLE automatically rolls back changes. However, you cannot specify the RELEASE option in a ROLLBACK TO SAVEPOINT statement.

When MODE={ANSI13 | ORACLE}, issuing a ROLLBACK closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by ROLLBACK and if open, remain open. However, when MODE={ANSI | ANSI14}, issuing a ROLLBACK closes all explicit cursors. ROLLBACK has no effect on host variables or the flow of control in a program.

You use FORCE to rollback an in-doubt distributed transaction. The transaction must be identified by a quoted literal containing the transaction ID, which can be found in the data dictionary view DBA_2PC_PENDING. FORCE rolls back only the specified transaction and does not affect your current transaction. You cannot roll back in-doubt transactions to a savepoint.

Example

The following examples illustrate the use of ROLLBACK:

```
EXEC SQL ROLLBACK WORK RELEASE ;
...
EXEC SQL ROLLBACK TO SAVEPOINT begin_insert;
...
EXEC SQL ROLLBACK FORCE '22.31.83';
```

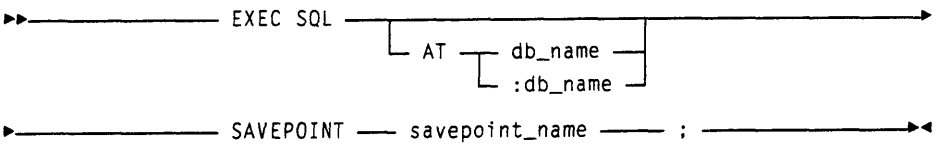
SAVEPOINT

Purpose

The SAVEPOINT statement names and marks the current point in the processing of a transaction. It is used with the ROLLBACK statement to undo part of a transaction.

Syntax

The following railroad diagram shows how to construct a SAVEPOINT statement



where *db_name* identifies a non-default connection, and *savepoint_name* identifies a marked savepoint.

Usage Notes

The *db_name* can be an undeclared identifier or a host variable. The *savepoint_name* is an undeclared identifier used by the precompiled, *not* a host or program variable.

To undo part of a transaction, you use savepoints with the ROLLBACK statement and its TO SAVEPOINT clause. Savepoints let you divide long, complex transactions, giving you more control over them.

Rolling back to a savepoint erases any savepoints marked after that savepoint. The savepoint to which you roll back, however, is not erased. For example, if you mark five savepoints, then rollback to the third, only the fourth and fifth are erased.

If you give two savepoints the same name, the earlier savepoint is erased. A COMMIT or ROLLBACK statement erases all savepoints.

You can use the FORCE clause in a ROLLBACK statement to roll back in-doubt distributed transactions. However, you cannot roll back in-doubt transactions to a savepoint.

Example

The following example illustrates the use of SAVEPOINT:

```
EXEC SQL SAVEPOINT begin_update;
EXEC SQL UPDATE EMP SET . . .
```

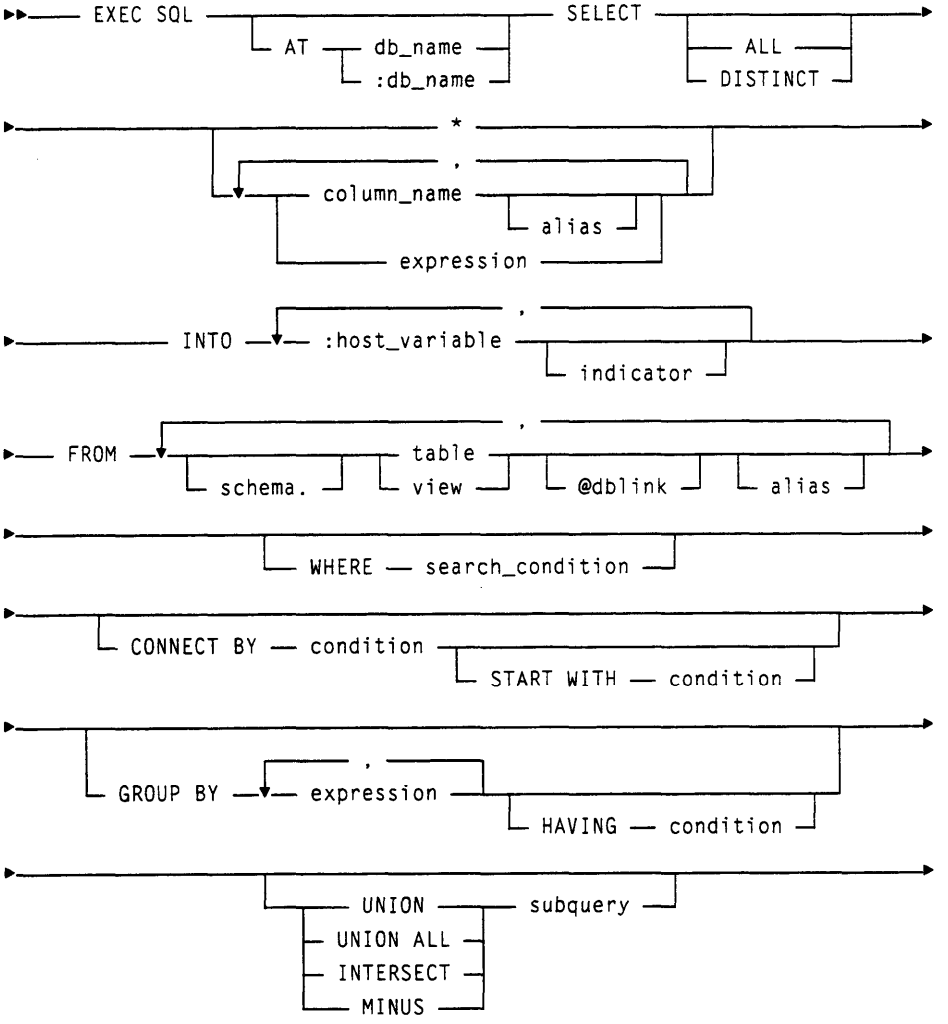
SELECT

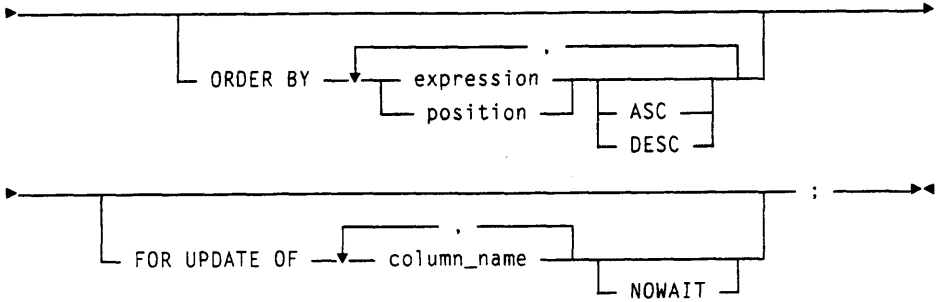
Purpose

The SELECT statement retrieves rows from one or more tables and assigns returned column values to output host variables.

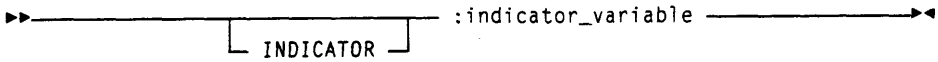
Syntax

The following railroad diagram shows how to construct a SELECT statement:





where *db_name* identifies a non-default connection, *table_name* identifies the table from which rows will be selected, *column_name* identifies a column whose value is selected, *search condition* is a Boolean expression containing references to host variables or host arrays, *subquery* is a SELECT statement with no INTO clause, and *indicator* stands for the following syntax:



For descriptions of the other keywords and parameters, refer to the *ORACLE7 Server SQL Language Reference Manual.*

Usage Notes

The *db_name* can be an undeclared identifier or a host variable. If *db_name* is a host variable, all database tables referenced by the SELECT statement must be defined in DECLARE TABLE statements.

When MODE=ANSI14, array SELECTS are not allowed; that is, you can reference host arrays in a SELECT statement only when MODE={ANSI | ANSI13 | ORACLE}. If one of the host variables in the INTO clause is an array, all must be arrays. The host arrays can have different dimensions, in which case the number of array elements processed is determined by comparing the dimension of the smallest host array in the SQL statement with the optional FOR-clause variable. The lesser value is used. Using host arrays in the WHERE clause is not allowed.

The cumulative number of rows selected is returned to the third element of SQLERRD in the SQLCA. However, if no rows meet the *search_condition,* none are selected and the "no data found" ORACLE error code is retuned to SQLCODE in the SQLCA.

The FOR clause is *not allowed in* SELECT statements. Instead, DECLARE a cursor and use the FOR clause in a FETCH statement, as follows:

```
EXEC SQL FOR :limit FETCH emp_cursor INTO . . .
```

Example

The following example illustrates the use of SELECT:

```
EXEC SQL SELECT ENAME , SAL + 100, JOB
    INTO :emp_name, :salary, :job_title
    FROM EMP WHERE EMPNO = :emp_number;
```

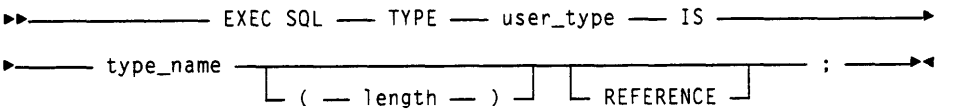
TYPE

Purpose

The TYPE statement assigns an ORACLE external datatype to a whole class of host variables by equivalencing the external datatype to a user-defined datatype. This is called *user-defined type equivalencing.*

Syntax

The following railroad diagram shows how to construct a TYPE statement



where *user_type* is a user-defined (not built-in) datatype identifier declared earlier inside or outside the Declare Section, *type_name* is the name of a valid external datatype such as RAW or STRING, *length* is an integer literal specifying a valid length in bytes, and the optional keyword REFERENCE denotes a pointer user-type.

Usage Notes

The TYPE statement is available only in Pro*C and Pro*Pascal.

The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by an n-byte data field, where *n* lies in the range 1.. 65,533. So, if *type_name is* VARCHAR or VARRAW, *user_type* must be at least three bytes long.

You must specify length unless *type_name* is NUMBER, VARNUM, ROWID, or DATE, in which case you cannot specify *length* because it is predefine. The value of *length* must be large enough to accommodate the external datatype.

When specifying *length,* if *type_name* is VARCHAR or VARRAW, use the maximum length of the data field only. The precompiled accounts for the length field.

With arrays, *length* must match exactly the buffer size required to hold the user-defined datatype. Also, ORACLE expects VARCHAR and VARRAW arrays to be word-aligned. So, when you equivalence an array type to the VARCHAR or VARRAW datatype, make sure that *length* + 2 is divisible by 4.

Examples

The following examples illustrate the use of TYPE:

Pro*C Example

```
struct  screen {short  len;
                char buff [4000];
               };
typedef  struct screen graphics;

EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL TYPE graphics IS VARRAW (4000);
    graphics  crt;  -- host variable of type graphics
    ...
EXEC SQL END DECLARE SECTION;
```

Another Pro*C Example

```
typedef  int * my_int;

EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL TYPE my_int IS INTEGER REFERENCE;
    my_int  ptr; -- pointer host variable of type my_int
    ...
EXEC SQL END DECLARE SECTION;
```

Pro*Pascal Example

```
Type
    OraDate = Record
                Cent, Year, Month, Day, Hour, Min, See: Byte
              End;

Var
    EXEC SQL BEGIN DECLARE SECTION;
        EXEC SQL TYPE OraDate IS DATE;
        Birthday: OraDate;  -- host variable of type OraDate
        ...
    EXEC SQL END DECLARE SECTION;
```

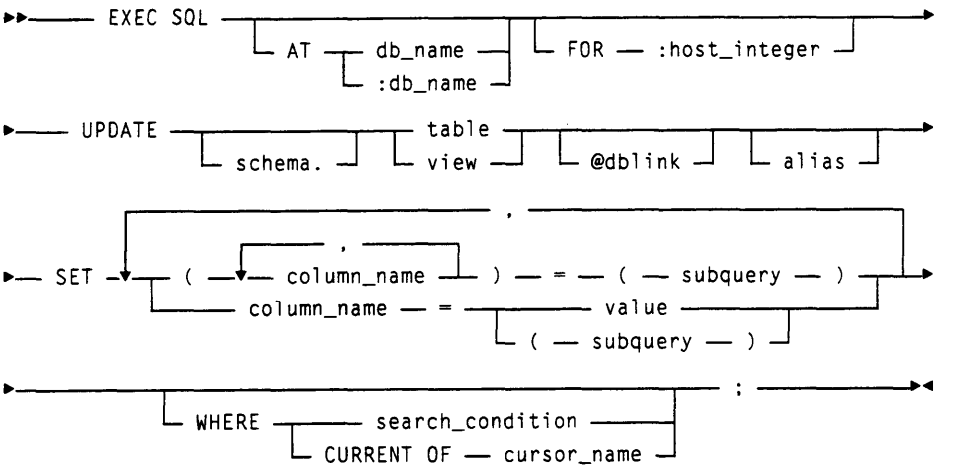
UPDATE

Purpose

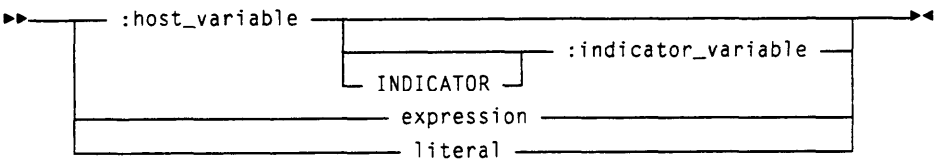
The UPDATE statement changes the values of specified columns in a table or view.

Syntax

The following railroad diagram shows how to construct an UPDATE statement



where *db_name* identifies a non-default connection, FOR *:host_integer* specifies the maximum number of host array elements processed, *table_name* identifies the table that will be updated, *alias is* another name assigned to the table, *column_name* identifies a column for which a value is provided, *subquery* is a SELECT statement with no INTO clause, *search condition* is a Boolean expression containing references to host variables or host arrays, CURRENT OF refers to the latest row processed by the FETCH statement, *cursor_name* refers to a cursor defined in a previous DECLARE CURSOR statement, and *value stands* for the following syntax:



Usage Notes

The *db_name* can be an undeclared identifier or a host variable. If *db_name* is a host variable, all database tables referenced by the UPDATE statement must be defined in DECLARE TABLE statements.

When MODE=ANSI14, array UPDATEs are *not* allowed; that is, you can reference host arrays in a UPDATE statement only when MODE={ANSI | ANSI13 | ORACLE]. If one of the host variables in the SET or WHERE clause is an array, all must be arrays. The host arrays can have different dimensions, in which case the number of array elements processed is determined by comparing the dimension of the *smallest* host array in the SQL statement with the optional FOR-clause variable. The lesser value is used. If the host variables in the WHERE clause are arrays, the UPDATE statement is executed once for each set of array elements.

The cumulative number of rows updated is returned to the third element of SQLERRD in the SQLCA. The number does *not* include rows processed by an update cascade.

If no rows meet the *search_condition*, none are updated, and the "no date found" ORACLE error code is returned to SQLCODE in the SQLCA.

If you omit the WHERE clause, the fifth element of SQLWARN in the SQLCA is set.

Several restrictions apply to the CURRENT OF clause; see the section "WHERE Clause" at the end of this chapter.

Examples

The following examples illustrate the use of UPDATE:

```
EXEC SQL UPDATE EMP
    SET SAL = :salary, JOB = :job_title
    WHERE EMPNO = :emp_number;
...
EXEC SQL UPDATE EMP
    SET SAL = (SELECT AVG ( SAL ) *1 .1 FROM EMP WHERE DEPTNO = 20 )
    WHERE EMPNO = :emp_number;
```

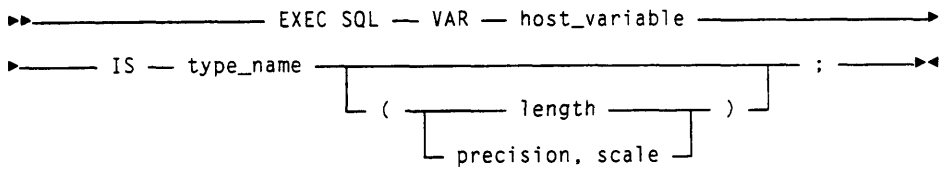
VAR

Purpose

The VAR statement assigns a specific ORACLE external datatype to an individual host variable, overriding the default datatype assignment. This is called *host variable equivalencing.*

Syntax

The following railroad diagram shows how to construct a VAR statement:



where *host_variable* is an input or output host variable (or host array) declared earlier in the Declare Section, *type_name* is the name of a valid external datatype such as STRING or RAW, *length* is an integer literal specifying a valid length in bytes, and *precision* and *scale* are integer literals specifying a precision and scale allowed by the equivalence external datatype.

Usage Notes

The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by a *n* -byte data field, where *n* lies in the range 1..65,533. So, if *type_name* is VARCHAR or VARRAW, *host_variable* must beat least three bytes long.

When *type_name* is DECIMAL or DISPLAY, you must specify *precision* and *scale* instead of *length.* When *type_name* is NUMBER, VARNUM, ROWID, or DATE, you cannot specify *length* because it is predefine. For other ORACLE external datatypes, *length* is optional. It defaults to the length of *host_variable.* The value of *length* must be large enough to accommodate the external datatype.

When specifying *length,* if *type_name is* VARCHAR or VARRAW, use the maximum length of the data field only. The precompiled ‘accounts for the length field.

You can specify a *precision* of 1..99 and a *scale* of-84..99. However, the maximum precision and scale of a database column are 38 and 127, respectively. So, if *precision* exceeds 38, you cannot INSERT the value of *host_variable* into a database column. On the other hand, if the scale of a column value exceeds 99, you cannot SELECT or FETCH the value into *host_variable.*

The *precision* and scale parameters are for COBOL and PL/I packed decimals or COBOL "USAGE IS DISPLAY' numbers. Specify *precision* and *scale* only when *type_name* is DECIMAL or DISPLAY.

Examples

The following examples illustrate the use of VAR:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    dept_name CHARACTER (15) ;   -- default datatype is CHAR
    EXEC SQL VAR dept_name IS STRING;   -- reset to STRING
    ...
    buffer     CHARACTER (200 ) ;   -- default datatype is CHAR
    EXEC SQL VAR buffer IS RAW (200);   -- reset to RAW
EXEC SQL END DECLARE SECTION;
```

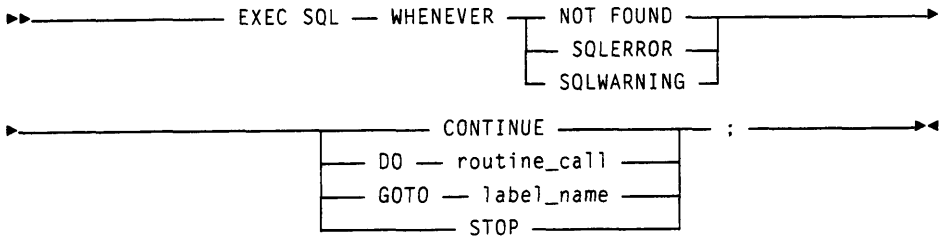
WHENEVER

Purpose

The WHENEVER statement specifies the action to be taken when an error or warning condition results from executing an embedded SQL statement.

Syntax

The following railroad diagram shows how to construct a WHENEVER statement



where *routine_call* is a host language call to an error handling routine and *label_name* labels the statement to which WHENEVER GOTO branches.

Usage Notes

If a SQL statement fails, you can have your program transfer control to an error handling routine. When the end of the routine is reached, control transfers to the statement that follows the failed SQL statement.

A routine is any functional program unit that can be invoked, such as a C function, COBOL paragraph, FORTRAN subroutine, Pascal procedure, or PL/I internal procedure. In this context, separately compiled programs such as COBOL subroutines and external PL/I procedures are *not* routines. The usual rules for entering and exiting a routine apply. However, passing parameters to the routine is not allowed. Furthermore, the routine cannot return a value.

In Pascal, the STOP action is illegal because Pascal has no equivalent command.

The scope of a WHENEVER statement is positional, not logical. That is, it tests all executable SQL statements that physically follow it in the source file, not in the flow of program logic. A WHENEVER statement stays in effect until superseded by another WHENEVER statement checking for the same condition.

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes an executable SQL statement, be sure to code WHENEVER SQLERROR CONTINUE before the SQL statement. That way, your program will not enter an infinite loop if the SQL statement fails with an error.

Declaring the SQLCA is optional when MODE={ANSI | ANSI14} but you cannot use the WHENEVER SQLWARNING statement without the SQLCA. So, if you want to use the WHENEVER SQLWARNING statement, you must declare the SQLCA.

Examples

The following examples illustrate the use of WHENEVER:

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...
EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
...
sqlerror:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
```

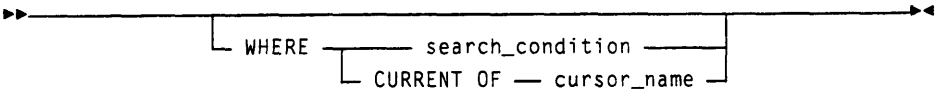
WHERE

Purpose

The optional WHERE clause specifies a condition under which rows are SELECTed, DELETEd, or UPDATEd.

Syntax

The following railroad diagram shows how to construct a WHERE clause:



where *search condition* is a Boolean expression containing references to host variables or host arrays, CURRENT OF refers to the latest row processed by a FETCH statement, and *cursor.name* refers to a cursor defined in a previous DECLARE CURSOR statement

Usage Notes

If you omit the WHERE clause, all rows in the table are processed. If you omit the WHERE clause in a DELETE or UPDATE statement, the fifth element of SQLWARN in the SQLCA is set.

If one of the host variables used in the WHERE clause is an array, all must be arrays. The FOR-clause variable specifies the number of array elements to be processed. Make sure the number is not larger than the smallest array dimension. Also, the number must be positive. If it is negative or zero, no rows are processed and ORACLE issues an error message.

When using the CURRENT OF clause, the named cursor must be open and positioned on a row. If no FETCH has been done or the cursor is not open, the CURRENT OF clause processes no rows. In addition, the following restrictions apply

- You cannot FETCH from a FOR UPDATE cursor after a COMMIT.
- You cannot reference multiple tables in an associated FOR UPDATE OF clause.
- You cannot use host arrays or dynamic SQL with the CURRENT OF clause.

Example

The following example illustrates the use of WHERE:

```
EXEC SQL INSERT INTO NEW_EMP ( EMPNO , SAL)
    SELECT EMPNO, SAL FROM EMP WHERE DEPTNO = : dept_number;
```

APPENDIX

C

ORACLE Reserved Words and Keywords

ORACLE Reserved Words

The following words are reserved by ORACLE. That is, they have a special meaning to ORACLE and so cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes.

```
ACCESS
ADD
ALL
ALTER
AND
ANY
ARRAYLEN
AS
ASC
AUDIT
BETWEEN
BY
CHAR
CHECK
CLUSTER
COLUMN
COMMENT
COMPRESS
CONNECT
CREATE
CURRENT
DATE
 DECIMAL
DEFAULT
DELETE
DESC
DISTINCT
DROP
ELSE
EXCLUSIVE
EXISTS
FILE
FLOAT
FOR
FROM
GRANT
GROUP
HAVING
IDENTIFIED
IMMEDIATE
IN
INCREMENT
INDEX
INITIAL
INSERT
INTEGER
INTERSECT
INTO
IS
LEVEL
LIKE
LOCK
LONG
MAXEXTENTS
MINUS
MODE
MODIFY
NOAUDIT
NOCOMPRESS
NOT
NOTFOUND
NOWAIT
NULL
NUMBER
OF
OFFLINE
ON
ONLINE
OPTION
OR
ORDER
PCTFREE
PRIOR
PRIVILEGES
PUBLIC
RAW
RENAME
RESOURCE
REVOKE
ROW
ROWID
ROWLABEL
ROWNUM
ROWS
SELECT
SESSION
SET
SHARE
SIZE
SMALLINT
SQLBUF
START
SUCCESSFUL
SYNONYM
SYSDATE
TABLE
THEN
TO
TRIGGER
UID
UNION
UNIQUE
UPDATE
USER
VALIDATE
VALUES
VARCHAR
VARCHAR2
VIEW
WHENEVER
WHERE
WITH
```

ORACLE Keywords

The following words also have a special meaning to ORACLE but are not reserved words and so can be redfined. However, some might eventually become reserved words.

ADMIN
AFTER
ALLOCATE
ANALYZE
ARCHIVE
ARCHIVELOG
AUTHORIZATION
BACKUP
BEGIN
BECOME
BEFORE
BLOCK
BODY
CACHE
CANCEL
CASCADE
CHANGE
CHARACTER
CHECKPOINT
CLOSE
COMMIT
COMPILE
CONSTRAINT
CONSTRAINTS
CONTENTS
CONTINUE
CONTROLFILE
CYCLE
DATABASE
DATAFILE
DBA
DEC
DECLARE
DISABLE
DISMOUNT
DOUBLE
DUMP
EACH
ENABLE
END
ESCAPE
EVENTS
EXCEPT
EXCEPTIONS
EXPLAIN
EXECUTE
EXTENT
EXTERNALLY
FLUSH
FREELIST
FREELISTS
FORCE
FOREIGN
FUNCTION
GROUPS
INCLUDING
INDICATOR
INITRANS
INSTANCE
INT
KEY
LAYER
LINK
LISTS
LOGFILE
MANAGE
MANUAL
MAX
MAXARCHLOGS
MAXDATAFILES
MAXINSTANCES
MAXLOGFILES
MAXLOGMEMBERS
MAXTRANS
MAXVALUE
MIN
MINEXTENTS
MINVALUE
MOUNT
NEXT
NEW
NOARCHIVELOG
NOCACHE
NOCYCLE
NOMAXVALUE
NOMINVALUE
NONE
NOORDER
NORESETLOGS
NORMAL
NOSORT
NUMERIC
ONLY
OBJNO
OFF
OLD
ONLY
OPCODE
OPTIMAL
OPEN
OWN
PACKAGE
PARALLEL
PCTINCREASE
PCTUSED
PLAN
PRIMARY
PRIVATE
PROCEDURE
PROFILE
QUOTA
RBA
READ
REAL
RECOVER
REFERENCES
REFERENCING
RESETLOGS
RESTRICTED
REUSE

```
ROLE
ROLES
ROLLBACK
SAVEPOINT
SCHEMA
SCN
SEGMENT
SEQUENCE
SHARED
SNAPSHOT
SOME
SORT
STATEMENT_ID
STATISTICS
STOP
STORAGE
SWITCH
SYSTEM
TABLES
TABLESPACE
TEMPORARY
THREAD
TIME
TRACING
TRANSACTION
TRIGGERS
TRUNCATE
UNDER
UNLIMITED
UNTIL
USE
USING
WHEN
WRITE
WORK
```

APPENDIX

D

ERROR MESSAGES

This appendix lists error messages you might see when running the ORACLE Precompilers. Also listed are error messages that the ORACLE runtime library might return to the SQLCA. For each error, the probable cause and corrective action are given.

Coping with Error Messages

The ORACLE Recompilers issue various warning and error messages. For example, you might see the following error code and message:

```
PCC-U-0036: No input file name specified
```

The error code consists of a prefix, severity code, and sequence number. The prefix PCC shows that the error was issued by an ORACLE Precompiled. There are four severity codes; the following table gives their meanings:

Code	*Meaning*
W	Warning-despite an error, a compilable output file was created.
S	Severe error-despite an error, an output file was created. However, it might not be compilable.
F	Fatal error-no output file was created because of an internal problem or because a resource (such as memory) was unavailable or ran out.
U	Unrecoverable error-no output file was created because an input requirement was not met.

Recognizing Message Inserts

To help you find and fix errors, the ORACLE Precompilers insert object names and numbers in some error messages. In this appendix, these names and numbers are represented by "X" and N respectively. For example, the error message listed as

```
PCC-0032 : Invalid option "X'
```

might actually appear as

```
PCC-U-0032: Invalid option "MADE"
```

Calling Oracle Customer Support

Some error messages recommend calling Oracle Customer Support for assistance. For the procedure to follow, see the ORACLE7 Server ***Messages and Codes Manual.***

ORACLE Precompiled Error Messages

The following error messages might be issued at precompiled time by the ORACLE Precompilers.

PCC-0001: Unable to open file "X"

Cause: The precompiled was unable to open a temporary file for internal use. There might be insufficient disk space, too many open files, or read-only protection on the output directory.

Action: Make sure there is enough disk space, that the limit for open files is set high enough (check with your system manager), and that protection on the directory allows opening a file for writing.

PCC-0002: Invalid syntax at column N in line N of file "X"

Cause: There is a syntax error in an EXEC statement or the statement is not properly terminated.

Action: Correct the syntax of the EXEC statement. If the error occurred at the end of the input file, make sure the last EXEC statement is properly terminated.

PCC-0003: Invalid SQL Identifier at column N in line N of file "X"

Cause: The symbol in a conditional precompilation statement (such as EXEC ORACLE IFDEF) is invalid, or the name of a SQL descriptor, statement, or cursor is invalid or was not properly declared.

Action: Check the statement syntax and your spelling of the identifier, and make sure you did not use a reserved word. If necessary, define the identifier in a variable declaration or DECLARE statement ahead of the line in error.

PCC-0004: Mismatched IF/ELSE/ENDIF block at line N in file "X"

Cause: There is an EXEC ORACLE ELSE or EXEC ORACLE ENDIF statement without a matching EXEC ORACLE IFDEF statement.

Action: Add the missing EXEC ORACLE IFDEF statement, or delete or move the EXEC ORACLE ELSE or EXEC ORACLE ENDIF statement.

PCC-0005: Unsupported datatype in line N of file "X"

Cause: A host variable defined in the Declare Section has an unsupported datatype or has a scale or precision outside the supported range.

Action: Redefine the host variable using a supported datatype. Make sure the scale and precision of a numeric variable are in the accepted range.

PCC-0007: Invalid WHENEVER condition at column N in line N of file "X"

Cause: A condition other than SQLERROR, SQLWARNING, or NOT FOUND was specified in an EXEC SQL WHENEVER statement, or one of these was used, but spelled incorrectly.

Action: Correct the spelling of the WHENEVER condition, or use a host-language IF statement to test the special condition.

PCC-0008: Invalid WHENEVER action at column N in line N of file "X"

Cause: An action other than CONTINUE, GOTO, or STOP was specified in an EXEC SQL WHENEVER statement, or one of these was spelled incorrectly. Or, the host language does not allow the action (STOP is illegal in Pro*Pascal programs), or a GOTO label is invalid.

Action: Make sure your host language allows the specified WHENEVER action. If necessary, correct the spelling of the WHENEVER action or correct the GOTO label.

PCC-0009: Invalid host variable at column N in line N of file "X"

Cause: A host variable used in an EXEC SQL statement was not declared in the Declare Section or has an unsupported datatype.

Action: Declare the host variable in the Declare Section, making sure it has one of the supported datatypes.

PCC-0010: Statement out of place at line N in file "X"

Cause: An EXEC statement was not placed properly in the host program. For example, there might be a data manipulation statement in the Declare Section. In a Pro*COBOL program, the Declare Section might be outside the WORKING-STORAGE SECTION.

Action: Remove or relocate the statement.

PCC-0011: Already in a Declare Section at line N in file "X"

Cause: A BEGIN DECLARE SECTION statement was found inside a Declare Section.

Action: Remove the extra BEGIN DECLARE SECTION statement.

PCC-0012: Not in a Declare Section at line N in file "X"

Cause: An END DECLARE SECTION statement without a matching BEGIN DECLARE SECTION statement was found. Either the BEGIN DECLARE SECTION statement is missing or misspelled or the END DECLARE SECTION statement is an extra.

Action: Add or correct the BEGIN DECLARE SECTION statement or remove the extra END DECLARE SECTION statement.

PCC-0013: Unable to open INCLUDE file "X" at line N in file "X"

Cause: The precompiled was unable to open the input file specified in the INCLUDE statement. Some possible causes follow:

- The filename is misspelled.
- The file does not exist.
- The search path to the file is incorrect.
- You have insufficient file access privileges.
- Another user has locked the file.
- There is not enough disk space.
- There are too many open files.

Action: Make sure the file exists, that the search path to the file is correct, that you have sufficient privileges to access the file, and that it is not locked by another user. Also make sure there is enough disk space and that the limit for open files is set high enough (check with your system manager).

PCC-0014: Undeclared SQL identifier "X" at line N in file "X"

Cause: The name of a descriptor, statement, or cursor was not declared or is misspelled.

Action: Add or correct the descriptor, statement, or cursor declaration.

PCC-0015: Unrecognized host language syntax ignored at line N in file "X"

Cause: The host language syntax used to define a host variable in the Declare Section is incorrect.

Action: Check the syntax and your spelling, then correct the declaration.

PCC-0016: Unable to open a cursor at line N in file "X"

Cause: The syntax in a SQL statement is faulty. The precompiled was expecting a host variable but found something else.

Action: Check the syntax and your spelling, then correct the SQL statement.

PCC-0017: Unable to parse statement at line N in file "X"

Cause: There is a syntax error in an array declaration. The precompiled was expecting a right bracket (]) but found something else.

Action: Check the syntax, then correct the array declaration.

PCC-0018: Expected "X", but found "X" at line N in file "X"

Cause: The syntax in a SQL statement is faulty. The precompiled found an unexpected or illegal token.

Action Check the syntax and your spelling, then correct the SQL statement.

PCC-0019: Unable to obtain bind variables at line N in file "X"

Cause: The precompiled was unable to find information about an input host variable (bind variable) used in a SQL statement.

Action: Make sure the input host variable is declared in the Declare Section and used properly in the SQL statement.

PCC-0020: Unable to obtain define variables at line N in file "X"

Cause: The precompiled was unable to find information about an output host variable (define variable) used in a SQL statement.

Action: Make sure the output host variable is declared in the Declare Section and used properly in the SQL statement.

PCC-0021: ORACLE Error: N

Cause: An internal ORACLE error occurred.

Action: Look up the ORACLE error number in the *ORACLE7 Server Messages and Codes Manual*, then take the recommended action.

PCC-0022: Out of space - unable to allocate N bytes

Cause: The precompiled process ran out of memory.

Action: Allocate more memory to the process, then retry.

PCC-0023: Unable to log off ORACLE

Cause: An ORACLE connection error occurred while the precompiled was trying to log off, probably because ORACLE has been shut down.

Action: Make sure ORACLE is available, then retry.

PCC-0024: Indicator variable "X" has wrong type or length at line N in file "X"

Cause: An indicator variable was not declared in the Declare Section as a 2-byte integer. Indicator variables must be defined as 2-byte integers.

Action: Redefine the indicator variable as a 2-byte integer.

PCC-0025: Undeclared indicator variable "X" at line N in file "X"

Cause: The name of an indicator variable used in a SQL statement was not declared in the Declare Section or is misspelled.

Action: Add or correct the indicator variable declaration.

PCC-0026: Undeclared host variable "X" at line N in file "X"

Cause: The name of a host variable used in a SQL statement was not declared in the Declare Section or is misspelled.

Action: Add or correct the host variable declaration.

PCC-0027: Redeclared SQL identifier "X" at line N in file "X"

Cause: **The** name of a SQL descriptor, statement, or cursor was redeclared (that is, declared twice).

Action: Check your spelling of the identifier, then, if necessary, remove the extra declaration.

PCC-0028: Option "X" not legal as EXEC ORACLE OPTION

Cause: A precompiled option was specified inline in an EXEC ORACLE statement instead of on the command line. Some options can be specified only on the command line. For example, you cannot specify INAME inline.

Action: Respecify the precompiled option on the command line instead of in an EXEC ORACLE statement. To see an online display of the precompiled options, enter the precompiled command (with no options) at your operating system prompt.

PCC-0029: Ambiguous option "X"

Cause: The name of a precompiled option was abbreviated ambiguously. For example, MAX= might refer to MAXLITERAL or MAXOPENCURSORS.

Action: Respecify the full option name or an unambiguous abbreviation. To see an online display of the precompiled options, enter the precompiled command (with no options) at your operating system prompt.

PCC-0031: Invalid value given for option "X"

Cause: A precompiled option has an invalid operand, probably because the operand value is misspelled (LTYPE=HORT, for example) or illegal (PAGELEN=-55, for example).

Action: Check the operand, making sure it is spelled correctly and within the legal range.

PCC-0032: Invalid option "X"

Cause: The precompiled found an invalid precompiled option name. Some possible causes follow:

- The option name is misspelled.
- The specified option does not exist.
- The equal sign (=) between the option name and operand is missing or has space around it.
- The name of the input file was not preceded by INAME=.

Action: Make sure the option exits and that its name is spelled correctly. To see an online display of the precompiled options, enter the precompiled command (with no options) at your operating system prompt. Make sure there is an equal sign between the option name and operand.

PCC-0033: Missing operand for option "X"

Cause: No operand was specified for a precompiled option. Either the operand is missing or there is space around the equal sign (as in LTYPE =SHORT).

Action: Make sure each option has an operand and that there is no space around the equal sign.

PCC-0035: No host language specified

Cause: The precompiled was unable to determine the host language of the input file. If you use anon-standard input file extension when specifying the INAME precompiled option, you must also specify the HOST option. However, most operating systems are set up to prevent this error (by automatically specifying the HOST option when a precompiled is invoked).

Action: Either add the standard input file extension or specify the HOST option. If your operating system is supposed to prevent this error, call Oracle Customer Support for assistance.

PCC-0036: No input file name specified

Cause: The input file was not specified on the command line.

Action: Use the INAME command-line option to specify the input file.

PCC-0037: Unable to log on to ORACLE with "X". ORACLE error number: N

Cause: The precompiled was unable to logon to ORACLE with the specified username and password. An ORACLE error with given number occurred when the logon was attempted.

Action: Look up the ORACLE error number in the *ORACLE7 Server Messages and Codes Manual*, then take the recommended action.

PCC-0038: Unable to open a cursor

Cause: This is an internal error message not normally issued.

Action: Call Oracle Customer Support for assistance. If your application does not require syntactic or semantic checking of SQL statements and does not use PL/SQL, specify SQLCHECK=NONE on the command line.

PCC-0039: Unable to open input file "X"

Cause: The precompiled was unable to open the input file specified by the INAME precompiled option. Some possible causes follow:

- The filename is misspelled.
- The file does not exist.
- The search path to the file is incorrect.
- You have insufficient file access privileges.
- Another user has locked the file.
- There is not enough disk space.
- There are too many open files.

Action: Make sure the file exists, that the search path to the file is correct, that you have sufficient privileges to access the file, and that it is not locked by another user. Also make sure there is enough disk space and that the limit for open files is set high enough (check with your system manager).

PCC-0040: Unable to open listing file "X"

Cause: The precompiled was unable to open the listing file specified by the LNAME precompiled option. Some possible causes follow:

- The filename is misspelled.
- The file does not exist.
- The search path to the file is incorrect.
- You have insufficient file access privileges.
- Another user has locked the file.
- There is not enough disk space.
- There are too many open files.

Action: Make sure the file exists, that the search path to the file is correct, that you have sufficient privileges to access the file, and that it is not locked by another user. Also make sure there is enough disk space and that the limit for open files is set high enough (check with your system manager). If you do not need a listing file, specify LTYPE=NONE on the command line.

PCC-0041: Unable to open output file "X"

Cause: The precompiled was unable to open the output file specified by the ONAME precompiled option. Some possible causes follow:

- The filename is misspelled.
- The file does not exist.
- The search path to the file is incorrect.
- You have insufficient file access privileges.
- Another user has locked the file.
- There is not enough disk space.
- There are too many open files.

Action: Make sure the file exists, that the search path to the file is correct, that you have sufficient privileges to access the file, and that it is not locked by another user. Also make sure there is enough disk space and that the limit for open files is set high enough (check with your system manager).

PCC-0042: Must include SQLCA file when MODE=ANSI and WHENEVER SQLWARNING used

Cause: When MODE={ANSI | ANSI14}, you tried to use the WHENEVER SQLWARNING statement without declaring the SQLCA. When MODE={ANSI | ANSI14}, declaring the SQLCA is optional, but to use the WHENEVER SQLWARNING statement, you must declare the SQLCA.

Action: Remove all WHENEVER SQLWARNING statements from your program, or declare the SQLCA by hardcoding it or copying it into you program with the INCLUDE statement.

PCC-0044: Array size mismatch in INTO/USING. Minimum is "X"(N:N)

Cause: The size of an array variable in an INTO / USING clause ***is*** too small for the number of rows processed.

Action: Declare all array variables in the lNTO / USING clause to have at least the minimum dimension given.

PCC-0045: "X" clause inappropriate at line N in file "X". Ignored

Cause: There is a misplaced clause at the end of an EXEC SQL statement (an AT clause at the end of a SELECT statement, for example). Or, the action specified in a FOR clause is invalid (for example, FOR :loop INTO **. ..).**

Action **Check the** statement syntax, then relocate or correct the misplaced or invalid clause.

PCC-0047: Unterminated comment/string constant beginning near line N in file "X"

Cause: A string constant is missing an ending quote, or a comment is missing an ending delimiter.

Action: Make sure all comments are delimited and all string constants are enclosed by quotes.

PCC-0048: PRO* configured without ORACLE. INLINE=NO ignored

Cause: Currently the ORACLE Precompilers generate inline code, not access modules. Thus, the INLINE=NO coremand-line option is ignored. However, future versions of the ORACLE Precompilers will be able to generate access modules.

Action: Do not specify the INLINE option.

PCC-0050: Unable to generate descriptor in program unit ending line N in file "X"

Cause: Part of a descriptor was generated incorrectly, or the precompiled was unable to generate a descriptor in a program unit terminated by an end-of-file.

Action Call Oracle Customer Support for assistance.

PCC-0051: Size of VARCHAR "X" at N is larger than 65533 at line N in file "X"

Cause: The declared size of a VARCHAR host variable exceeds the precompiled limit of 65533 bytes.

Action: Check the Declare Section, making sure the size of each VARCHAR variable does not exceed 65533 bytes.

PCC-0053: FOR variable "X" is invalid type at line N in file "X"

Cause: The count variable in a FOR clause has the wrong datatype. The datatype must be NUMBER or LONG (or compatible with NUMBER or LONG).

Action Check the declaration and make sure the count variable has a datatype of NUMBER or LONG (or a compatible ORACLE or host-language datatype).

PCC-0054: Expected end-of-statement at column N in line N of file "X"

Cause: The precompiled expected to find a statement terminator at the end of an EXEC statement, but found something else. This can happen if you embed tabs in your source code (because the precompiled has no way of knowing how many spaces a tab represents).

Action: If you find tabs embedded in the source code, replace them with spaces. Check the statement syntax and make sure each EXEC statement has a terminator. For embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statements and for embedded PL/SQL blocks, make sure the statement terminator is END-EXEC.

PCC-0055: Array "X" not allowed as bind variable at line N in file "X"

Cause: A host array was used as a bind (input) variable in the WHERE clause of a SELECT' statement. This is not allowed.

Action: Remove the host array or replace the it with a simple host variable.

PCC-0056: FOR clause not allowed in SELECT statement at line N in file "X"

Cause: **This** message warns that the FOR :loop SELECT . . . construct will eventually be disallowed.

Action: Keep in mind that, eventually, code containing this construct will have to be modified.

PCC-0060: Both CURSOR and STATEMENT have AT clauses at line N of file "X"

Cause: Two AT clauses, one in a DECLARE STATEMENT statement, the other in a DECLARE CURSOR statement, pertain to the same SQL statement. You can specify the AT clause with either DECLARE STATEMENT or DECLARE CURSOR, but not with both.

Action: Remove the AT clause from one of the statements.

PCC-0061: Error at line N, column N. PLS-N: "X"

Cause: The precompiled found a syntax error while parsing an embedded SQL statement or PL/SQL block. If the error was found in a PL/SQL block, the PL/SQL error message code and text are given.

Action Correct the syntax error. For more information about a PL/SQL syntax error, look it up in *ORACLE7 Server Messages and Codes Manual.*

PCC-0062: Must use option SQLCHECK=SEMANTICS when there is embedded PL/SQL

Cause: The precompiled tried to parse an embedded PL/SQL block when SQLCHECK={SYNTAX | NONE}. PL/SQL blocks can be parsed only when you specify SQLCHECK=SEMANTICS.

Action: Remove the PL/SQL block or specify SQLCHECK=SEMANTICS.

PCC-0063: Reached end of file "X" before end-of-statement at line N

Cause: The precompiled encountered an end-of-file while parsing a PL/SQL block.

Action: Add the appropriate statement terminator (;) or end-of-block statement (END;) to the PL/SQL block.

PCC-0064: All uses of a given host variable must use identical indicator variables

Cause: Two or more occurrences of a host variable in an EXEC SQL statement are associated with different indicator variables. This is not allowed.

Action: Rename the indicator variables so that each occurrence of the host variable is associated with the same indicator variable.

PCC-0065: USERID required, but not specified

Cause: You specified the SQLCHECK=SEMANTICS option, but failed to specify the USERID option on the command line.

Action: Specify USERID=username/password, or enter a username and password when prompted. Or, specify SQLCHECK={SYNTAX | NONE}.

PCC-0066: USERID only used when SQLCHECK=SEMANTICS, USERID ignored

Cause: You specified the USERID option when SQLCHECK={SYNTAX | NONE}. This is unnecessary.

Action: Specify the USERID option only when SQLCHECK=SEMANTICS.

PCC-0067: IRECLEN exceeded. Line N in file "X" truncated

Cause: While reading the input file, the precompiled found a line longer than IRECLEN.

Action: Either shorten the input line or specify a larger IRECLEN value on the command line.

PCC-0068: Host and indicator variables may not have the same name

Cause: In an EXEC SQL statement, an indicator variable has the same name as a host variable. The names of a host variable and its associated indicator variable must be different. Also, an indicator variable cannot be used as a host variable.

Action: Rename the host or indicator variable.

PCC-0069: Host variable "X" has unsupported datatype at line N in file "X"

Cause: A host variable has an unsupported datatype. For a list of supported datatypes, see your *Supplement to the ORACLE Precompilers Guide.*

Action: Redefine the host variable in the Declare Section, giving it a supported datatype.

PCC-0070: Illegal syntax. Exponential value in SQL statement "X"

Cause: The precompiled found a syntax error while parsing a number coded in scientific notation. The precompiled expected to find a signed integer following the exponentiation indicator (E), but found something else.

Action: Reformat the number correctly.

PCC-0072: Input filename length exceeds 14 characters

Cause: On some platforms, the maximum length of a file name is 14 characters. The file name you specified exceeds the maximum length.

Action Use a filename of 14 or fewer characters.

PCC-0073: Cursor is declared but never OPENed at line N in file "X"

Cause: You DECLAREd a cursor, but did not code an OPEN statement for it. This is only an informational message.

Action: Remove the cursor declaration or code an OPEN statement for the cursor.

PCC-0074: FIPS warning: Multiply defined host variable in line N of file "X"

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, you reused the name of a global host variable to declare a local host variable. This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. But, for ANSI/ISO compliance, do not reuse the names of global host variables to declare local host variables.

PCC-0075: ":" expected before indicator variable

Cause: An indicator variable is not prefixed with a colon as required.

Action: Prefix a colon to the indicator variable in question.

PCC-0076: DISPLAY type must be SIGN LEADING SEPARATE

Cause: This message is normally issued only by Pro*COBOL. DISPLAY SIGN LEADING SEPARATE is the only DISPLAY type supported by Pro*COBOL.

Action: Check your spelling of the variable declaration. If necessary, remove the reference to the unsupported DISPLAY type.

PCC-0077: Colon usage with numeric label in WHENEVER statement is not ANSI

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, a numeric WHENEVER . . . GOTO label was prefixed with a colon. For example, you might have coded

```
EXEC SQL WHENEVER SQLERROR GOTO :99;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. But, for ANSI/ISO compliance, prefix alphanumeric (but not numeric) WHENEVER . . . GOTO labels with a colon.

PCC-0078: FIPS warning: Invalid ANSI SQL identifier

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, the name you gave to a host variable

- is longer than 18 characters,
- does not begin with a letter, or
- contains consecutive or trailing underscores

In the following Pro*C example, the host variable name is 19 characters long and therefore non-compliant

```
EXEC SQL BEGIN DECLARE SECTION;
    int  department_location;   -- not ANSI-compliant
    ...
EXEC SQL END DECLARE SECTION;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. But, for ANSI/ISO compliance, change the host variable name so that it is ≤ 18 characters long, begins with a letter, and does not contain consecutive or trailing underscores.

PCC-0079: ANSI requires colon on label in WHENEVER statement

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. specifically, an alphanumeric WHENEVER . . . GOTO label was not prefixed with a colon. For example, you might have coded

```
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
```

This message is only a warning issued by the FIPS Flagger when FIPs=YES.

Action: None required. But, for ANSI/ISO compliance, prefix alphanumeric (but not numeric) WHENEVER . . . GOTO labels with a colon.

PCC-0080: TYPE identifier already TYPEd

Cause: The identifier being TYPEd in an EXEC SQL TYPE statement appeared in a previous EXEC SQL TYPE statement. A given identifier can appear in only one EXEC SQL TYPE statement.

Action: Check your spelling of the identifiers. Use different identifiers in the EXEC SQL TYPE statements, or remove one of the EXEC SQL TYPE statements.

PCC-008k: Scale specification not allowed for given datatype

Cause: The ORACLE external datatype referenced in an EXEC SQL VAR or EXEC SQL TYPE statement does not allow a scale specification.

Action: Check the precision specification and remove the scale specification.

PCC-0082: Length and scale specifications must bean integer

Cause: You used a floating point number or a non-number to specify a length or scale. Only integers can be used.

Action: Correct or remove the length and / or scale specification.

PCC-0083: Bind and define variables not allowed in CREATE statement

Cause: Host variables cannot appear in a CREATE statement. If the makeup of a CREATE statement cannot be known until run time, you must use dynamic SQL to execute it. That is, your program must acceptor build the CREATE statement at run time, store it in a host string, then EXECUTE it.

Action: Correct or remove the erroneous CREATE statement.

PCC-0085: Error writing to file "X"

Cause: The precompiled was unable to write to the named output file. Some possible causes follow:

- You have insufficient file access privileges.
- Another user has locked the file.
- There is not enough disk space.
- There are too many open files.

Action: Make sure that you have sufficient privileges to access the file and that it is not locked by another user. Also make sure there is enough disk space and that the limit for open files is set high enough (check with your system manager).

PCC-0086: Source file 'X" has zero length

Cause: The source file you specified on the command line contains no code. Consequently, there is nothing for the precompiled to process.

Action: Specify a valid source file containing embedded SQL statements.

PCC-008R: EXEC SQL TYPE statement not allowed for this host language

Cause: You used the EXEC SQL TYPE statement with a host language that does not support user-defined datatype equivalencing. This feature is available only in Pro*C and Pro*Pascal.

Action: Remove the offending EXEC SQL TYPE statement.

PCC-0088: User defined type identifier expected

Cause: The user-defined datatype name in an EXEC SQL TYPE statement is missing or misspelled, is a reserved word, is not a legal identifier in your host language, or conflicts with a base datatype in that language.

Action: Check your spelling of the user-defined datatype name. If necessary, declare a valid user-defined datatype. User-defined datatype equivalencing is available only in Pro*C and Pro*Pascal.

PCC-0089: Invalid ORACLE TYPE specification

Cause: The ORACLE external datatype name in an EXEC SQL TYPE or EXEC SQL VAR statement is missing or misspelled.

Action: Check your spelling of the external datatype name. If necessary, supply the missing datatype name.

PCC-0090: Precision/scale specification must be given for DECIMAL datatype

Cause: You omitted a precision and/or scale specification for the ORACLE external datatype DECIMAL in an EXEC SQL TYPE or EXEC SQL VAR statement.

Action: Add the precision and/or scale specification to the EXEC SQL TYPE or EXEC SQL VAR statement.

PCC-0091: TYPE statement requires format specification for this ORACLE datatype

Cause: You omitted a length, precision, and/or scale specification for an ORACLE external datatype in an EXEC SQL TYPE or EXEC SQL VAR statement.

Action: Add the length, precision, and/or scale specification for the external datatype to the EXEC SQL TYPE or EXEC SQL VAR statement.

PCC-0092: Length and/or scale incompatible with specified ORACLE datatype

Cause: You specified an invalid length or scale for an ORACLE external datatype in an EXEC SQL TYPE or EXEC SQL VAR statement.

Action: Make sure to specify a length that is large enough to accommodate the external datatype. If you specify a scale, make sure it lies in the range -84..99.

PCC-0093: Invalid or obsolete option, ignored

Cause: The precompiled found an option available in a prior version or different host language, but not in your version or host language.

Action: Remove the option specification.

PCC-0094: Array length for char[n] datatype must be=> 2

Cause: When MODE= {ANSI | ANSI14}, you specified a length of less than 2 characters for a **char[n]** host variable. When MODE= {ANSI | ANSI14}, the length must beat least 2 characters. This message is issued only by the Pro*C Precompiled.

Action: Correct the declaration so that it specifies a length of at least 2 characters.

PCC-0095: Missing PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement

Cause: FORTRAN source files are expected to have at least one PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement, which the precompiled uses to detect the beginning of a routine or compilation unit.

Action: Add one of these statements to the source file.

PCC-0096: Array FETCH not allowed for MODE=ANSI14

Cause: When MODE=ANSI14, you attempted an array SELECT or FETCH. However, array operations are not allowed when MODE=ANSI14.

Action: If you must specify MODE=ANSI14, place the SELECT or FETCH statement in a host-language loop instead of using the array interface.

PCC-0097: Use of DECIMAL and DISPLAY types allowed only for COBOL and PLI

Cause: You used the DECIMAL or DISPLAY external datatype in an EXEC SQL VAR statement with an ORACLE Precompiled other than Pro*COBOL or Pro*PL/I. These external datatypes are available only in Pro*COBOL and Pro*PL/I.

Action: Remove the reference to the DECIMAL or DISPLAY external datatype from the EXEC SQL VAR statement

PCC-0098: Scale specification cannot be used in this context

Cause: In a Pro*C, Pro*FORTRAN, or Pro*Pascal program, you cannot specify scale in an EXEC SQL TYPE or EXEC SQL VAR statement in the current context.

Action: Remove the scale specification from the EXEC SQL TYPE or EXEC SQL VAR statement.

Pee-0099: Length cannot be given for types ROWID and DATE

Cause: You specified a length for the ROWID or DATE external datatype in an EXEC SQL TYPE or EXEC SQL VAR statement. This is unnecessary because those types are fixed-length.

Action: Remove the length specification from the EXEC SQL TYPE or EXEC SQL VAR statement.

Pcc-olO(k: Non integer label is not ANSI

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, anon-integer WHENEVER . . . GOTO label was in a Pro*Pascal program. For example, you might have coded

```
EXEC SQL WHENEVER NOT FOUND GOTO 5.0 ;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Actiom: None required. However, for ANSI/ISO compliance, use only integer WHENEVER . . . GOTO labels in a Pro*Pascal program.

PCC-0101: Lower case 'e' in floating point number is not ANSI

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, a lower-case 'e' was used in scientific notation. For example, you might have coded

```
maxnum = 10e38;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, use an upper-case 'E' in scientific notation.

PCC-0102: FOR UPDATE is an Oracle extension

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, the FOR UPDATE OF clause was used in a cursor declaration. For example, you might have coded

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME , SAL FROM EMP WHERE DEPTNO = : dept_number
        FOR UPDATE OF SAL;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use the FOR UPDATE OF clause.

PCC-0103 AT clause is an Oracle extension

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, the AT <db_name> clause was used in a SQL statement. For example, you might have coded

```
EXEC SQL AT oracle3 COMMIT RELEASE;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use the AT <db_name> clause.

PCC-0104: FOR clause is an Oracle extension

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, the FOR clause was used in an array-processing SQL statement. For example, you might have coded

```
EXEC SQL FOR : limit INSERT INTO EMP ( EMPNO, JOB, DEPTNO )
    VALUES ( : emp_number, : job_title, : dept_number) ;
```

This message is only a warning issued by the FIPS Flagger when -FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use the FOR clause.

PCC-0105: Keyword WORK required here by ANSI

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, the keyword WORK was used in a COMMIT or ROLLBACK statement. For example, you might have coded

```
EXEC SQL COMMIT WORK RELEASE;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use the keyword WORK.

PCC-0106: RELEASE is an Oracle extension to the COMMIT and ROLLBACK statements

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, the parameter RELEASE was used in a COMMIT or ROLLBACK statement. For example, you might have coded

```
EXEC SQL ROLLBACK RELEASE;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use the parameter RELEASE.

PCC-0107: The CONNECT statement is Oracle implementation dependent

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, the CONNECT statement was used to logon to ORACLE. For example, you might have coded

```
EXEC SQL CONNECT : username IDENTIFIED BY : password;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required.

PCC-0108: This statement is not supported by ANSI

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, a nonconforming SQL statement such as PREPARE was used. For example, you might have coded

```
EXEC SQL PREPARE sql_statement FROM : sql_string;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use the nonconforming SQL statement.

PCC-0109: Dynamic SQL and PL/SQL are an Oracle extensions to ANSI SQL

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, dynamic SQL or embedded PL/SQL was used. For example, you might have coded

```
EXEC SQL EXECUTE
    BEGIN
        SELECT . . .
        ...
    END;
END-EXEC;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use dynamic SQL or embedded PL/SQL.

PCC-0110: Oracle extension to the WHENEVER statement

Cause: You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, a nonconforming keyword such as NOTFOUND, STOP, RAISE, or DO was used in the WHENEVER statement. (Note that NOT FOUND is ANSI-compliant.) For example, you might have coded

```
EXEC SQL WHENEVER SQLERROR STOP;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use the nonconforming keyword.

PCC-0111 SQLCHECK value in EXEC ORACLE statement exceeds command line value

Cause: You entered the SQLCHECK option inline specifying a level of checking higher than the level you specified (or accepted by default) on the command line. This is not allowed. For example, if you specify SQLCHECK={SYNTAX | LIMITED} on the command line, you cannot specify SQLCHECK={SEMANTICS | FULL} inline.

This message is only a warning; the precompiled ignores the inline value and continues processing.

Action: Revise the EXEC ORACLE statement or specify a lower level of checking on the command line.

PCC-0112: Datatype not supported by ANSI

Cause You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, a pointer or nonconforming datatype such as VARCHAR was used. For example, you might have coded

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR username [20] ;
    ...
EXEC SQL END DECLARE SECTION;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. However, for ANSI/ISO compliance, do not use pointers or nonconforming datatypes.

PCC-0113: Value of DBMS option invalid with given value of MODE option

Cause: When MODE= {ANSI14 | ANSI13}, you specified DBMS=V7, or when MODE=ANSI, you specified DBMS=V6. These option settings are incompatible. Note that the DBMS option is available only with Version 1.5 of the ORACLE Precompilers.

Action: With DBMS=V7, instead of MODE= {ANSI14 | MODE=ANSI13}, you can specify MODE= {ANSI | MODE=ORACLE}. With DBMS=V6, instead of MODE= {ANSI14 | MODE=ANSI13}, you can specify MODE= {ANSI14 | MODE=ANSI13 | ORACLE} but MODE=ORACLE is recommended.

PCC-0114: Length spec required in EXEC SQL VAR statements for VARxxx types

Cause: In a EXEC SQL VAR statement, a VARCHAR or VARRAW external datatype was specified without a length. Unlike other types, the VARCHAR and VARRAW types have a 2-byte length field followed by a variable-length data field. So, you must specify the maximum length of the data field.

Action: Add a length specification to the EXEC SQL VAR statement.

PCC-0115: Array required here

Cause: In an ARRAYLEN statement, you failed to specify the name of a previously declared host array. The first host variable in an ARRAYLEN statement must be an array; the second host variable, which specifies an array dimension, must be a 4-byte integer. The correct syntax follows:

```
EXEC SQL ARRAYLEN host_array (dimension) ;
```

The ARRAYLEN statement must appear in the Declare Section along with, but somewhere after, the declarations of *host_array* and *dimension.*

Action Check your spelling of both identifiers in the ARRAYLEN statement. If necessary, supply the missing host array name.

PCC-0116: This array already given in an ARRAYLEN statement

Cause: You specified the same host array in two different ARRAYLEN statements. You cannot reference a given host array in more than one ARRAYLEN statement.

Action Check your spelling of the host array names in both ARRAYLEN statements. Change one of the names so that they refer to different host arrays, or remove one of the ARRAYLEN statements.

PCC-0117: Invalid ARRAYLEN length variable type

Cause: In an ARRAYLEN statement, you failed to specify a valid array dimension. You must specify the array dimension using a previously declared 4-byte, integer host variable, not a literal or expression. For example, the following statement is illegal:

```
EXEC SQL ARRAYLEN ename_array (25) ; -- illegal dimension
```

Action Supply a valid array dimension. If necessary, declare a 4-byte, integer host variable for use in the ARRAYLEN statement.

PCC-0118: Use of host variable initialization not supported by ANSI SQL

Cause You used an ORACLE extension to the ANSI/ISO SQL standard. Specifically, you initialized a host variable in its declaration, as shown in the following Pro*C example:

```
EXEC SQL BEGIN DECLARE SECTION;
    int dept_number = 20;   -- not ANSI-compliant
    ...
EXEC SQL END DECLARE SECTION;
```

This message is only a warning issued by the FIPS Flagger when FIPS=YES.

Action: None required. But, for ANSI/ISO compliance, do not initialize host variables in their declarations.

PCC-0119: Value of const variable in INTO clause will be modified

Cause: You used a constant instead of an output host variable in the INTO clause of a SELECT statement. The SELECT INTO statement retrieves a value for each item in the select list, then assigns the values to corresponding variables in the INTO clause. Since the values of constants are fixed, you cannot use them as output host variables.

Action: Check your spelling of all identifiers in the INTO clause. If necessary, change the constant declaration to a variable declaration.

PCC-0120: File I/O error during code generation

Cause: This is an internal error message not normally issued.

Action: Call Oracle Customer Support for assistance.

PCC-0121: Arrays of VARCHAR pointers are not supported

Cause: You tried to declare an array of pointers, which is not allowed. However, pointers to scalar types are allowed. With Pro*C, declare pointers to char[n] and VRCHAR[n] variables as pointers to char or VARCHAR (with no length specification).

Action: Correct or remove the declaration.

PCC-0122: Input file name and output file name are identical

Cause: On the command line, you mistakenly specified the same path/filename for INAME and ONAME, which designate the precompiled input and output files, respectively.

Action: Change one of the path/filenames.

PCC-0123: Entire VARCHAR declaration must be on same line

Cause: In a Pro*C program, a VARCHAR declaration spans more than one line, which is not allowed.

Action: Revise the declaration so that it uses only one line.

PCC-0124: COMMON_NAME option must be before PROGRAM or subroutine beginning

Cause: In a FORTRAN program, subroutine, or function, you mistakenly specified the precompiled option COMMON_NAME after the PROGRAM, SUBROUTINE, or FUNCTION statement. If you specify COMMON_NAME inline, its EXEC ORACLE OPTION statement must precede the PROGRAM, SUBROUTINE, or FUNCTION statement.

Action: Relocate the EXEC ORACLE OPTION statement, or specify COMMON_NAME on the command line.

PCC-1000: You are not authorized to run Pro*COBOL

Cause: Your authorization or license to run the Pro*COBOL Precompiled has expired.

Action: **Call** Oracle Customer Support for assistance.

PCC-1001: Your Pro* COBOL authorization is about to expire

Cause: Your authorization or license to run the Pro*COBOL Precompiled is about to expire.

Action: Call Oracle Customer Support for assistance.

PCC-1002: Invalid character 'X" in indicator area at line N in file "X"

Cause: In a Pro*COBOL Precompiler program, only a blank, hyphen (-), asterisk (*), slash (/), or letter "D" is allowed in the indicator area, but the precompiled found another character.

Action: Remove or replace the invalid character. If you specified the FORMAT=ANSI option, check for an end-of-line in column 7.

PCC-1003: Invalid continuation at line N in file "X"

Cause: In a Pro*COBOL program, a continuation line was completely blank, except for the continuation character.

Action: Remove or replace the empty continuation line.

PCC-1004: In an EXEC statement at end-of-file

Cause: In a Pro*COBOL input file, the last EXEC statement was not terminated properly.

Action: T erminate the last EXEC statement with an END-EXEC.

PCC-1005: PROCEDURE DIVISION not found

Cause: The precompiled could not find the PROCEDURE DIVISION header in a Pro*COBOL program. Some possible causes follow:

- a keyword in the header is missing or misspelled
- there is an apostrophe in the REMARKS section (the precompiled mistook the apostrophe for the beginning of a string literal)
- there is an unterminated literal in the WORKING-STORAGE SECTION
- you specified the wrong value for the FORMAT option

Action: Make sure the PROCEDURE DIVISION header is in place and spelled correctly, that there is no apostrophe in the REMARKS section, that all literals in the WORKING-STORAGE SECTION are terminated, and that you specify the right value for the FORMAT option.

PCC-1006: EXEC statement cannot begin in Area A at line N in file "X"

Cause: In a Pro*COBOL program, EXEC statements must begin in Area B, but the precompiled found a statement beginning in Area A.

Action: Move the statement rightward so that it begins in Area B.

PCC-1002: WORKING-STORAGE SECTION not found

Cause: The precompiled could not find the WORKING-STORAGE SECTION header in a Pro*COBOL program, probably because a keyword is missing or misspelled. Or, you might have specified the wrong value for the FORMAT option.

Action: Make sure the WORKING-STORAGE SECTION header is in place and spelled correctly and that you specify the right value for the FORMAT option.

PCC-1008: Multiple element records not allowed in Declare Section

Cause: A few COBOL compilers do not allow group items to be passed as parameters in a CALL statement. (Check your COBOL compiler user's guide.) If your compiler is one of these, group items within the Declare Section can contain only one elementary item.

Action: Assign each host variable its own group item.

PCC-1009: For HOST=COB74, a SQL statement must be followed by ELSE or "."

Cause: In a Pro*COBOL program, an EXEC SQL statement is followed by another statement in the same sentence. An EXEC SQL statement must be the last statement in a COBOL-74 sentence and so must be terminated by the keyword ELSE or a period.

Action: Change the program logic, making the EXEC SQL statement the last statement in the sentence.

PCC-1010: Invalid use of NULL character in character literal

Cause: A null character (binary zero) was found in a string literal. This is not allowed by Pro*COBOL.

Action: Remove the null character from the string literal.

PCC-1100: You are not authorized to run Pro*FORTRAN

Cause: Your authorization or license to run the Pro*FORTRAN Precompiled has expired.

Action: Call Oracle Customer Support for assistance.

PCC-1101: Your Pro*FORTRAN authorization is about to expire

Cause: Your authorization or license to run the Pro*FORTRAN Precompiled is about to expire.

Action: Call Oracle Customer Support for assistance.

PCC-1102: Invalid label at line N in file "X"

Cause: The Pro*FORTRAN Precompiled found an invalid FORTRAN statement label in columns 1 through 6.

Action: Correct or remove the statement label.

PCC-1200: You are not authorized to run Pro*C

Cause: Your authorization or license to run the Pro*C Precompiled has expired.

Action: Call Oracle Customer Support for assistance.

PCC-1201: Your Pro*C authorization is about to expire

Cause: Your authorization or license to run the Pro*C Precompiled is about to expire.

Action: Call Oracle Customer Support for assistance.

PCC-1300: You are not authorized to run Pro*PL/I

Cause: Your authorization or license to run the Pro*PL/I Precompiled has expired.

Action: Call Oracle Customer Support for assistance.

PCC-1301: Your Pro*PL/I authorization is about to expire

Cause: Your authorization or license to run the Pro*PL/I Precompiled is about to expire.

Action: Call Oracle Customer Support for assistance.

PCC-1400: You are not authorized to run Pro*Pascal

Cause: Your authorization or license to run the Pro*Pascal Precompiled has expired.

Action: Call Oracle Customer Support for assistance.

PCC-1401: Your Pro*Pascal authorization is about to expire

Cause: Your authorization or license to run the Pro*Pascal Precompiled is about to expire.

Action: Call Oracle Customer Support for assistance.

ORACLE Runtime Library Error Messages

The following error messages might be issued at run time by the ORACLE runtime library. The error code prefix RTL stands for "runtime library."

RTL-2100: out of memory (i.e., could not allocate)

Cause: The ORACLE runtime library was unable to allocate enough memory to execute your program.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2101: Inconsistent cursor cache (UCE/CUC mismatch)

Cause: The precompiled generates a unit cursor entry (UCE) array. An element in this array corresponds to an entry in the cursor cache (CUC). While doing a consistency check on the cursor cache, the ORACLE runtime library found that the unit cursor entry or cursor cache entry for an EXEC SQL statement is missing.

Action: The unit cursor entry must be regenerated, so rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2102: Inconsistent cursor cache (no CUC entry for this UCE)

Cause: The precompiled generates a unit cursor entry (UCE) array. An element in this array corresponds to an entry in the cursor cache (CUC). While doing a consistency check on the cursor cache, the ORACLE runtime library was unable to find a cursor cache entry in the UCE array. This happens only if your program runs out of memory.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2103: Inconsistent cursor cache (out-of-range CUC ref)

Cause: The precompiled generates a unit cursor entry (UCE) array. An element in this array corresponds to an entry in the cursor cache (CUC). While doing a consistency check on the cursor cache, the ORACLE runtime library found that the UCE array contains an ordinal value that is either too large or less than zero. This happens only if your program runs out of memory.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2104: Inconsistent cursor cache (no CUC available)

Cause: No cursor cache is available. The ORACLE runtime library was unable to allocate enough memory to execute your program.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2105: Inconsistent cursor cache (no CUC entry in cache)

Cause: While doing a consistency check on the cursor cache, the ORACLE runtime library found that an entry in the cursor cache was missing. This happens only if your program runs out of memory.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2106: Inconsistent tumor cache (invalid ORACLE cursor number)

Cause: While carrying out a SQL operation, the ORACLE runtime library found an invalid ORACLE cursor number.

Action: This is an internal error message not normally issued. Call Oracle Customer Support for assistance.

RTL-2107: Program too old for runtime library; please re-precompile it

Cause: Your program was precompiled by an older-version ORACLE Precompiled that is not compatible with the ORACLE runtime library.

Action: Precompiled the program using a newer-version ORACLE Precompiler.

RTL-2108: Invalid descriptor passed to runtime library

Cause: While carrying out a SQL operation, the ORACLE runtime library found an invalid descriptor.

Action: This is an internal error message not normally issued. Call Oracle Customer Support for assistance.

RTL-2109: Inconsistent host cache (out-of-range SIT ref)

Cause: The ORACLE runtime library found an SIT reference that is either too large or less than zero. This usually happens when your program runs out of memory.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2110: Inconsistent host cache (invalid SQI type)

Cause: The ORACLE runtime library found a SQI type value that is either too large or less than zero. This usually happens when your program runs out of memory.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2111: Heap consistency error

Cause: After dynamically allocating or freeing memory, the runtime library found an error while doing a consistency check on the heap. This usually happens when your program runs out of memory.

Action: Allocate more memory to your user session, then rerun the program. If the error persists, call Oracle Customer Support for assistance.

RTL-2112: SELECT..INTO returns too many rows

Cause: When you tried to SELECT into a host array, the number of rows returned by the query was larger than the dimension of host array.

Action: Either specify a larger array dimension or declare a cursor for use with the FETCH statement.

RTL-2114: Invalid SQL Cursor usage: trying to close a closed cursor

Cause: You tried to CLOSE a never OPENed or already CLOSEd cursor or you misspelled the cursor name. Only a currently open cursor can be CLOSEd.

Action: Check your spelling of the cursor name and make sure the cursor is open before you try to CLOSE it.

RTL-2115: Code interpretation problem - check COMMON_NAME usage"

Cause: With Pro*FORTRAN, this error occurs if you specify the precompiled option COMMON_NAME incorrectly. With other ORACLE Recompilers, this error occurs when the precompiled cannot generate a segment of code.

Action: With Pro*FORTRAN, when using COMMON_NAME to precompiled two or more source modules, make sure to specify a different common name for each module. With other ORACLE Recompilers, if the error persists, call Oracle Customer Support for assistance.

RTL-2116: FATAL ERROR: Reentrant code generator gave invalid context

Cause: The code generator assigned an invalid value to a system-specific parameter.

Action **This is an** internal error **message** not normally issued. Call Oracle Customer Support for assistance.

RTL-2117: Invalid SQL Cursor usage: trying to open an opened cursor

Cause: When MODE={ANSI | ANSI14 | ANSI13}, you tried to OPEN an already open cursor. However, you can reOPEN an already open cursor only when MODE=ORACLE.

Action: When MODE={ANSI | ANSI14 | ANSI13}, make sure you CLOSE a cursor before trying to reOPEN it. If you want to reOPEN an already open cursor to avoid reparsing specify MODE=ORACLE.

RTL-2118: Invalid row for a WHERE CURRENT OF operation

Cause: You tried to reference a nonexistent row using the CURRENT OF clause in an UPDATE or DELETE statement. This happens when no FETCH has been executed or when FETCH returns a "no data found" error that your program fails to trap.

Action: Make sure the last cursor operation succeeded and that the current row of the cursor is valid. You can check the outcome of a cursor operation in two ways: implicit checking with the WHENEVER statement or explicit checking of SQLCODE in the SQLCA.

RTL-2122: Invalid OPEN or PREPARE for this database connection

Cause: You tried to execute an OPEN or PREPARE statement using a cursor that is currently open for another database connection and therefore cannot be used for this connection.

Action: Close the cursor to make it available for this connection, or use a different cursor for this connection.

APPENDIX

E

PERFORMANCE TUNING

This appendix shows you some simple, easy-to-apply methods for improving the performance of your applications. Using these methods, you can often reduce processing time by 25% or more.

What Causes Poor Performance?

One cause of poor performance is high ORACLE communication overhead. ORACLE must process SQL statements one at a time. Thus, each statement results in another call to ORACLE and higher overhead. In a networked environment, SQL statements must be sent over the network, adding to network traffic. Heavy network traffic can slow down your application significantly.

Another cause of poor performance is inefficient SQL statements. Because SQL is so flexible, you can get the same result with two different statements, but one statement might be less efficient. For example, the following two SELECT statements return the same rows (the name and number of every department having at least one employee):

```
EXEC SQL SELECT DNAME , DEPTNO
    FROM DEPT
    WHERE DEPTNO IN ( SELECT DEPTNO FROM EMP ) ;

EXEC SQL SELECT DNAME , DEPTNO
    FROM DEPT
    WHERE EXISTS
     ( SELECT DEPTNO FROM EMP WHERE DEPT. DEPTNO = EMP . DEPTNO ) ;
```

However, the first statement is slower because it does a time-consuming full scan of the EMP table for every department number in the DEPT table. Even if the DEPTNO column in EMP is indexed, the index is not used because the subquery lacks a WHERE clause naming DEPTNO.

A third cause of poor performance is unnecessary parsing and binding. Recall that before executing a SQL statement, ORACLE must parse and bind it. Parsing means examining the SQL statement to make sure it follows syntax rules and refers to valid database objects. Binding means associating host variables in the SQL statement with their addresses so that ORACLE can read or write their values.

Many applications manage cursors poorly. This results in unnecessary parsing and binding, which adds noticeably to processing overhead.

How Can Performance be Improved?

If you are unhappy with the performance of your precompiled programs, there are several ways you can reduce overhead.

You can greatly reduce ORACLE communication overhead, especially in networked environments, by

- using host arrays
- using embedded PL/SQL

You can reduce processing overhead-sometimes dramatically-by

- optimizing SQL statements
- using indexes
- taking advantage of row-level locking
- eliminating unnecessary parsing

The following sections look at each of these ways to cut overhead.

Using Host Arrays

Host arrays can boost performance because they let you manipulate an entire collection of data with a single SQL statement. For example, suppose you want to INSERT salaries for 300 employees into the EMP table. Without arrays your program must do 300 individual INSERTs—one for each employee. With arrays, only one INSERT is necessary. Consider the following statement:

```
EXEC SQL INSERT INTO EMP (SAL) VALUES (: salary ) ;
```

If *salary* is a simple host variable, ORACLE executes the INSERT statement once, inserting a single row into the EMP table. In that row, the SAL column has the value of *salary*. To insert 300 rows this way, you must execute the INSERT statement 300 times.

However, if *salary* is a host array of size 300, ORACLE inserts all 300 rows into the EMP table at once. In each row, the SAL column has the value of an element in the *salary* array.

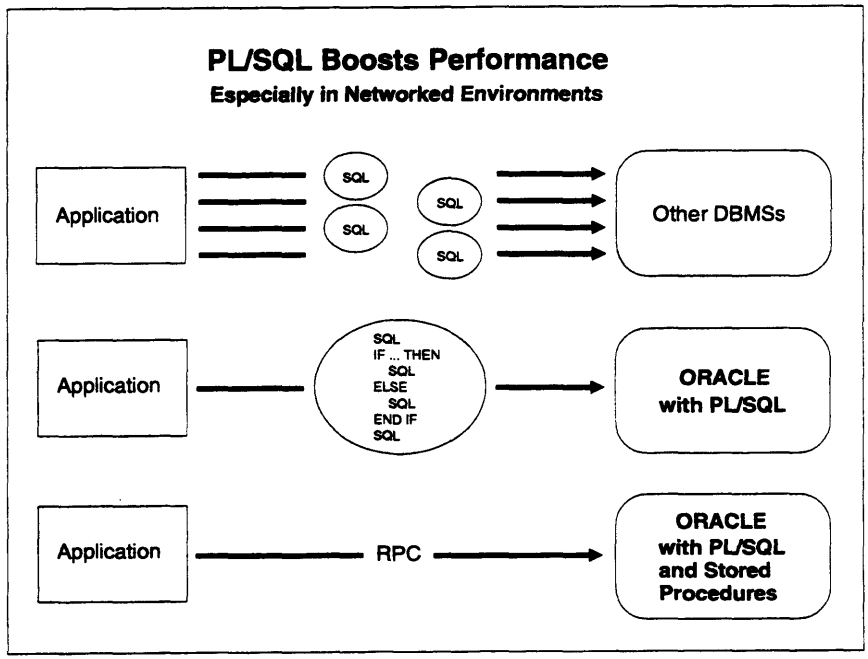
For more information, see Chapter 8, “Using Host Arrays.”

Using Embedded PL/SQL

As Figure E-1 shows, if your application is database-intensive, you can use control structures to group SQL statements in a PL/SQL block, then send the entire block to ORACLE. This can drastically reduce communication between your application and ORACLE.

Also, you can use PL/SQL subprograms to reduce calls from your application to ORACLE. For example, to execute ten individual SQL statements, ten calls are required, but to execute a subprogram containing ten SQL statements, only one call is required.

Figure E-1
PL/SQL Boosts Performance



PL/SQL can also cooperate with ORACLE application development tools such as SQL*Forms, SQL*Menu, and SQL*ReportWriter. By adding procedural processing power to these tools, PL/SQL boosts performance. Using PL/SQL, a tool can do any computation quickly and efficiently without calling on ORACLE. This saves time and reduces network traffic.

For more information, see Chapter 5 and the *PL/SQL User's Guide and Reference.*

Optimizing SQL Statements

For every SQL statement, the ORACLE optimizer generates an execution plan, which is a series of steps that ORACLE takes to execute the statement. These steps are determined by rules given in the *0RACLE7 Server Application Developer's Guide.* Following these rules will help you write optimal SQL statements.

Optimizer Hints

In some cases, you can suggest to ORACLE the way to optimize a SQL statement. These suggestions, called *hints,* let you influence decisions made by the optimizer.

Hints are not directives; they merely help the optimizer do its job. Some hints limit the scope of information used to optimize a SQL statement, while others suggest overall strategies.

You can use hints to specify the

- optimization approach for a SQL statement
- access path for each referenced table
- join order for a join
- method used to join tables

Hence, hints fall into the following four categories:

- Optimization Approach
- Access Path
- Join Order
- Join Operation

For example, the two optimization approach hints, COST and NOCOST, invoke the cost-based optimizer and the rule-based optimizer, respectively.

You give hints to the optimizer by placing them in a C-style comment immediately after the verb in a SELECT, UPDATE, or DELETE statement. For instance, the optimizer uses the cost-based approach for the following statement

```
SELECT /*+ COST */ ename, sal INTO . . .
```

For more information about optimizer hints, see the *ORACLE7 Server Application Developer's Guide.*

Trace Facility

You can use the SQL trace facility and the EXPLAIN PLAN statement to identify SQL statements that might be slowing down your application.

The SQL trace facility generates statistics for every SQL statement executed by ORACLE. From these statistics, you can determine which statements take the most time to process. Then, you can concentrate your tuning efforts on those statements.

The EXPLAIN PLAN statement shows the execution plan for each SQL statement in your application. An *execution plan* describes the database operations that ORACLE must carry out to execute a SQL statement. You can use the execution plan to identify inefficient SQL statements.

For instructions on using these tools and analyzing their output, see the *ORACLE7 Server Application Developer's Guide.*

Using Indexes

Using ROWIDS, an *index* associates each distinct value in a table column with the rows containing that value. An index is created with the CREATE INDEX statement. For details, see the *ORACLE7 Server SQL Language Reference Manual.*

You can use indexes to boost the performance of queries that return less than 15% of the rows in a table. A query that returns 15% or more of the rows in a table is executed faster by *full scan,* that is, by reading all rows sequentially.

Any query that names an indexed column in its WHERE clause can use the index. For guidelines that help you choose which columns to index, see the *ORACLE7 Server Application Developer's Guide.*

Taking Advantage of Row-level Locking

By default, ORACLE locks data at the row level rather than the table level. Row-level locking allows multiple users to access different rows in the same table concurrently. The resulting performance gain is significant.

You can specify table-level locking, but it lessens the effectiveness of the transaction processing option. For more information about table locking, see the section “Using LOCK TABLE” in Chapter 6.

Applications that do online transaction processing benefit most from row-level locking. If your application relies on table-level locking, modify it to take advantage of row-level locking. In general, avoid explicit table-level locking.

Eliminating Unnecessary Parsing

Eliminating unnecessary parsing requires correct handling of cursors and selective use of the following cursor management options:

- MAXOPENCURSORS
- HOLD_CURSOR
- RELEASE_CURSOR

These options affect implicit and explicit cursors, the cursor cache, and private SQL areas.

Handling Explicit Cursors

Recall that there are two types of cursors: implicit and explicit. ORACLE implicitly declares a cursor for all data definition and data manipulation statements. However, for queries that return more than one row, you must explicitly declare a cursor (or use host arrays). You use the DECLARE CURSOR statement to declare an explicit cursor. The way you handle the opening and closing of explicit cursors affects performance.

If you need to reevaluate the active set, simply reOPEN the cursor. OPEN will use any new host-variable values. You can save processing time if you do not CLOSE the cursor first.

Note: To make performance tuning easier, ORACLE lets you reOPEN an already open cursor. However, this is ANSI extension. So, when MODE={ANSI | ANSI14 | ANSI13}, you must CLOSE a cursor before reOPENing it.

Only CLOSE a cursor when you want to free the resources (memory and locks) acquired by OPENing the cursor. For example, your program should CLOSE all cursors before exiting.

Cursor Control

In general, there are two ways to control an explicitly declared cursor:

- use DECLARE, OPEN, and CLOSE
- use PREPARE, DECLARE, OPEN, and CLOSE

With the first way, beware of unnecessary parsing. OPEN does the parsing, but only if the parsed statement is unavailable because the cursor was CLOSEd or never OPENed. Your program should DECLARE the cursor, reOPEN it every time the value of a host variable changes, and CLOSE it only when the SQL statement is no longer needed.

With the second way (for dynamic SQL Methods 3 and 4), PREPARE does the parsing, and the parsed statement is available until a CLOSE is executed. Your program should PREPARE the SQL statement and DECLARE the cursor, reOPEN the cursor every time the value of a host variable changes, rePREPARE the SQL statement and reOPEN the cursor if the SQL statement changes, and CLOSE the cursor only when the SQL statement is no longer needed.

Placing OPEN and CLOSE statements in a loop might cause a reparse at every OPEN. In the next example, the SQL statement associated with *emp_cursor* might be reparsed for each iteration of the FOR loop:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME FROM EMP WHERE SAL > : salary;

set salary = 1000;

WHILE  salary <= 5000 DO
    EXEC SQL OPEN  emp_cursor;
    LOOP
        EXEC SQL FETCH emp_cursor INTO . . .
        ...
    ENDLOOP;
    EXEC SQL CLOSE  emp_cursor;
ENDWHILE;
```

Whenever possible, place OPEN and CLOSE statements outside loops to avoid reparsing. In the last example, the CLOSE statement should follow ENDWHILE.

Using the Cursor Management Options

A SQL statement need be parsed only once unless you change its makeup. For example, you change the makeup of a query by adding a column to its select list or WHERE clause. The HOLD_CURSOR, RELEASE_CURSOR, and MAXOPENCURSORS options give you some control over how ORACLE manages the parsing and reparsing of SQL statements. Declaring an explicit cursor gives you maximum control over parsing.

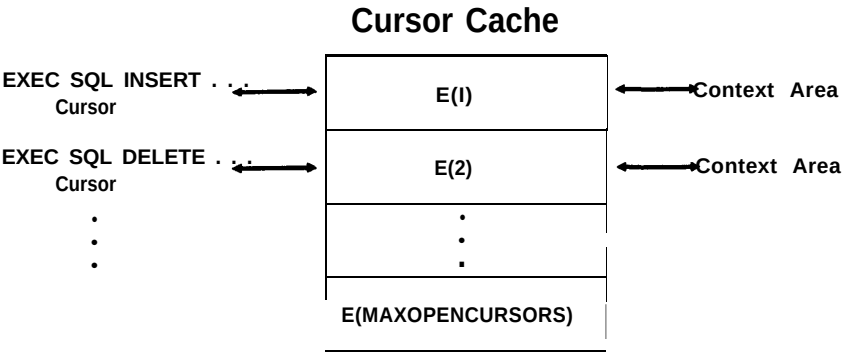
Private SQL Areas and Cursor Cache

When a data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache is a continuously updated area of memory used for cursor management. The cursor cache entry is in turn linked to a private SQL area.

The private SQL area, a work area created dynamically at run time by ORACLE, contains the parsed SQL statement, the addresses of host variables, and other information needed to process the statement. An explicit cursor lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

Figure E-2 represents the cursor cache after your program has done an INSERT and a DELETE.

Figure E-2
Cursors Linked via the Cursor Cache



Resource Use

The maximum number of open cursors peruser session is set by the ORACLE initialization parameter OPEN_CURSORS. You can override this parameter by using MAXOPENCURSORS to specify a lower (but not higher) value.

MAXOPENCURSORS specifies the *initial* size of the cursor cache. If a new cursor is needed and there are no free cache entries, ORA`CLE tries to reuse an entry. Its success depends on the values of HOLD_CURSOR and RELEASE_CURSOR and, for explicit cursors, on the status of the cursor itself.

If the value of MAXOPENCURSORS is less than the number of cache entries actually needed, ORACLE uses the first cache entry marked as reusable. For example, suppose an INSERT statement's cache entry E(1) is marked as reusable, and the number of cache entries already equals MAXOPENCURSORS. If the program executes anew statement, cache entry E(1) and its private SQL area might be reassigned to the new statement. To reexecute the INSERT statement, ORACLE would have to reparse it and reassign another cache entry.

ORACLE allocates an additional cache entry if it cannot find one to reuse. For example, if MAXOPENCURSORS=8 and all eight entries are active, a ninth is created. If necessary, ORACLE keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN_CURSORS. This dynamic allocation adds to processing overhead.

Thus, specifying a low value for MAXOPENCURSORS saves memory but causes potentially expensive dynamic allocations and deallocation of new cache entries. Specifying a high value for MAXOPENCURSORS assures speedy execution but uses more memory.

Infrequent Execution

Sometimes, the link between an *infrequently* executed SQL statement and its private SQL area should be temporary.

When HOLD_CURSOR=NO (the default), after ORACLE executes the SQL statement and the cursor is closed, the precompiled marks the link between the cursor and cursor cache as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks. However, because a PREPAREd cursor must remain active, its link is maintained even when HOLD_CURSOR=NO.

When RELEASE_CURSOR=YES, after ORACLE executes the SQL statement and the cursor is closed, the private SQL area is automatically freed and the parsed statement lost. This might be necessary if, for example, MAXOPENCURSORS is set low at your site to conserve memory.

If a data manipulation statement precedes a data definition statement and they reference the same tables, specify RELEASE_CURSOR=YES for the data manipulation statement. This avoids a conflict between the parse lock obtained by the data manipulation statement and the exclusive lock required by the data definition statement.

When RELEASE_CURSOR=YES, the link between the private SQL area and the cache entry is immediately removed and the private SQL area freed. Even if you specify HOLD_CURSOR=YES, ORACLE must still reallocate memory for a private SQL area and reparse the SQL statement before executing it because RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES.

Note: When DBMS=V6 and RELEASE_CURSOR=YES, after ORACLE executes a SQL statement and its cursor is closed, its parsed representation is lost. So, before the SQL statement can be reexecuted, it must be reparsed. However, when DBMS=V7 and RELEASE_CURSOR=YES, the reparse might require no processing because ORACLE caches the parsed representations of SQL statements and PL/SQL blocks in its *Shared SQL Cache.* Even if its cursor is closed, the parsed representation remains available until it is aged out of the cache.

Frequent Execution

The links between a frequently executed SQL statement and its private SQL area should be maintained because the private SQL area contains all the information needed to execute the statement. Maintaining access to this information makes subsequent execution of the statement much faster.

When HOLD_CURSOR=YES, the link between the cursor and cursor cache is maintained after ORACLE executes the SQL statement. Thus, the parsed statement and allocated memory remain available. This is useful for SQL statements that you want to keep active because it avoids unnecessary reparsing.

When RELEASE_CURSOR=NO (the default), the link between the cache entry and the private SQL area is maintained after ORACLE executes the SQL statement and is not reused unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because the parsed statement and allocated memory remain available.

Note: With prior versions of ORACLE, when RELEASE_CURSOR=NO and HOLD_CURSOR=YES, after ORACLE executes a SQL statement, its parsed representation remains available. But, with ORACLE7, when RELEASE_CURSOR=NO and HOLD_CURSOR=YES, the parsed representation remains available only until it is aged out of the Shared SQL Cache. Normally, this is not a problem, but you might get unexpected results if the definition of a referenced object changes before the SQL statement is reparsed.

Parameter Interactions

The following table shows how HOLD_CURSOR and RELEASE_CURSOR interact. Notice that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO and that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES.

HOLD_CURSOR	*RELEASE_CURSOR*	*Links are . . .*
NO	NO	marked as reusable
YES	NO	maintained
NO	YES	removed immediately
YES	YES	removed immediately

APPENDIX

F

SYNTACTIC AND SEMANTIC CHECKING

By checking the syntax and semantics of embedded SQL statements and PL/SQL blocks, the ORACLE Precompilers help you quickly find and fix coding mistakes. This appendix shows you how to use the SQLCHECK option to control the type and extent of checking.

You can specify the following values for SQLCHECK:

- SEMANTICS | FULL
- SYNTAX | LIMITED
- N O N E

The values SEMANTICS and FULL are equivalent, as are the values SYNTAX and LIMITED. The default value is SYNTAX.

The use of SQLCHECK does not affect the normal syntax checking done on data control, cursor control, and dynamic SQL statements.

Specifying SQLCHECK={SEMANTICS | FULL}

When SQLCHECK={SEMANTICS | FULL}, the precompiled checks the syntax and semantics of

- data manipulation statements (INSERT, UPDATE, and so on)
- PL/SQL blocks
- host variable datatypes

as well as the syntax of

- data definition statements (CREATE, ALTER, and so on)

However, only syntactic checking is done on data manipulation statements that use the AT *db_name* clause.

When SQLCHECK={SEMANTICS | FULL}, the precompiled gets information needed for a semantic check by using embedded DECLARE TABLE statements or if you specify the USERID option on the command line, by connecting to ORACLE and accessing the data dictionary. You need not connect to ORACLE if every table referenced in a data manipulation statement or PL/SQL block is defined in a DECLARE TABLE statement.

If you connect to ORACLE, but some needed information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. A DECLARE TABLE definition overrides a data dictionary definition if they conflict.

If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK={SEMANTICS | FULL}.

What Is Syntactic and Semantic Checking?

Rules of syntax specify how language elements are sequenced to form valid statements. Thus, *syntactic checking* verifies that keywords, object names, operators, delimiters, and soon are placed correctly in your SQL statement. For example, the following embedded SQL statements contain syntax errors:

```
EXEC SQL DELETE FROM EM? WHER DEPTNO = 20 ;
    -- misspelled keyword WHERE

EXEC SQL INSERT INTO EMP COMM, SAL VALUES (NULL, 1500 ) ;
    -- missing parentheses around column names COMM and SAL
```

Rules of semantics specify how valid external references are made. Thus, *semantic checking* verifies that references to database objects and host variables are valid and that host variable datatypes are correct. For example, the following embedded SQL statements contain semantic errors:

```
EXEC SQL DELETE FROM EMPP WHERE DEPTNO = 20 ;
    -- nonexistent table, EMPP

EXEC SQL SELECT * FROM EMP WHERE ENAME = : emp_name;
    -- undeclared host variable, emp_name
```

The rules of SQL syntax and semantics are defined in the *SQL Language Reference Manual.*

Controlling the Type and Extent of Checking

You control the type and extent of checking by specifying the SQLCHECK option on the command line. With SQLCHECK, the type of checking can be syntactic, semantic, or both. The *extent* of checking can include the following:

- data definition statements (such as CREATE and GRANT)
- data manipulation statements (such as SELECT and INSERT)
- PL/SQL blocks

However, SQLCHECK cannot check dynamic SQL statements because they are not fully defined until run time.

When checking data manipulation statements, the precompiled uses the ORACLE7 set of syntax rules found in the *ORACLE7 Server SQL Language Reference Manual*, but uses a stricter set of semantic rules. In particular, stricter datatype checking is done. As a result, existing applications written for earlier versions of ORACLE might not precompiled successfully when SQLCHECK={SEMANTICS | FULL}.

Specify SQLCHECK={SEMANTICS | FULL} when you precompiled new programs or want stricter datatype checking.

Enabling a Semantic Check

When SQLCHECK={SEMANTICS | FULL}, the precompiled can get information needed for a semantic check in either of the following ways:

- connect to ORACLE and access the data dictionary
- use embedded DECLARE TABLE statements

Connecting to ORACLE

To do a semantic check, the precompiled can connect to an ORACLE database that maintains definitions of tables and views referenced in your host program.

After connecting to ORACLE, the precompiled accesses the data dictionary for needed information. The *data dictionary* stores table and column names, table and column constraints, column lengths, column datatypes, and so on.

If some of the needed information cannot be found in the data dictionary (because your program refers to a table not yet created, for example), you must supply the missing information using the DECLARE TABLE statement (discussed later in this appendix).

To connect to ORACLE, specify the USERID option on the command line, using the syntax

```
USERID=usernarne/password
```

where *username* and *password* comprise a valid ORACLE userid. If you omit the password, you are prompted for it.

If, instead of a username and password, you specify

```
USERID=/
```

the precompiled attempts to automatically connect to ORACLE. The attempt succeeds only if an existing ORACLE username matches your operating system ID prefixed with "OPS$". For example, if your operating system ID is MBLAKE, an automatic connect only succeeds if OPS$MBLAKE is a valid ORACLE username.

If you omit the USERID option, the precompiled must get needed information from embedded DECLARE TABLE statements.

If you try connecting to ORACLE but cannot (because the database is unavailable, for example), an error message is issued and your program is not precompiled.

Using DECLARE TABLE

The precompiled can do a semantic check without connecting to ORACLE. To do the check, the precompiler must get information about tables and views from embedded DECLARE TABLE statements Thus, every table referenced in a data manipulation statement or PL/SQL block must be defined in a DECLARE TABLE statement.

The syntax of the DECLARE TABLE statement is

```
EXEC SQL DECLARE table_name TABLE
    ( col_name col_datatype [DEFAULT expr] [NULL |NOT NULL |, . . . ) ;
```

where *expr* is any expression that can be used as a default column value in the CREATE TABLE statement.

Though not preferred, the following syntax can be used to maintain compatibility with IBM's DB2:

```
EXEC SQL DECLARE table_name TABLE
    ( col_name col_datatype [NOT NULL [WITH DEFAULT] ] , . . . ) ;
```

Whichever syntax you use, the NOT NULL and DEFAULT information is currently ignored. It serves only as documentation.

If you use DECLARE TABLE to define a database table that already exists, the precompiled uses your definition, ignoring the one in the data dictionary.

Specifying SQLCHECK={SYNTAX | LIMITED}

When SQLCHECK={SYNTAX | LIMITED}, the precompiled checks the syntax of

- data manipulation statements
- data definition statements
- host variable datatypes

No semantic check is done, and the following restrictions apply

- No connection to ORACLE is attempted and USERID becomes an invalid option. If you specify USERID, a warning message is issued.
- DECLARE TABLE statements are ignored; they serve only as documentation.
- PL/SQL blocks are not allowed. If the precompiled finds a PL/SQL block, an error message is issued.

Note: The values LIMITED and SYNTAX are equivalent; that is, they specify the same action. However, under Version 1.3 of the ORACLE Precompilers, the value LIMITED specifies a different action. With Version 1.3, when SQLCHECK=LIMITED, the precompiled checks the syntax and semantics of data manipulation statements and PL/SQL blocks. Also, to do the semantic check, the precompiled connects to ORACLE, so you must specify the USERID option.

When checking data manipulation statements, the precompiled uses ORACLE7 syntax rules. These rules are downwardly compatible, so specify SQLCHECK={SYNTAX | LIMITED} when migrating your precompiled programs.

Specifying SQLCHECK=NONE

When SQLCHECK=NONE (the default), no semantic check is done, a minimal syntactic check is done, and the following restrictions apply

- No connection to ORACLE is attempted and USERID becomes an invalid option. If you specify USERID, a warning message is issued.
- DECLARE TABLE statements are ignored; they serve only as documentation.
- PL/SQL blocks are not allowed. If the precompiled finds a PL/SQL block, an error message is issued.

Specify SQLCHECK=NONE if your program references tables not yet created and is missing DECLARE TABLE statements for them or if stricter datatype checking is undesirable.

Entering the SQLCHECK Option

You can enter the SQLCHECK option inline or on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line. For example, if you specify SQLCHECK={SYNTAX | LIMITED} on the command line, you cannot specify SQLCHECK={SEMANTICS | FULL} inline.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X