Pro*COBOL® Precompiler

Programmer's Guide

Release 8.0

December, 1997

Part No. A58232-01

ORACLE®

Enabling the Information Age™

Pro*COBOL® Precompiler Programmer's Guide

Part No. A58232-01

Release 8.0

© Copyright 1997, Oracle Corporation. All rights reserved.

Primary Author: Jack Melnick

Contributors: Michael Chiocca, Maura Joglekar, Thomas Kurian, Shiao-yen Lin, Diana Lorentz, Lee Osborne, Jacqui Pons, Ajay Popat, Pamela Rothman, Gael Turk

Graphic Designer: Valarie Moore

Contents

2 Learning the Basics

3 Writing a Pro*COBOL Program

4   Advanced Pro*COBOL Programs

5   Using Embedded SQL

6   Using Embedded PL/SQL

7   Running the Pro*COBOL Precompiler

8   Defining and Controlling Transactions

9   Error Handling and Diagnostics

11   Using Dynamic SQL

13   Writing User Exits

B Operating System Dependencies

C Oracle8 Reserved Words, Keywords, and Namespaces

D Performance Tuning

E Syntactic and Semantic Checking

F Embedded SQL Commands and Precompiler Directives

Send Us Your Comments

Pro*COBOL® Precompiler Programmer's Guide, Release 8.0

Part No. A58232-01

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

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- electronic mail - infodev@us.oracle.com
- FAX - (650) 506-7228 Attn: Information Development
- postal service:

  Oracle Corporation
  Server Technologies Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

Preface

This manual is a comprehensive user's guide and reference to the Oracle Pro*COBOL Precompiler. It shows you how to develop COBOL programs that use the database languages SQL and PL/SQL to access and manipulate Oracle data. See the *Oracle8 SQL Reference* and *PL/SQL User's Guide and Reference* for more information on SQL and PL/SQL.

This preface covers these topics:

- What This Manual Has to Offer

- Who Should Read This Manual?

- How This Manual Is Organized

- Conventions Used in This Manual

- Sample Programs

- Does the Pro*COBOL Precompiler Meet Industry Standards?

- Your Comments Are Welcome

What This Manual Has to Offer

This manual shows you how the Oracle Pro*COBOL Precompiler and embedded SQL can benefit your entire applications development process. It gives you lessons in 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 manual is its emphasis on getting the most out of Pro*COBOL and embedded SQL. To help you master these tools, this manual 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 manual. For that kind of information, refer to your system-specific Oracle documentation.

> For information about migrating your applications from Oracle7 to Oracle8, see *Oracle8 Migration*.

Who Should Read This Manual?

Anyone developing new COBOL applications or converting existing applications to run in the Oracle8 environment will benefit from reading this manual. Written especially for programmers, this comprehensive treatment of Pro*COBOL will also be of value to systems analysts, project managers, and others interested in embedded SQL applications.

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

- applications programming in COBOL
- the SQL database language
- Oracle8 concepts and terminology

How This Manual Is Organized

A brief summary of what you will find in each chapter and appendix follows.

Chapter 1: Introduction

This chapter introduces you to Pro*COBOL. You look at its 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: Writing a Pro*COBOL Program

This chapter provides the basic information you need to write a Pro*COBOL program. You learn programming guidelines, coding conventions, language-specific features and restrictions.

Chapter 4: Advanced Pro*COBOL Programs

This chapter describes National Language Support (NLS), discusses datatype equivalencing, shows you how to declare SQL communications areas, and how to connect to an Oracle database. In addition, this chapter shows you how to embed Oracle Call Interface (OCI) calls in your program and how to develop X/Open applications.

Chapter 5: Using Embedded SQL

This chapter teaches you the essentials of embedded SQL programming. You learn how to use host variables, indicator variables, cursors, cursor variables, and the fundamental SQL commands that insert, update, select, and delete Oracle data.

Chapter 6: 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 subprograms, host arrays, and dynamic PL/SQL.

Chapter 7: Running the Pro*COBOL Precompiler

This chapter details the requirements for running the Pro*COBOL Precompiler. You learn what happens during precompilation, how to issue the Pro*COBOL com-

mand, how to specify the many useful precompiler options, and how to do conditional and separate precompilations.

Chapter 8: Defining and Controlling Transactions

This chapter describes transaction processing. You learn the basic techniques that safeguard the consistency of your database.

Chapter 9: Error Handling and Diagnostics

This chapter provides an in-depth discussion of error reporting and recovery. You learn how to detect and handle errors using the status variable SQLSTATE, the SQLCA structure, and the WHENEVER statement. You also learn how to diagnose problems using the ORACA.

Chapter 10: Using Host Tables

This chapter looks at using tables to improve program performance. You learn how to manipulate Oracle data using tables, how to operate on all the elements of a table with a single SQL statement, and how to limit the number of table elements processed.

Chapter 11: 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 let users build SQL statements interactively at run time.

Chapter 12: Using Dynamic SQL: Advanced Concepts

This chapter shows you how to implement dynamic SQL Method 4, an advanced programming technique that lets you write highly flexible applications. Numerous examples are used to illustrate the method.

Chapter 13: Writing User Exits

This chapter focuses on writing user exits for your SQL*Forms or Oracle Forms applications. First, you learn the commands that allow a Forms application to interface with user exits. Then, you learn how to write and link a Forms user exit.

Appendix A: New Features

This appendix highlights the improvements and new features introduced with Release 8.0 of the Pro*COBOL Precompiler.

Appendix B: Operating System Dependencies

Some details of Pro*COBOL programming vary from one system to another. So, you are occasionally referred to other manuals for system-specific information. For convenience, this appendix collects all such external references.

Appendix C: Oracle8 Reserved Words, Keywords, and Namespaces

This appendix lists words that have a special meaning to Oracle and namespaces that are reserved for Oracle libraries.

Appendix D: Performance Tuning

This appendix gives you some simple methods for improving the performance of your applications.

Appendix E: 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.

Appendix F: Embedded SQL Commands and Precompiler Directives

This appendix contains descriptions of precompiler directives, embedded SQL commands, and Oracle embedded SQL extensions. These commands are prefaced in your source code with the keywords, EXEC SQL.

Conventions Used in This Manual

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.

Notation

The following notation is used in this manual:

< >	Angle brackets enclose the name of a syntactic element.
.	A dot separates an object name from a component name and so qualifies a reference.
..	Two dots separate the lowest and highest values in a range.
...	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.

#	This character is used in text to represent blank spaces when referring to the content of a database column.

Syntax Description

Embedded SQL syntax is described using a variant of Backus-Naur Form (BNF), which includes the following symbols:

[ ]	Brackets enclose optional items.
{ }	Braces enclose items only one of which is required.
|	A vertical bar separates alternatives within brackets or braces.
...	An ellipsis shows that the preceding parameter can be repeated.

Sample Programs

This manual provides several Pro*COBOL programs to help you in writing your own. These programs illustrate the key concepts and features of Pro*COBOL programming and demonstrate techniques that let you take full advantage of SQL's power and flexibility.

Each sample program in this manual is available on-line. The following table shows the usual filenames of the sample programs. However, the exact filenames are system-dependent. For exact filenames, see your Oracle system-specific documentation.

Filename	Demonstrates...
SAMPLE1.PCO	Simple Query
SAMPLE2.PCO	Cursor Operations
SAMPLE3.PCO	Host Tables
SAMPLE4.PCO	Datatype Equivalencing
SAMPLE5.PCO	Oracle Forms User Exit
SAMPLE6.PCO	Dynamic Sql Method 1
SAMPLE7.PCO	Dynamic Sql Method 2
SAMPLE8.PCO	Dynamic Sql Method 3
SAMPLE9.PCO	Calling A Stored Procedure
SAMPLE10.PCO	Dynamic SQL Method 4
SAMPLE11.PCO	Cursor Variable Operations

Does the Pro*COBOL Precompiler Meet Industry Standards?

SQL has become the standard language for relational database management systems. This section describes how the Pro*COBOL Precompiler conforms to the latest 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)

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

- ANSI Document ANSI X3.135-1992, *Database Language SQL*

- ANSI Document ANSI X3.168-1992, *Database Language Embedded SQL*

- International Standard ISO/IEC 9075:1992, *Database Language SQL*

- NIST Federal Information Processing Standard FIPS PUB 127-2, *Database Language SQL*

Requirements

ANSI X3.135-1992 (known informally as SQL92) specifies a "conforming SQL language" and, to allow implementation in stages, defines three language levels:

- Full SQL

- Intermediate SQL (a subset of Full SQL)

- Entry SQL (a subset of Intermediate SQL)

A conforming SQL implementation must support at least Entry SQL.

ANSI X3.168-1992 specifies the syntax and semantics for embedding SQL statements in application programs written in a standard programming language such as COBOL-74 and COBOL-85.

ISO/IEC 9075-1992 fully adopts the ANSI standards.

FIPS PUB 127-2, which applies to RDBMS software acquired for federal use, also adopts the ANSI/ISO 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 Oracle8, the Pro*COBOL Precompiler complies 100% with the ANSI, ISO, and NIST standards. As required, they support Entry SQL and provide a FIPS Flagger.

FIPS Flagger

According to FIPS 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 Pro*COBOL Precompiler provides the *FIPS Flagger*, which flags ANSI extensions. An *extension* is any SQL ele-

ment that violates ANSI format or syntax rules, except privilege enforcement rules. For a list of Oracle extensions to standard SQL, see the *Oracle8 SQL Reference*.

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

An 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 command-line option FIPS, see "FIPS" on page 7-20.

Certification

NIST tested the Pro*COBOL Precompiler for ANSI Entry SQL compliance using the *SQL Test Suite*, which consists of nearly 300 test programs. Specifically, the programs tested for conformance to the COBOL embedded SQL standards. As a result, the Pro*COBOL Precompiler was 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

MIA/SPIRIT

The Pro*COBOL Precompiler provides National Language Support (NLS) of multi-byte character data by complying with the Multivendor Integration Architecture (MIA) specification, Version 1.3, and the Service Providers Integrated Requirements for Information Technology (SPIRIT) specification, Issue 2.

Your Comments Are Welcome

The Oracle Corporation technical staff values your comments. As we write and revise, your opinions are the most important feedback we receive. Please use the

Reader's Comment Form to tell us what you like and dislike about this Oracle publication.

1

Introduction

This chapter introduces you to the Pro*COBOL Precompiler. You look at its role in developing application programs that manipulate Oracle data and find out what it allows your applications to do. The following questions are answered:

- What Is Pro*COBOL?
- Why Use the Pro*COBOL Precompiler?
- Why Use SQL?
- Why Use PL/SQL?
- What Does Pro*COBOL Offer?

What Is Pro*COBOL?

The Pro*COBOL Precompiler is a programming tool that allows you to embed SQL statements in a high-level host program. As Figure 1–1 shows, the precompiler accepts the host program as input, translates the embedded SQL statements into standard Oracle run-time library calls, and generates a source program that you can compile, link, and execute in the usual way.

Figure 1–1   Embedded SQL Program Development

Editor

Host Program   With embedded SQL statements

Oracle Precompiler

Source Program   With all SQL statements replaced by library calls

Compiler

Object Program

Linker   Oracle Runtime Library (SQLLIB)

To resolve calls

Executable Program
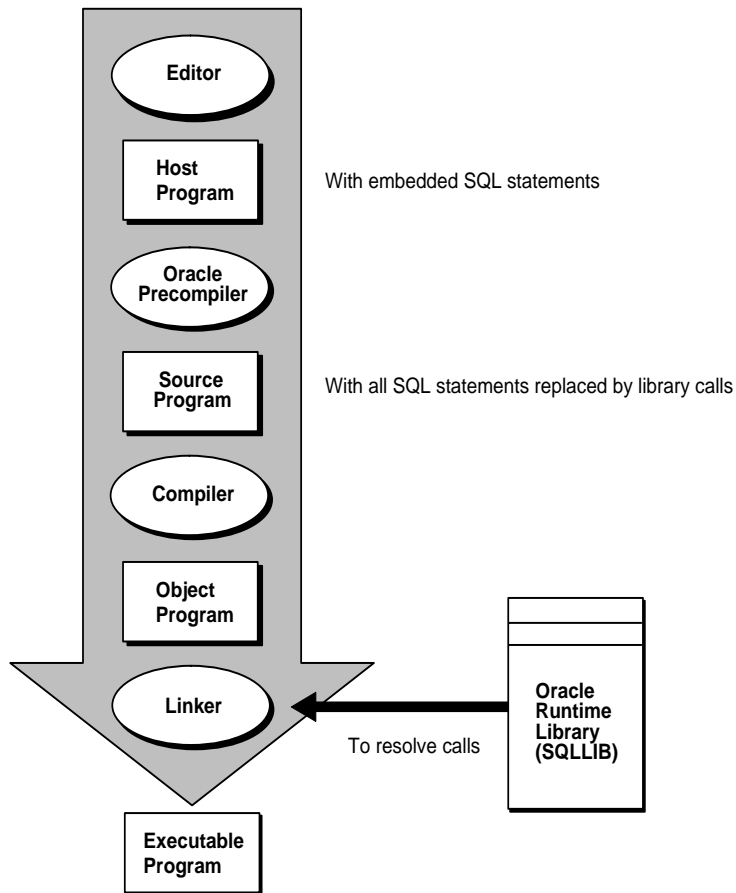
Language Alternatives

Oracle Precompilers are available (but not on all systems) for the following high-level languages:

- C/C++

- COBOL

- FORTRAN

Pro*Pascal, Pro*ADA and Pro*PL/I will not be released with Oracle8. However, Oracle will continue to issue patch releases for Pro*FORTRAN as bugs are reported and corrected.

Why Use the Pro*COBOL Precompiler?

The Pro*COBOL Precompiler lets you pack the power and flexibility of SQL into your application programs. You can use SQL in popular high-level languages such as COBOL. A convenient, easy to use interface lets your application access Oracle directly.

Unlike many application development tools, Pro*COBOL lets 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, with Pro*COBOL you can fine-tune your applications. They allow close monitoring of resource usage, SQL statement execution, and various run-time indicators. With this information, you can adjust program parameters for maximum performance.

Why Use SQL?

If you want to access and manipulate Oracle data, you need SQL. Whether you use SQL interactively or embedded in an application program depends on the job at hand. If the job requires the procedural processing power of COBOL, 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 or Server Manager. 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 SQL".

What Does Pro*COBOL Offer?

As Figure 1–2 shows, Pro*COBOL offers many features and benefits that help you to develop effective, reliable applications.

Figure 1–2   Features and Benefits
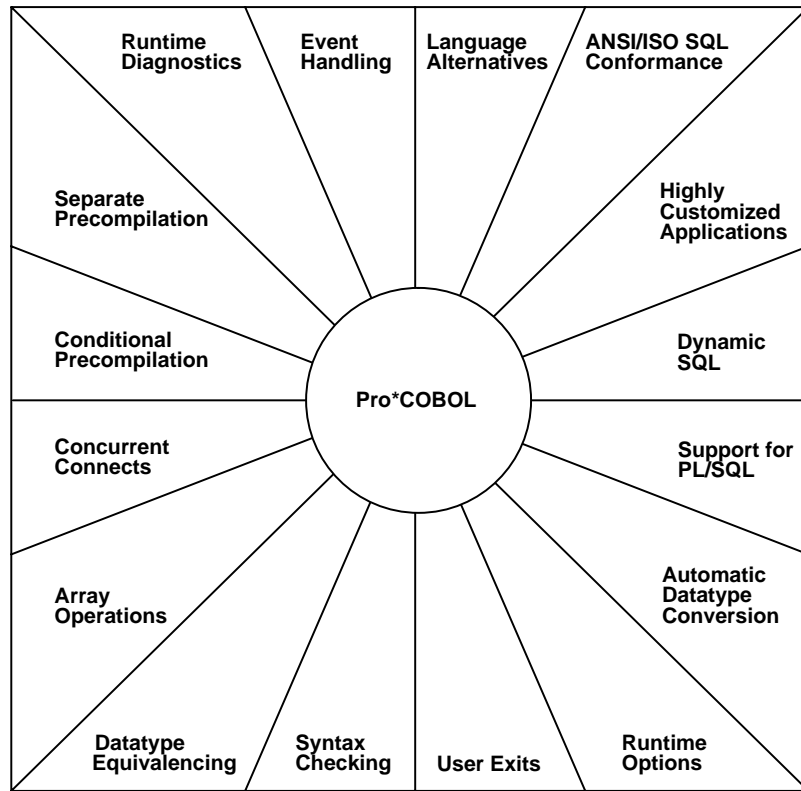
For example, the Pro*COBOL Precompiler allows you to:

- write your application in COBOL

- conform to the ANSI/ISO embedded SQL standard

- take advantage of dynamic SQL, an advanced programming technique that lets your program accept or build any valid SQL statement at run-time in a COBOL program

- design and develop highly customized applications

- convert automatically between Oracle8 internal datatypes and COBOL datatypes
- improve performance by embedding PL/SQL transaction processing blocks in your COBOL application program
- specify useful precompiler options and change their values during precompilation
- use datatype equivalencing to control the way Oracle8 interprets input data and formats output data
- precompile several program modules separately, then link them into one executable program
- check the syntax and semantics of embedded SQL data manipulation statements and PL/SQL blocks
- access Oracle8 databases on multiple nodes concurrently using Net8
- use arrays as input and output program variables
- precompile sections of code conditionally so that your host program can run in different environments
- interface with tools such as Oracle Forms and Oracle Reports via user exits written in a high-level language
- handle errors and warnings with the ANSI-approved status variables SQLSTATE and SQLCODE, and/or the SQL Communications Area (SQLCA) and WHENEVER statement
- use an enhanced set of diagnostics provided by the Oracle Communications Area (ORACA)

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.

Topics covered are:

- Key Concepts of Embedded SQL Programming
- Steps in Developing an Embedded SQL Application
- The Format of SQL Statements
- INCLUDE Statements
- The SQLCA
- Oracle8 Datatypes
- Declaring and Referencing Host Variables
- Declaring and Referencing Indicator Variables
- Sample Tables
- A Program Example 1: Simple Query

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

- tables

- 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 the application program houses the SQL statements, it is called a *host program*, and the language in which it is written is called the *host language.* For example, with Pro*COBOL you can embed SQL statements in a COBOL host program.

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 criteria.

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 SQL statements generate calls to the database. They include almost all queries, DML (Data Manipulation Language), DDL (Data Definition Language), and DCL (Data Control Language) statements.

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.

Table 2–1 groups the various embedded SQL statements:

Table 2–1   Embedded SQL Statements

Declarative SQL	
STATEMENT	PURPOSE
ARRAYLEN*	To use host tables with PL/SQL
BEGIN DECLARE SECTION* END DECLARE SECTION*	To declare host variables
DECLARE*	To name Oracle objects
INCLUDE*	To copy in files
VAR*	To equivalence variables
WHENEVER*	To handle runtime errors
Executable SQL	
STATEMENT	PURPOSE
ALLOCATE* ALTER ANALYZE AUDIT COMMENT CONNECT* CREATE DROP GRANT NOAUDIT RENAME REVOKE TRUNCATE	To define and control Oracle data

Table 2–1   Embedded SQL Statements

CLOSE* DELETE EXPLAIN PLAN FETCH* INSERT LOCK TABLE OPEN* SELECT UPDATE	To query 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 words EXEC SQL and end them with the token END-EXEC. Pro*COBOL translates all executable 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. Compare the following interactive and embedded ROLLBACK statements:

```
ROLLBACK WORK;          -- interactive
```

```
*  embedded
      EXEC SQL
          ROLLBACK WORK
      END-EXEC.
```

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. See Chapter 5, "Using Embedded SQL".

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

Dynamic SQL is an advanced programming technique that lets your program accept or build SQL statements at run time and take explicit control over datatype conversion. See Chapter 11, "Using Dynamic SQL" and Chapter 12, "Using Dynamic SQL: Advanced Concepts".

Embedded PL/SQL Blocks

Pro*COBOL treats 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 6, "Using Embedded PL/SQL".

Host and Indicator Variables

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

You use *input* host variables to pass data to the database. You use *output* host variables to pass data and status information from the database to your program.

Host variables can be used anywhere an expression can be used. But, in SQL statements, host variables must be prefixed with a colon, ':', to set them apart from database schema names.

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 (unless, to improve readability, you precede the indicator variable with the optional keyword INDICATOR).

Oracle Datatypes

Typically, a host program inputs data to the database, and the database outputs data to the program. Oracle inserts input data into database tables and selects output data into 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 pseudo-columns, 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 database column. When Oracle outputs data to your host program, if necessary, Oracle converts between the internal datatype of the database column and the external datatype of the output host variable.

Tables

Pro*COBOL lets you define table host variables (called *host tables*) and operate on them with a single SQL statement. Using the table SELECT, FETCH, DELETE, INSERT, and UPDATE statements, you can query and manipulate large volumes of data with ease.

Datatype Equivalencing

Pro*COBOL adds 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 Oracle external datatypes. For more information, see "Datatype Equivalencing" on page 4-20

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 tables).

The set of rows retrieved 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, which is 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, allowing 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. The current transaction consists of all data manipulation statements executed since the last data definition, COMMIT, or ROLLBACK statement was executed.

To help ensure the consistency of your database, Pro*COBOL lets 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 a transaction; used with ROLL-BACK, 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. Pro*COBOL provides these error handling mechanisms:

- SQLCODE status variable

- SQLSTATE status variable

- SQL Communications Area (SQLCA)

- WHENEVER statement

- Oracle Communications Area (ORACA)

SQLCODE/SQLSTATE Status Variables

After executing a SQL statement, the Oracle Server returns a status code to a variable named SQLCODE or SQLSTATE. The status code indicates whether the SQL statement executed successfully or caused an error or warning condition.

SQLCA Status Variable

The SQLCA is a data structure that 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.

WHENEVER Statement

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.

ORACA

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA. The ORACA is a data structure that handles Oracle communication. It contains cursor statistics, information about the current SQL statement, option settings, and system statistics.

Steps in Developing an Embedded SQL Application

Figure 2–1 walks you through the embedded SQL application development process.

Figure 2–1   Application Development Process
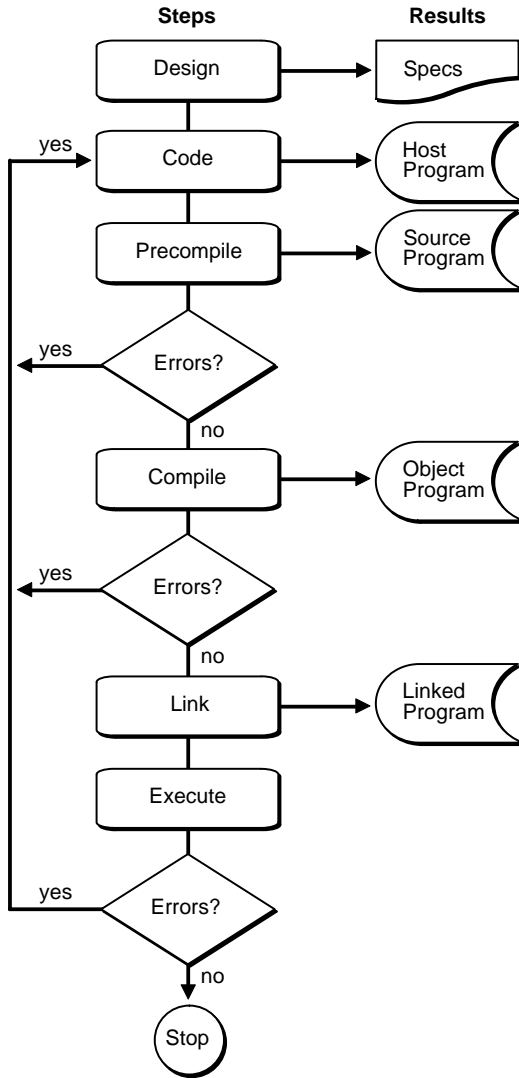
As you can see, precompiling results in a source file that can be compiled normally. Although precompiling adds a step to the traditional development process, that step is well worth taking because it lets you write very flexible applications.

The Format of SQL Statements

SQL statements begin with EXEC SQL and end with END-EXEC. A period or any other terminator can follows a SQL statement. Either of the following is allowed:

```
EXEC SQL ... END-EXEC,
EXEC SQL ... END-EXEC.
```

INCLUDE Statements

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

```
*    copy in the SQLCA file
     EXEC SQL INCLUDE SQLCA END-EXEC.
```

When you precompile 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, Pro*COBOL assumes the default extension, *.cob.*

You can set a directory path for INCLUDEd files by specifying the precompiler option

```
INCLUDE=<path>
```

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

Pro*COBOL 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 (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 your system-specific Oracle8 manuals.

The 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 Oracle8. After executing a SQL statement, Oracle8 sets SQLCA variables to indicate the outcome, as illustrated in Figure 2–2.

Figure 2–2   Updating the SQLCA
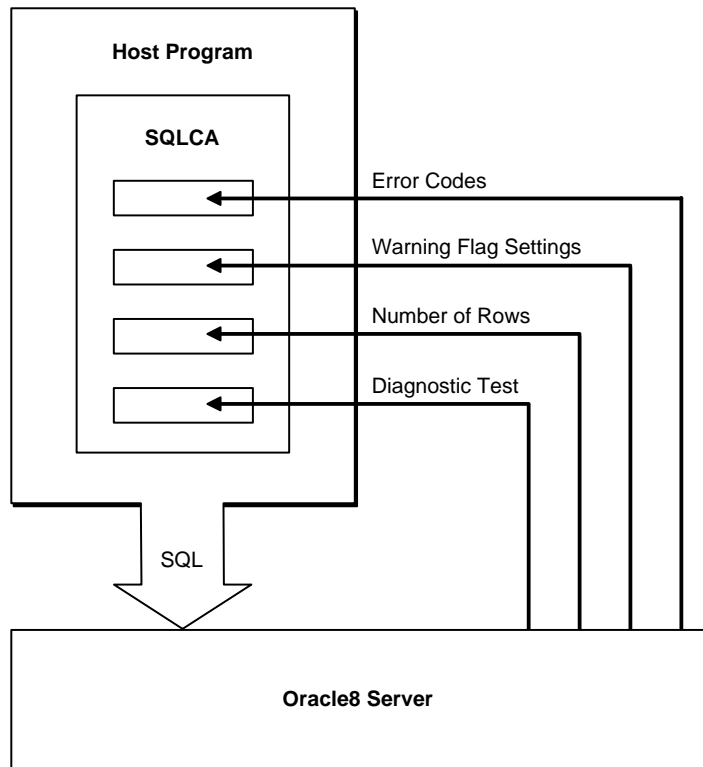
Thus, you can check to see if an INSERT, UPDATE, or DELETE statement succeeded and if so, how many rows were affected. Or, if the statement failed, you can get more information about what happened.

When MODE={ANSI13 | ORACLE}, you must declare the SQLCA by hard-coding it or by copying it into your program with the INCLUDE statement. The section "Using the SQL Communications Area" on page 9-19 shows you how to declare and use the SQLCA.

Oracle8 Datatypes

Oracle8 recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle8 stores data in database columns. Oracle8 also uses internal datatypes to represent database pseudo-columns. An external datatype specifies how data is stored in a host variable.

At precompile time, each host variable 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 Oracle8. Oracle8 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 datatype equivalencing, see "Datatype Equivalencing" on page 4-20.

Declaring and Referencing Host Variables

Every program variable used in a SQL statement must be declared according to the rules of the COBOL language. Normal rules of scope apply. COBOL variable names can be any length, but only the first 30 characters are significant for Pro*COBOL.. Any valid COBOL identifier can be used as host variables, including those beginning with digits.

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 4–6, "Conversions Between Internal and External Datatypes" shows the compatible datatypes between which Oracle8 converts automatically when necessary.

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, Pro*COBOL expands the VARCHAR declaration

```
01  ENAME  PIC X(20) VARYING.
```

into the following COBOL group item with array and length members:

```
01  ENAME.
    05  ENAME-LEN  PIC S9(4) COMP.
    05  ENAME-ARR  PIC X(20).
```

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 "VARCHAR Variables" on page 3-36.

Host Variable Guidelines

The following guidelines apply to declaring and referencing host variables. A host variable must be

- prefixed with a colon (:) in SQL statements and PL/SQL blocks
- of a datatype supported by COBOL
- of a datatype compatible with that of its source or target database column

A host variable must *not* be

- prefixed with a colon in COBOL statements
- used in data definition statements such as ALTER and CREATE

A host variable can be

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

Declaring and Referencing Indicator Variables

You can associate every host variable with an optional indicator variable. An indicator variable must be defined as a signed 4-digit computational number and, in SQL statements, must be prefixed with a colon and must directly follows its host variable unless you use the keyword INDICATOR. For a detailed discussion, see "Indicator Variables" on page 3-30.

Sample Tables

Most of the complete program examples in this guide use two sample database tables: DEPT and EMP. Their definitions follow:

```
CREATE TABLE DEPT
    (DEPTNO    NUMBER(2),
     DNAME     VARCHAR2(14),
     LOC       VARCHAR2(13))

CREATE TABLE EMP
    (EMPNO     NUMBER(4) primary key,
     ENAME     VARCHAR2(10),
     JOB       VARCHAR2(9),
     MGR       NUMBER(4),
     HIREDATE  DATE,
     SAL       NUMBER(7,2),
     COMM      NUMBER(7,2),
     DEPTNO    NUMBER(2))
```

Sample Data

Respectively, the DEPT and EMP tables contain the following rows of data:

```
DEPTNO  DNAME       LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500		30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

A Program Example 1: Simple Query

A good way to get acquainted with embedded SQL is to look at a program example.

This program logs on to the database, prompts the user for an employee number, queries the database table EMP for the employee's name, salary, and commission. The selected results are stored in host variables EMP-NAME, SALARY, and COM-MISSION. The program uses the host indicator variable, COMM-IND to detect null values in column COMMISSION. See "Indicator Variables" on page 3-30.

The paragraph DISPLAY-INFO then displays the result.

The COBOL variables USERNAME, PASSWD, and EMP-NUMBER are declared using the VARYING clause, which allows you to use a variable-length string external Oracle datatype called VARCHAR. This datatype is explained in "VARCHAR Variables" on page 3-36.

The SQLCA Communications Area is included to handle errors. If an error occurs, paragraph SQL-ERROR is performed. See "Using the SQL Communications Area" on page 9-19.

The BEGIN DECLARE SECTION and END DECLARE SECTION statements used are optional, unless you set the Precompiler option DECLARE_SECTION to YES.

The program ends when the user enters a zero employee number.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. QUERY.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME          PIC X(10) VARYING.
        01  PASSWD            PIC X(10) VARYING.
        01  EMP-REC-VARS.
            05  EMP-NAME       PIC X(10) VARYING.
            05  EMP-NUMBER     PIC S9(4) COMP VALUE ZERO.
            05  SALARY         PIC S9(5)V99 COMP-3 VALUE ZERO.
            05  COMMISSION     PIC S9(5)V99 COMP-3 VALUE ZERO.
            05  COMM-IND       PIC S9(4) COMP VALUE ZERO.
    EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL INCLUDE SQLCA END-EXEC.

        01  DISPLAY-VARIABLES.
            05  D-EMP-NAME     PIC X(10).
```

```
                       05  D-SALARY      PIC Z(4)9.99.
                       05  D-COMMISSION  PIC Z(4)9.99.

                 01  D-TOTAL-QUERIED   PIC 9(4) VALUE ZERO.

       PROCEDURE DIVISION.
       BEGIN-PGM.
           EXEC SQL
               WHENEVER SQLERROR DO PERFORM SQL-ERROR
           END-EXEC.
           PERFORM LOGON.

       QUERY-LOOP.
           DISPLAY " ".
           DISPLAY "ENTER EMP NUMBER (0 TO QUIT): " WITH NO ADVANCING.
           ACCEPT EMP-NUMBER.
           IF (EMP-NUMBER = 0) PERFORM SIGN-OFF.
           MOVE SPACES TO EMP-NAME-ARR.
           EXEC SQL
               WHENEVER NOT FOUND GOTO NO-EMP
           END-EXEC.
           EXEC SQL
               SELECT ENAME, SAL, COMM
               INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
               FROM EMP
               WHERE EMPNO = :EMP-NUMBER
           END-EXEC.
           PERFORM DISPLAY-INFO.
           ADD 1 TO D-TOTAL-QUERIED.
           GO TO QUERY-LOOP.

       NO-EMP.
           DISPLAY "NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.".
           GO TO QUERY-LOOP.

       LOGON.
           MOVE "SCOTT" TO USERNAME-ARR.
           MOVE 5 TO USERNAME-LEN.
           MOVE "TIGER" TO PASSWD-ARR.
           MOVE 5 TO PASSWD-LEN.
           EXEC SQL
               CONNECT :USERNAME IDENTIFIED BY :PASSWD
           END-EXEC.
           DISPLAY " ".
           DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
```

```
DISPLAY-INFO.
    DISPLAY " ".
    DISPLAY "EMPLOYEE    SALARY    COMMISSION".
    DISPLAY "--------    ------    ----------".
    MOVE EMP-NAME-ARR TO D-EMP-NAME.
    MOVE SALARY TO D-SALARY.
    IF COMM-IND = -1
        DISPLAY D-EMP-NAME, D-SALARY, "          NULL"
    ELSE
        MOVE COMMISSION TO D-COMMISSION
        DISPLAY D-EMP-NAME, D-SALARY, "      ", D-COMMISSION
    END-IF.

SIGN-OFF.
    DISPLAY " ".
    DISPLAY "TOTAL NUMBER QUERIED WAS ", D-TOTAL-QUERIED, ".".
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL
        COMMIT WORK RELEASE
    END-EXEC.
    STOP RUN.

SQL-ERROR.
    EXEC SQL
        WHENEVER SQLERROR CONTINUE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL
        ROLLBACK WORK RELEASE
    END-EXEC.
    STOP RUN.
```

3

Writing a Pro*COBOL Program

This chapter provides the basic information you need to write a Pro*COBOL program, including:

- Programming Guidelines
- Required Declarations and SQL Statements
- Host Variables
- Nested Programs
- Indicator Variables
- Host Tables
- VARCHAR Variables
- Connecting to Oracle
- Handling Character Data
- Concurrent Logons
- Changing Passwords at Runtime

Programming Guidelines

This section deals with embedded SQL syntax, coding conventions, and Pro*COBOL-specific features and restrictions. Topics are arranged alphabetically for quick reference.

Abbreviations

You can use the standard COBOL abbreviations, such as PIC for PICTURE IS and COMP for USAGE IS COMPUTATIONAL.

Case-insensitivity

Pro*COBOL precompiler options and values as well as all EXEC SQL statements, inline commands, and COBOL statements are case-insensitive. The precompiler accepts both upper- and lower-case tokens.

COBOL Versions

Pro*COBOL supports the standard implementation of COBOL for your operating system (usually COBOL-85 or COBOL-74). Some platforms may support both COBOL implementations. For more information, see your Oracle8 system-specific documentation.

Coding Area

You must code EXEC SQL and EXEC ORACLE statements in columns 12 through 72 (columns 73 through 80 are ignored).

Note: The precompiler option FORMAT, specifies the format of COBOL input lines. If you specify FORMAT=ANSI (default), columns 1 through 6 can contain an optional sequence number, and column 7 indicates comments or continuation lines. Division headers, section headers, paragraph names, FD and 01 statements begin in columns 8 through 11 (area A). Other statements begin in columns 12 through 72 (area B).

If you specify FORMAT=TERMINAL, columns 1 through 6 are omitted, making column 7 the left-most column.

Note: In this manual, program examples use the FORMAT=TERMINAL setting. The online sample programs use FORMAT=ANSI.

Commas

In SQL, you must use commas to separate list items, as the following example shows:

```
EXEC SQL SELECT ENAME, JOB, SAL
    INTO :EMP-NAME, :JOB-TITLE, :SALARY
    FROM EMP
    WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

In COBOL, you can use commas or blanks to separate list items. For example, the following two statements are equivalent:

```
ADD AMT1, AMT2, AMT3 TO TOTAL-AMT.
ADD AMT1 AMT2 AMT3 TO TOTAL-AMT.
```

Comments

You can place COBOL Comment lines within SQL statements. COBOL Comment lines start with an asterisk (*) in column 7. You can also place ANSI SQL-style Comments (-- ...) within SQL statements at the end of a line (but not after the last line of the SQL statement), and you can place C-style Comments (/* ... */) in SQL statements.

The following example shows all three styles of Comments:

```
    EXEC SQL SELECT ENAME, SAL
*   assign column values to output host variables
        INTO :EMP-NAME, :SALARY     -- output host variables
    /* column values assigned to output host variables */
        FROM EMP
        WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.    -- illegal Comment
```

However, you cannot nest Comments or place them on the last line of a SQL statement after the terminator END-EXEC.

Continuation Lines

You can continue SQL statements from one line to the next, according to the rules of COBOL, as this example shows:

```
EXEC SQL SELECT ENAME, SAL INTO :EMP-NAME, :SALARY FROM EMP
    WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

No continuation indicator is needed.

To continue a string literal from one line to the next, code the literal through column 72. On the next line, code a hyphen (-) in column 7, a quote in column 12 or beyond, and then the rest of the literal. An example follows:

```
WORKING STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        ...
        01  UPDATE-STATEMENT  PIC X(80) VALUE "UPDATE EMP SET BON
-               "US = 500 WHERE DEPTNO = 20".
    EXEC SQL END DECLARE SECTION END-EXEC.
```

Delimiters

The LITDELIM option specifies the delimiters for COBOL string constants and literals. If you specify LITDELIM=APOST, the Pro*COBOL uses apostrophes when generating COBOL code. If you specify LITDELIM=QUOTE (default), quotation marks are used, as in

```
CALL "SQLROL" USING SQL-TMP0.
```

In SQL statements, you must use quotation marks to delimit identifiers containing special or lowercase characters, as in

```
EXEC SQL CREATE TABLE "Emp2" END-EXEC.
```

However, you must use apostrophes to delimit string constants, as in

```
EXEC SQL SELECT ENAME FROM EMP WHERE JOB = 'CLERK' END-EXEC.
```

Regardless of which delimiter is used in the Pro*COBOL source file, Pro*COBOL generates the delimiter specified by the LITDELIM value.

Embedded SQL Syntax

To use a SQL statement in your Pro*COBOL program, precede the SQL statement with the EXEC SQL clause, and end the statement with the END-EXEC keyword. Embedded SQL syntax is described in the *Oracle8 Server SQL Reference*.

Figurative Constants

Figurative constants, such as HIGH-VALUE, ZERO, and SPACE, cannot be used in SQL statements. For example, the following is *invalid*:

```
EXEC SQL DELETE FROM EMP WHERE COMM = ZERO END-EXEC.
```

Instead, use the following:

```
EXEC SQL DELETE FROM EMP WHERE COMM = 0 END-EXEC.
```

File Length

Pro*COBOL cannot process arbitrarily long source files. Some of the variables used internally limit the size of the generated file. There is no absolute limit to the number of lines allowed, but the following aspects of the source file are contributing factors to the file-size constraint:

- complexity of the embedded SQL statements (for example, the number of bind and define variables)

- whether a database name is used (for example, connecting to a database with an AT clause)

- number of embedded SQL statements

To prevent problems related to this limitation, use multiple program units to sufficiently reduce the size of the source files.

Host Variable Names

Any valid standard COBOL identifier can be used as a host variable. Variable names can be any length, but only the first 30 characters are significant. The maximum number of significant characters recognized by COBOL compilers is 30.

Hyphenated Names

You can use hyphenated host-variable names in static SQL statements but *not* in dynamic SQL. For example, the following usage is *invalid*:

```
MOVE "DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER" TO SQLSTMT.
    EXEC SQL PREPARE STMT1 FROM SQLSTMT END-EXEC.
```

Level Numbers

When declaring host variables, you can use level numbers 01 through 49, and 77. Pro*COBOL does not allow variables containing the VARYING clause, or SQL-CURSOR variables to be declared level 49 or 77.

MAXLITERAL Default

With the MAXLITERAL option, you can specify the maximum length of string literals generated by Pro*COBOL, so that compiler limits are not exceeded. For Pro*COBOL, the default value is 256, but you might have to specify a lower value.

Multi-Byte (NCHAR) Datatypes

ANSI standard National Character (NCHAR) datatypes are supported for handling multi-byte character data. The PIC N or PIC G clause declares variables that store fixed-length NCHAR strings. You can store variable-length, multi-byte NCHAR strings using COBOL group items consisting of a length field and a string field, or using the modifier VARYING.

Beginning with Oracle8, the environmental variable NLS_NCHAR is made available to specify a client-side National Character Set.

When NLS_LOCAL=YES

When NLS_LOCAL=YES, because dynamic SQL statements are not processed at precompile time, and the Oracle8 Server does not itself process multi-byte NLS strings, you cannot embed multi-byte NLS strings in dynamic SQL statements.

Also, when NLS_LOCAL=YES, columns storing multi-byte NLS data cannot be used in embedded data definition language (DDL) statements. This restriction cannot be enforced when precompiling, so the use of these column types within embedded DDL statements results in an execution error rather than a precompile error.

Nulls

In SQL, a null represents a missing, unknown, or inapplicable column value; it equates neither to zero nor to a blank. Use the NVL function to convert nulls to non-null values, use the IS [NOT] NULL comparison operator to search for nulls, and use indicator variables to insert and test for nulls.

Paragraph Names

You can associate standard COBOL paragraph names with SQL statements, as shown in the following example:

```
LOAD-DATA.
    EXEC SQL
        INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
            VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
    END-EXEC.
```

Also, you can reference paragraph names in a WHENEVER ... DO or WHENEVER ... GOTO statement, as the next example shows:

```
PROCEDURE DIVISION.
MAIN.
    EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
    ...
SQL-ERROR.
    ...
```

You must begin all paragraph names in columns 8 through 11.

REDEFINES Clause

You can use the COBOL REDEFINES clause to redefine group or elementary items. For example, the following declarations are valid:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01  REC-ID   PIC X(4).
    01  REC-NUM  REDEFINES  REC-ID  PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```

And:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01  STOCK.
        05  DIVIDEND     PIC X(5).
        05  PRICE        PIC X(6).
    01  BOND  REDEFINES  STOCK.
        05  COUPON-RATE  PIC X(4).
        05  PRICE        PIC X(7).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Pro*COBOL issues no warning or error if a single INTO clause uses items from both a group item host variable and from its re-definition.

Relational Operators

COBOL relational operators differ from their SQL equivalents, as shown in
Table 3–1. Furthermore, COBOL allows the use of words instead of symbols,
whereas SQL does not.

Table 3–1   Relational Operators

SQL Operators	COBOL Operators
=	=, EQUAL TO
< >, !=, ^=	NOT=, NOT EQUAL TO
>	>, GREATER THAN
<	<, LESS THAN
>=	>=, GREATER THAN OR EQUAL TO
<=	<=, LESS THAN OR EQUAL TO

Sentence Terminator

A COBOL *sentence* includes one or more COBOL and/or SQL statements and ends with a
period. In conditional sentences, only the last statement must end with a period, as the fol-
lowing example shows:

```
IF EMP-NUMBER = ZERO
    MOVE FALSE TO VALID-DATA
    PERFORM GET-EMP-NUM UNTIL VALID-DATA = TRUE
ELSE
    EXEC SQL DELETE FROM EMP
        WHERE EMPNO = :EMP-NUMBER
    END-EXEC
    ADD 1 TO DELETE-TOTAL.
END-IF.
```

With COBOL-74, however, if you use WHENEVER ... GOTO or WHENEVER ...
STOP to handle errors for a SQL statement, the SQL statement must be terminated
by a period or followed by an ELSE.

The DELETE statement below is repositioned to meet this requirement:

```
EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
    IF EMP-NUMBER = ZERO
        MOVE FALSE TO VALID-DATA
```

```
                              PERFORM GET-EMP-NUM UNTIL VALID-DATA = TRUE
                      ELSE
                          ADD 1 TO DELETE-TOTAL
                          EXEC SQL DELETE FROM EMP
                              WHERE EMPNO = :EMP-NUMBER
                          END-EXEC.
```

Alternatively, you can place the SQL statement in a separate paragraph and PER-FORM that paragraph.

FILLER is Allowed

The word FILLER is allowed in host variable declarations. The word FILLER is used to specify an elementary item of a group that cannot be referred to explicitly. The following declaration is valid:

```
01  STOCK.
    05  DIVIDEND     PIC X(5).
    05  FILLER       PIC X.
    05  PRICE        PIC X(6).
```

Required Declarations and SQL Statements

Passing data between Oracle8 and your application program requires host vari-ables and error handling. This section shows you how to meet these requirements.

Declare Section is Optional

When DECLARE_SECTION is set to NO (the default), the Declare Section is optional. This is a change from Pro*COBOL prior to release 8.0. (See Chapter 7, "Running the Pro*COBOL Precompiler" for details of the precompiler options.)

If DECLARE_SECTION is YES, you must declare all program variables used in SQL statements in the *Declare Section*, which begins with the statement

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

and ends with the statement

```
EXEC SQL END DECLARE SECTION END-EXEC.
```

Between these two statements only the following are allowed:

■ host-variable and indicator-variable declarations

- non-host COBOL variables

- EXEC SQL DECLARE statements

- EXEC SQL INCLUDE statements

- EXEC SQL VAR statements

- EXEC ORACLE statements

- COBOL Comments

If DECLARE_SECTION is set to NO, you may or may not use a Declare Section. Declarations of host variables and indicator variables can be made either inside or outside a Declare Section.

Precompiler Option `DECLARE_SECTION`

For backward compatibility with releases prior to 8.0, Pro*COBOL provides this command-line option for explicit control over whether only declarations in the Declare Section are allowed as host variables. This option is

DECLARE_SECTION = YES | NO (default NO)

You must use the DECLARE_SECTION option on the command line or in a configuration file. When MODE=ORACLE and DECLARE_SECTION=YES, only variables declared inside the Declare Section are allowed as host variables. When MODE=ANSI then DECLARE_SECTION is implicitly set to YES. See the discussion of macro and micro options in "Macro and Micro Options" on page 7-4.

Multiple Declare Sections are allowed per precompiled unit. Furthermore, a host program can contain several independently precompiled units.

An Example

In the following example, you declare four host variables for use later in your program.

```
 WORKING-STORAGE SECTION.
   ...
* The next line is optional
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       ...
       01 EMP-NUMBER      PIC 9(4)  COMP VALUE ZERO.
       01 EMP-NAME        PIC X(10) VARYING.
       01 SALARY          PIC S9(5)V99 COMP-3 VALUE ZERO.
       01 COMMISSION      PIC S9(5)V99 COMP-3 VALUE ZERO.
* The next line is optional
```

```
      EXEC SQL END DECLARE SECTION END-EXEC.
```

Using the INCLUDE Statement

The INCLUDE statement lets you copy files into your host program, as the following example shows:

```
*     Copy in the SQL Communications Area (SQLCA)
      EXEC SQL INCLUDE SQLCA END-EXEC.
*     Copy in the Oracle Communications Area (ORACA)
      EXEC SQL INCLUDE ORACA END-EXEC.
```

You can INCLUDE any file. When you precompile your Pro*COBOL program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

Filename Extensions

If your system uses file extensions but you do not specify one, Pro*COBOL assumes the default extension for source files (usually COB). The default extension is system-dependent. For more information, see your Oracle system-specific documentation.

Search Paths

If your system uses directories, you can set a search path for included files using the INCLUDE option, as follows:

INCLUDE=*path*

where *path* defaults to the current directory.

Pro*COBOL first searches the current directory, then the directory specified by the INCLUDE option, and finally the directory for standard INCLUDE files. You need not specify a path for standard files such as the SQLCA and ORACA. However, a path is required for nonstandard files unless they are stored in the current directory.

You can also specify multiple paths on the command line, as follows:

```
... INCLUDE=<path1> INCLUDE=<path2> ...
```

When multiple paths are specified, Pro*COBOL searches the current directory first, then the *path1* directory, then the *path2* directory, and so on. The directory containing standard INCLUDE files is searched last. The path syntax is system specific. For more information, see your Oracle system-specific documentation.

Caution

Remember that Pro*COBOL searches for a file in the current directory first even if you specify a search path. If the file you want to INCLUDE is in another directory, make sure no file with the same name is in the current directory or any other directory that precedes it in the search path. If your operating system is case sensitive, be sure to specify the same upper/lowercase filename under which the file is stored.

Error Handling

Oracle returns the success or failure of SQL statements in status variables, SQL-STATE and SQLCODE. With Oracle mode, you can declare SQLCODE by including the SQLCA. With ANSI mode. you must declare either SQLSTATE or SQLCODE. For more information, see Chapter 9, "Error Handling and Diagnostics".

Host Variables

Host variables are the key to communication between your host program and Oracle8. Typically, a host program inputs data to Oracle8, and Oracle8 outputs data to the program. Oracle8 stores input data in database columns and stores output data in program host variables.

Declaring Host Variables

Host variables are declared according to COBOL rules, using the COBOL datatypes that are supported by Oracle8. COBOL datatypes must be compatible with the source/target database column.

The supported COBOL datatypes are shown in Table 3–2

Table 3–2   Host Variable Declarations

Variable Declaration	Description
PIC X...X PIC X(*n*) PIC X...X VARYING PIC X(*n*) VARYING	fixed-length string of 1-byte characters (1) *n*-length string of 1-byte characters variable-length string of 1-byte characters (1,2) variable-length (*n* max.) string of 1-byte characters (2)
PIC N...N PIC G...G PIC N(*n*) PIC G(n) PIC N...N VARYING PIC N(n) VARYING PIC G...G VARYING PIC G(n) VARYING	fixed-length string of multi-byte NCHAR characters (1,3) *n*-length string of multi-byte NCHAR characters (3) variable-length string of multi-byte characters (2,3) variable-length (*n* max.) string of multi-byte characters (2,3)
PIC S9...9 BINARY PIC S9(*n*) BINARY PIC S9...9 COMP PIC S9(*n*) COMP PIC S9...9 COMP-4 PIC S9(*n*) COMP-4	integer (4,5,7)
COMP-1 COMP-2	floating-point number (5)

PIC S9...9V9...9 COMP-3
PIC S9(*n*)V9(*n*) COMP-3
PIC S9...9V9...9
    PACKED-DECIMAL
PIC S9(*n*)V9(*n*)
    PACKED-DECIMAL
	packed-decimal (4,5)

Table 3–2   Host Variable Declarations

PIC S9...9 COMP-5 PIC S9(*n*) COMP-5	byte-swapped integer (4,5,6,7)

PIC S9...9V9...9 DISPLAY
          SIGN LEADING SEPARATE
PIC S9(*n*)V9(*m*) DISPLAY
          SIGN LEADING SEPARATE
	display leading (9,12)

PIC S9...9V9...9 DISPLAY
          SIGN TRAILING SEPARATE
PIC S9(*n*)V9(*m*) DISPLAY
          SIGN TRAILING SEPARATE
	display trailing (9)

PIC 9...9 DISPLAY PIC 9(n)V9(m) DISPLAY	unsigned display(10)

PIC S9...9V9...9 DISPLAY
          SIGN TRAILING
PIC S9(n)V9(m) DISPLAY
          SIGN TRAILING
PIC S9...9V9...9 DISPLAY
          SIGN LEADING
PIC S9(n)V9(m) DISPLAY
          SIGN LEADING
	over-punch trailing (10,11
	
	over-punch leading (10))

SQL-CURSOR	cursor variable

Notes:

1. X...X and 9...9 stand for a given number (*n*) of Xs or 9s. For variable-length strings, *n* is the maximum length.

2. The keyword VARYING assigns the VARCHAR external datatype to a character string. For more information, see "Declaring VARCHAR Variables" on page 3-36.

3. Before using the PIC N or PIC G datatype in your Pro*COBOL source files, verify that it is supported by your COBOL compiler.

4. Only signed numbers (PIC S...) are allowed. For floating-point numbers, however, PIC strings are not accepted.

5. Not all COBOL compilers support all of these datatypes.

6. With COMP or COMP-5, the number cannot have a fractional part; scaled binary numbers are not supported.

7. The maximum value of *n* ranges from 9 to 18, depending upon your system.

8. One-dimensional tables of COBOL types are also supported.

9. Both DISPLAY and SIGN are optional.

10. DISPLAY is optional

11. If TRAILING is omitted, the embedded sign position is operating-system dependent.

12. LEADING is optional.

Table 3–3 shows the compatible Oracle8 internal datatypes.

Table 3–3   Compatible Oracle8 Internal Datatypes

Internal Datatype		COBOL Datatype	Description
CHAR(*x*)	(13)	PIC [X...X | N...N | G...G]	character string
VARCHAR2(*y*)	(13)	PIC [X(*n*) | N(*n*) | G(n)]	*n*-character string
		PIC [X(*n*) | X(*n*)] VARYING	variable-length string
		PIC S9...9 COMP	integer
		PIC S9(*n*) COMP	
		PIC S9...9 BINARY	integer
		PIC S9(*n*) BINARY	
		PIC S9...9 COMP-5	integer
		PIC S9(*n*) COMP-5	
		COMP-1	floating point number
		COMP-2	
		PIC S9...9V9...9 COMP-3	packed decimal
		PIC S9(*n*)V9(*n*) COMP-3	
		PIC S9...9V9...9 DISPLAY	display
		PIC S9(*n*)V9(*n*) DISPLAY	

Table 3–3   Compatible Oracle8 Internal Datatypes

Internal Datatype		COBOL Datatype	Description
NUMBER		PIC S9...9 COMP	integer
NUMBER (*p,s*)	(14)	PIC S9(*n*) COMP	
		PIC S9...9 BINARY	integer
		PIC S9(*n*) BINARY	
		PIC S9...9 COMP-5	integer
		PIC S9(*n*) COMP-5	
		COMP-1	floating point number
		COMP-2	
		PIC S9...9V9...9 COMP-3	packed decimal
		PIC S9(*n*)V9(*n*) COMP-3	
		PIC S9...9V9...9 DISPLAY	display
		PIC S9(*n*)V9(*n*) DISPLAY	
		PIC [X...X | N...N | G...G]	character string (15)
		PIC [X(*n*) | N(*n*) | G(*n*)]	*n*-character string (15)
		PIC X...X VARYING	variable-length string
		PIC X(*n*) VARYING	*n*-byte variable-length string
DATE	(16)	PIC X(*n*)	*n*-byte character string
LONG			
RAW	(13)	PIC X...X VARYING	*n*-byte variable-length string
LONG RAW			
ROWID	(17)		
MLSLABEL	(18)		

Notes:

13. *x* ranges from 1 to 255, and 1 is the default. *y* ranges from 1 to 4000.

14. *p* ranges from 2 to 38. *s* ranges from -84 to 127.

15. Strings can be converted to NUMBERs only if they consist of convertible characters — 0 to 9, period (.), +, -, E, e. The NLS settings for your system might change the decimal point from a period (.) to a comma (,).

16. When converted to a string type, the default size of a DATE depends on the NCHAR settings in effect on your system. When converted to a binary value, the length is 7 bytes.

17. When converted to a string type, a ROWID requires from 18 to 256 bytes.

18. Trusted Oracle only.

Example Declarations

In the following example, you declare several host variables for use later in your Pro*COBOL program:

```
...
    01  STR1  PIC X(3).
    01  STR2  PIC X(3) VARYING.
    01  NUM1  PIC S9(5) COMP.
    01  NUM2  COMP-1.
    01  NUM3  COMP-2.
...
```

You can also declare one-dimensional tables of simple COBOL types, as the next example shows:

```
...
    01  XMP-TABLES.
        05  TAB1  PIC XXX OCCURS 3 TIMES.
        05  TAB2  PIC XXX VARYING OCCURS 3 TIMES.
        05  TAB3  PIC S999 COMP-3 OCCURS 3 TIMES.
...
```

Initialization

No error or warning is issued, but any VALUES clause on a pseudo-type variable is ignored and discarded.

You can initialize host variables, except pseudo-type host variables, using the VALUE clause, as shown in the following example:

```
        01  USERNAME   PIC X(10) VALUE "SCOTT".
        01  MAX-SALARY  PIC S9(4) COMP VALUE 5000.
```

If a string value assigned to a character variable is shorter than the declared length of the variable, the string is blank-padded on the right. If the string value assigned to a character variable is longer than the declared length, the string is truncated.

Restrictions

You cannot use alphabetic character (PIC A) variables or edited data items as host variables. Therefore, the following variable declarations cannot be made for *host* variables:

```
....
    01  AMOUNT-OF-CHECK  PIC ****9.99.
    01  FIRST-NAME       PIC A(10).
    01  BIRTH-DATE       PIC 99/99/99.

....
```

Referencing Host Variables

You use host variables in SQL data manipulation statements. A host variable must be prefixed with a colon (:) in SQL statements but must not be prefixed with a colon in COBOL statements, as this example shows:

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  EMP-NUMBER  PIC S9(4) COMP VALUE ZERO.
        01  EMP-NAME    PIC X(10) VALUE SPACE.
        01  SALARY      PIC S9(5)V99 COMP-3.
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    DISPLAY "Employee number? " WITH NO ADVANCING.
    ACCEPT EMP-NUMBER.
    EXEC SQL SELECT ENAME, SAL
        INTO :EMP-NAME, :SALARY FROM EMP
        WHERE EMPNO = :EMP-NUMBER
    END-EXEC.
    COMPUTE BONUS = SALARY / 10.
    ...
```

Though it might be confusing, you can give a host variable the same name as an Oracle8 table or column, as the following example shows:

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  EMPNO  PIC S9(4) COMP VALUE ZERO.
        01  ENAME  PIC X(10) VALUE SPACE.
        01  COMM   PIC S9(5)V99 COMP-3.
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    EXEC SQL SELECT ENAME, COMM
        INTO :ENAME, :COMM FROM EMP
        WHERE EMPNO = :EMPNO
    END-EXEC.
```

Group Items as Host Variables

Pro*COBOL now allows the use of group items in embedded SQL statements. Group items with elementary items (containing only one level) can be used as host variables. The host group items (also referred to as host structures) can be referenced in the INTO clause of a SELECT or a FETCH statement, and in the VALUES list of an INSERT statement. When a group item is used as a host variable, only the group name is used in the SQL statement. For example, given the following declaration

```
        01  DEPARTURE.
            05 HOUR    PIC X(2).
            05 MINUTE  PIC X(2).
```

the following statement is valid:

```
        EXEC SQL SELECT DHOUR, DMINUTE
            INTO :DEPARTURE
            FROM SCHEDULE
            WHERE ...
```

The order that the members are declared in the group item must match the order that the associated columns occur in the SQL statement, or in the database table if the column list in the INSERT statement is omitted. Using a group item as a host variable has the semantics of substituting the group item with elementary items. In

the above example, it would mean substituting :DEPARTURE with :DEPAR-TURE.HOUR, :DEPARTURE.MINUTE.

Group items used as host variables can contain host tables. In the following example, the group item containing tables is used to INSERT three entries into the SCHEDULE table:

```
01  DEPARTURE.
    05  HOUR    PIC X(2) OCCURS 3 TIMES.
    05  MINUTE  PIC X(2) OCCURS 3 TIMES.
...
EXEC SQL INSERT INTO SCHEDULE ( DHOUR, DMINUTE)
    VALUES ( :DEPARTURE) END-EXEC.
```

If VARCHAR=YES is specified, Pro*COBOL will recognize implicit VARCHARs. If the nested group item declaration resembles a VARCHAR host variable, then the entire group item is treated like an elementary item of VARYING type. See "VAR-CHAR" on page 7-38.

When referencing elementary items instead of the group items as host variables elementary names need not be unique because you can qualify them using the following syntax:

```
<group_item>.<elementary_item>
```

This naming convention is allowed only in SQL statements. It is similar to the IN (or OF) clause in COBOL, examples of which follow:

```
MOVE MINUTE IN DEPARTURE TO MINUTE-OUT.
DISPLAY HOUR OF DEPARTURE.
```

The COBOL IN (or OF) clause is *not* allowed in SQL statements. Qualify elementary names to avoid ambiguity. For example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01  DEPARTURE.
        05  HOUR    PIC X(2).
        05  MINUTE  PIC X(2).
    01  ARRIVAL.
        05  HOUR    PIC X(2).
        05  MINUTE  PIC X(2).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Restrictions

A host variable cannot substitute for a column, table, or other Oracle8 object in a SQL statement and must not be an Oracle8 reserved word. See Appendix C, "Oracle8 Reserved Words, Keywords, and Namespaces" for a list of Oracle8 reserved words and keywords.

Nested Programs

Nesting programs in COBOL means that you place one program inside another. The contained programs may reference some of the resources of the programs within which they are contained. The names within the higher-level program and the nested program can be the same, and describe different data items without conflict, because the names are known only within the programs. However, names described in the Configuration Section of the higher-level program can be referenced in the nested program.

The higher-level program can contain several nested programs. Likewise, nested programs can have programs nested within them. You must place the nested program directly before the END PROGRAM header of the program in which it is nested.

You can call a nested program only by a program in which it is either directly or indirectly nested. If you want a nested program to be called by any program, even one on a different branch of the nested tree structure, you code the COMMON clause in the PROGRAM-ID paragraph of the nested program. You can code COMMON only for nested programs:

```
PROGRAM-ID.   <nested-program-name> COMMON.
```

You can code the GLOBAL phrase for File Definitions and level 01 data items (any subordinate items automatically become global). This allows them to be referenced in all subprograms directly or indirectly contained within them. You code GLOBAL on the higher-level program. If the nested program defines the same name as one declared GLOBAL in a higher-level program, COBOL uses the declaration within the nested program. If the data item contains a REDEFINES clause, GLOBAL must follow it.

```
FD file-name GLOBAL ...
    01 data-name1 GLOBAL ...
    01 data-name2 REDEFINES data-name3 GLOBAL ...
```

Support for Nested Programs

Pro*COBOL allows nested programs with embedded SQL within a single source
file. All 01 level items which are marked as global in a containing program and are
valid host variables at the containing program level are usable as valid host vari-
ables in any programs directly or indirectly contained by the containing program.
Consider the following example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      01 REC1  GLOBAL.
          05    VAR1   PIC X(10).
          05    VAR2   PIC X(10).
      01 VAR1  PIC X(10) GLOBAL.
    EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
    ...
    <main program statements>
    ...
IDENTIFICATION DIVISION.
   PROGRAM-ID. NESTEDPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

     01 VAR1   PIC S9(4).

PROCEDURE DIVISION.
    ...
    EXEC SQL SELECT X, Y INTO :REC1 FROM ... END-EXEC.

    EXEC SQL SELECT X INTO :VAR1 FROM ... END-EXEC.

    EXEC SQL SELECT X INTO :REC1.VAR1 FROM ... END-EXEC.
    ...
END PROGRAM NESTEDPROG.
END PROGRAM MAINPROG.
```

The main program declares the host variable REC1 as global and thus the nested program can use REC1 in the first select statement without having to declare it. Since VAR1 is declared as a global variable and also as a local variable in the nested program, the second select statement will use the VAR1 declared as S9(4), overriding the global declaration. In the third select statement, the global VAR1 of REC1 declared as PIC X(10) is used.

The previous paragraph describes the results when DECLARE_SECTION=NO is used. When DECLARE_SECTION=YES, Pro*COBOL will not recognize host variables *unless* they are declared inside a Declare Section. If the above program is precompiled with DECLARE_SECTION=YES, then the second select statement would result in an ambiguous host variable error. The first and third select statements would function the same.

Note: Recursive nested programs are not supported

Declaring the SQLCA

About declaring the SQLCA for nested programs, (see "SQLCA" on page 9-3 and later), the included SQLCA definition provided will be declared as global, so the declaration of SQLCA is only required in the higher-level program. The SQLCA can change each time a new SQL statement is executed. The SQLCA provided can always be modified to remove the global specification if you want to declare additional SQLCA areas in the nested programs. The same will apply to SQLDA and ORACA.

Sample Nested Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  NESTED.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME          PIC X(10) VARYING.
        01  PASSWD            PIC X(10) VARYING.
        01  GEMP-REC-VAR1 GLOBAL.
            05  EMP-NUM        PIC S9(4) COMP.
            05  EMP-NAME       PIC X(10) VARYING.
            05  SALARY         PIC S9(6)V99
                DISPLAY SIGN LEADING SEPARATE.
            05  COMMISSION     PIC S9(6)V99
                DISPLAY SIGN LEADING SEPARATE.
        01  EMP-NAME          PIC X(10) VARYING GLOBAL.
```

```
                01  EMP-NUM          PIC S9(4) COMP GLOBAL.
                01  EMP-REC-VAR1.
                    05  EMP-NUM      PIC S9(4) COMP.
                    05  EMP-NAME     PIC X(10) VARYING.
                    05  SALARY       PIC S9(6)V99
                        DISPLAY SIGN LEADING SEPARATE.
                    05  COMMISSION   PIC S9(6)V99
                        DISPLAY SIGN LEADING SEPARATE.
                01  GEMP-REC-IND1 GLOBAL.
                    05  EMP-NUM-IND  PIC S9(4) COMP.
                    05  EMP-NAME-IND PIC S9(4) COMP.
                    05  EMP-SAL-IND  PIC S9(4) COMP.
                    05  EMP-COMM-IND PIC S9(4) COMP.
           EXEC SQL END DECLARE SECTION END-EXEC.

           EXEC SQL INCLUDE SQLCA END-EXEC.

                01  DISPLAY-VARIABLES GLOBAL.
                    05  D-EMP-NUM    PIC Z(3)9.
                    05  D-EMP-NAME   PIC X(10).
                    05  D-SALARY     PIC Z(4)9.99.
                    05  D-COMMISSION PIC Z(4)9.99.

       PROCEDURE DIVISION.
       BEGIN-PGM.
           EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.

           PERFORM LOGON.

           DISPLAY "In Main Program".

           CALL "INNER1".
           CALL "NESTED1".

           GO TO SIGN-OFF.

       LOGON.
           MOVE "SCOTT" TO USERNAME-ARR.
           MOVE 5 TO USERNAME-LEN.
           MOVE "TIGER" TO PASSWD-ARR.
           MOVE 5 TO PASSWD-LEN.
           EXEC SQL
               CONNECT :USERNAME IDENTIFIED BY :PASSWD
           END-EXEC.
           DISPLAY " ".
```

```
                     DISPLAY "CONNECTED TO ORACLE AS USER:  ", USERNAME-ARR.
                     DISPLAY " ".

              SIGN-OFF.
                     DISPLAY " ".
                     DISPLAY "HAVE A GOOD DAY.".
                     DISPLAY " ".
                     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
                     STOP RUN.

              SQL-ERROR.
                     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
                     DISPLAY " ".
                     DISPLAY "ORACLE ERROR DETECTED:".
                     DISPLAY " ".
                     DISPLAY SQLERRMC.
                     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
                     STOP RUN.

              IDENTIFICATION DIVISION.
              PROGRAM-ID.  INNER1 COMMON.
              ENVIRONMENT DIVISION.
              DATA DIVISION.
              WORKING-STORAGE SECTION.
                     01  EMP-NAME IS GLOBAL PIC X(15) VARYING.
              PROCEDURE DIVISION.
              P1.
                     DISPLAY "In Inner1 Nested Program".

          *    Using a global host var.
                     EXEC SQL SELECT EMPNO, ENAME, SAL, COMM
                          INTO :GEMP-REC-VAR1:GEMP-REC-IND1
                          FROM EMP WHERE EMPNO = 7566 END-EXEC.
                     DISPLAY " ".
                     DISPLAY "EMPNO  SALESPERSON      SALARY  COMMISSION".
                     DISPLAY "-----  ----------  ----------  ----------".
                     MOVE EMP-NUM OF GEMP-REC-VAR1 TO D-EMP-NUM.
                     MOVE EMP-NAME-ARR OF GEMP-REC-VAR1 TO D-EMP-NAME.
                     MOVE SALARY OF GEMP-REC-VAR1 TO D-SALARY.
                     MOVE COMMISSION OF GEMP-REC-VAR1 TO D-COMMISSION.
                     DISPLAY D-EMP-NUM, "   ", D-EMP-NAME, "     ", D-SALARY,
                             "    ", D-COMMISSION.
                     DISPLAY "Answers should be 7566, JONES, 2975, 0".

          *    overriding global host var with a local one.
```

```
*      should use PIC X(15) decl.
       DISPLAY " ".
       EXEC SQL SELECT ENAME INTO :EMP-NAME
           FROM EMP WHERE EMPNO = 7499 END-EXEC.
       DISPLAY "Emp Name: ", EMP-NAME, "**".
       DISPLAY "Emp Name should be ALLEN".

*      Using the element of a global host var.
       DISPLAY " ".
       EXEC SQL SELECT ENAME INTO :GEMP-REC-VAR1.EMP-NAME
           FROM EMP WHERE EMPNO = 7499 END-EXEC.
       DISPLAY "Emp Name: ", EMP-NAME, "**".
       DISPLAY "Emp Name should be ALLEN".


       CALL "INNER2".

 IDENTIFICATION DIVISION.
 PROGRAM-ID.   INNER2 COMMON.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
           01  EMP-NUM         PIC S9(4) COMP.
       EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
 P2.
       DISPLAY "In Inner2 Nested Program".
*      Using a global host var even when not directly within the
*      main program.
       EXEC SQL SELECT EMPNO, ENAME, SAL, COMM
           INTO :GEMP-REC-VAR1:GEMP-REC-IND1
           FROM EMP WHERE EMPNO = 7566 END-EXEC.
       DISPLAY " ".
       DISPLAY "EMPNO  SALESPERSON     SALARY  COMMISSION".
       DISPLAY "-----  -----------  ----------  ----------".
       MOVE EMP-NUM OF GEMP-REC-VAR1 TO D-EMP-NUM.
       MOVE EMP-NAME-ARR OF GEMP-REC-VAR1 TO D-EMP-NAME.
       MOVE SALARY OF GEMP-REC-VAR1 TO D-SALARY.
       MOVE COMMISSION OF GEMP-REC-VAR1 TO D-COMMISSION.
       DISPLAY D-EMP-NUM, "   ", D-EMP-NAME, "    ", D-SALARY,
               "    ", D-COMMISSION.
       DISPLAY "Answers should be 7566, JONES, 2975, 0".

*      Using a global host var in a nested function and a
*      local host var.  Should use PIC X(15) decl from
```

```
*      INNER1 for EMP-NAME, and local EMP-NUM.
       DISPLAY " ".
       EXEC SQL SELECT ENAME, EMPNO INTO :EMP-NAME, :EMP-NUM
           FROM EMP WHERE EMPNO = 7499 END-EXEC.
       DISPLAY "Emp Name: ", EMP-NAME, "Emp Number: ", EMP-NUM.
       DISPLAY "Emp name should be ALLEN and emp number 7499".

*      Using the element of a global host var even when indirectly
*      within the main program.
       DISPLAY " ".
       EXEC SQL SELECT ENAME INTO :GEMP-REC-VAR1.EMP-NAME
           FROM EMP WHERE EMPNO = 7499 END-EXEC.
       DISPLAY "Emp Name: ", EMP-NAME, "**".
       DISPLAY "Emp Name should be ALLEN".

  END PROGRAM INNER2.
  END PROGRAM INNER1.


  IDENTIFICATION DIVISION.
  PROGRAM-ID.  NESTED1.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
        01   EMP-NAME          PIC X(15) VARYING GLOBAL.
  PROCEDURE DIVISION.
  N1.

     DISPLAY "In Nested1 Nested Program".
     DISPLAY "Calling inner1".
*    Should work since INNER1 has the COMMON clause.
     CALL "INNER1".

  END PROGRAM NESTED1.
```

When you execute the COBOL program created by running Pro*COBOL, the resulting output is as follows:

```
> nested

CONNECTED TO ORACLE AS USER:   SCOTT

In Main Program
In Inner1 Nested Program

EMPNO  SALESPERSON      SALARY  COMMISSION
-----  -----------  ----------  ----------
```

```
7566    JONES         2975.00        0.00
Answers should be 7566, JONES, 2975, 0

Emp Name: ALLEN            **
Emp Name should be ALLEN

Emp Name: ALLEN            **
Emp Name should be ALLEN
In Inner2 Nested Program

EMPNO  SALESPERSON       SALARY  COMMISSION
-----  -----------  ----------  ----------
7566    JONES         2975.00        0.00
Answers should be 7566, JONES, 2975, 0

Emp Name: ALLEN          Emp Number: +7499
Emp name should be ALLEN and emp number 7499

Emp Name: ALLEN            **
Emp Name should be ALLEN
In Nested1 Nested Program
Calling inner1
In Inner1 Nested Program

EMPNO  SALESPERSON       SALARY  COMMISSION
-----  -----------  ----------  ----------
7566    JONES         2975.00        0.00
Answers should be 7566, JONES, 2975, 0

Emp Name: ALLEN            **
Emp Name should be ALLEN

Emp Name: ALLEN            **
Emp Name should be ALLEN
In Inner2 Nested Program

EMPNO  SALESPERSON       SALARY  COMMISSION
-----  -----------  ----------  ----------
7566    JONES         2975.00        0.00
Answers should be 7566, JONES, 2975, 0

Emp Name: ALLEN          Emp Number: +7499
Emp name should be ALLEN and emp number 7499

Emp Name: ALLEN            **
```

```
Emp Name should be ALLEN

HAVE A GOOD DAY.
```

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.

Declaring Indicator Variables

An indicator variable must be explicitly declared as PIC S9(4) COMP and must not be an Oracle8 reserved word. In the following example, you declare an indicator variable named COMM-IND (the name is arbitrary):

```
WORKING-STORAGE SECTION.
    ...
    01  EMP-NAME    PIC X(10) VALUE SPACE.
    01  SALARY      PIC S9(5)V99 COMP-3.
    01  COMMISSION  PIC S9(5)V99 COMP-3.
    01  COMM-IND    PIC S9(4) COMP.
    ...
```

Referencing Indicator Variables

In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable. In COBOL statements, an indicator variable must *not* be prefixed with a colon or appended to its associated host variable. An example follows:

```
EXEC SQL SELECT SAL, COMM
    INTO :SALARY, :COMMISSION:COMM-IND FROM EMP
    WHERE EMPNO = :EMP-NUMBER
END-EXEC.
IF COMM-IND = -1
    COMPUTE PAY = SALARY
ELSE
    COMPUTE PAY = SALARY + COMMISSION.
```

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>
```

and is equivalent to

```
:<host_variable>:<indicator_variable>
```

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

Restriction

Indicator variables *cannot* be used in the WHERE clause to search for nulls. For example, the following DELETE statement triggers an error at run time:

```
*     Set indicator variable.
      COMM-IND = -1
      EXEC SQL
          DELETE FROM EMP WHERE COMM = :COMMISSION:COMM-IND
      END-EXEC.
```

The correct syntax follows:

```
      EXEC SQL
          DELETE FROM EMP WHERE COMM IS NULL
      END-EXEC.
```

Oracle8 Restrictions

If you SELECT or FETCH a null into a host variable that has no indicator, Oracle8 issues the following error message:

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

You can disable the ORA-01405 message by also specifying UNSAFE_NULL=YES on the command line. For more information, see Chapter 7, "Running the Pro*COBOL Precompiler".

ANSI Requirements

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host variable that is not associated with an indicator variable, Oracle8 issues the following error message:

```
ORA-01406: fetched column value was truncated
```
However, when MODE={ANSI | ANSI14 | ANSI13}, no error is generated. Values for indicator variables are discussed in Chapter 5, "Using Embedded SQL".

Indicator Variables for Multi-Byte NCHAR Variables

Indicator variables for multi-byte NCHAR character variables can be used as with any other host variable. However, a positive value (the result of a SELECT or FETCH was truncated) represents the string length in multi-byte characters instead of 1-byte characters.

Indicator Variables with Host Group Items

To use indicator variables with a host group item, either setup a second group item that contains an indicator variable for each nullable variable in the group item or use a table of half-word integer variables. You do NOT have to have an indicator variable for each variable in the group item, but the nullable fields which you wish to use indicators for must be placed at the beginning of the data group item. The following indicator group item can be used with the DEPARTURE group item:

```
01  DEPARTURE-IND.
    05  HOUR-IND    PIC S9(4) COMP.
    05  MINUTE-IND PIC S9(4) COMP.
```

If you use an indicator table, you do NOT have to declare a table of as many elements as there are members in the host group item. The following indicator table can be used with the DEPARTURE group item:

```
01 DEPARTURE-IND PIC S9(4) COMP OCCURS 2 TIMES.
```

Reference the indicator group item in the SQL statement in the same way that a host indicator variable is referenced:

```
EXEC SQL SELECT DHOUR, DMINUTE
    INTO :DEPARTURE:DEPARTURE-IND
    FROM SCHEDULE
    WHERE ...
```

When the query completes, the NULL/NOT NULL status of each selected component is available in the host indicator group item. The restrictions on indicator host variables and the ANSI requirements also apply to host indicator group items.

Host Tables

Host tables can improve performance by letting you manipulate an entire collection of data items with a single SQL statement. With few exceptions, you can use host tables wherever scalar host variables are allowed. Also, you can associate an indicator table with any host table.

Declaring Host Tables

You declare and dimension host tables in the Data Division. In the following example, three host tables are declared, each dimensioned with 50 elements:

```
....
    01  EMP-TABLES.
        05  EMP-NUMBER  OCCURS 50 TIMES PIC S9(4) COMP.
        05  EMP-NAME    OCCURS 50 TIMES PIC X(10.
        05  SALARY      OCCURS 50 TIMES PIC S9(5)V99 COMP-3.
....
```

You can use the INDEXED BY phrase in the OCCURS clause to specify an index, as the next example shows:

```
...
    01  EMP-TABLES.
        05  EMP-NUMBER  PIC X(10) OCCURS 50 TIMES
                                    INDEXED BY EMP-INDX.
        ...
...
```

The INDEXED BY phrase implicitly declares the index item EMP-INDX.

Restrictions

Multi-dimensional host tables are not allowed. Thus, the two-dimensional host table declared in the following example is *invalid*:

```
...
    01  NATION.
        05  STATE               OCCURS 50 TIMES.
            10  STATE-NAME      PIC X(25).
            10  COUNTY          OCCURS 25 TIMES.
                15  COUNTY-NAME  PIX X(25).
...
```

Variable-length host tables are not allowed either. For example, the following declaration of EMP-REC is *invalid for a host variable*:

```
...
```

```
                    01  EMP-FILE.
                        05  REC-COUNT  PIC S9(3) COMP.
                        05  EMP-REC    OCCURS 0 TO 250 TIMES
                                       DEPENDING ON REC-COUNT.
            ...
```

Referencing Host Tables

If you use multiple host tables in a single SQL statement, their dimensions should
be the same. This is not a requirement, however, because Pro*COBOL always uses
the *smallest* dimension for the SQL operation. In the following example, only 25 rows are
INSERTed:

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
          01  EMP-TABLES.
              05  EMP-NUMBER   PIC S9(4) COMP OCCURS 50 TIMES.
              05  EMP-NAME     PIC X(10) OCCURS 50 TIMES.
              05  DEPT-NUMBER  PIC S9(4) COMP OCCURS 25 TIMES.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     ...
*     Populate host tables here.
     ...
     EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
         VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
     END-EXEC.
```

Host tables must *not* be subscripted in SQL statements. For example, the following
INSERT statement is *invalid*:

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
          01  EMP-TABLES.
              05  EMP-NUMBER   PIC S9(4) COMP OCCURS 50 TIMES.
              05  EMP-NAME     PIC X(10) OCCURS 50 TIMES.
              05  DEPT-NUMBER  PIC S9(4) COMP OCCURS 50 TIMES.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     ...
     PERFORM LOAD-EMP VARYING J FROM 1 BY 1 UNTIL J > 50.
     ...
 LOAD-EMP.
     EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
```

```
              VALUES (:EMP-NUMBER(J), :EMP-NAME(J),
                     :DEPT-NUMBER(J))
       END-EXEC.
```

You need not process host tables in a PERFORM VARYING statement. Instead, use the un-subscripted table names in your SQL statement. Oracle8 treats a SQL statement containing host tables of dimension n like the same statement executed n times with n different scalar host variables, except its more efficient. For more information, see "Host Tables" on page 3-33.

Using Indicator Tables

You can use indicator tables to assign nulls to elements in input host tables and to detect nulls or truncated values in output host tables. The following example shows how to INSERT with indicator tables:

```
WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01   EMP-TABLES.
     05   EMP-NUMBER   PIC S9(4) COMP OCCURS 50 TIMES.
     05   DEPT-NUMBER  PIC S9(4) COMP OCCURS 50 TIMES.
     05   COMMISSION   PIC S9(5)V99 COMP-3 OCCURS 50 TIMES.
     05   COMM-IND     PIC S9(4) COMP OCCURS 50 TIMES.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     ...
*    Populate the host and indicator tables.
*    Set indicator table to all zeros.
     ...
     EXEC SQL INSERT INTO EMP (EMPNO, DEPTNO, COMM)
         VALUES (:EMP-NUMBER, :DEPT-NUMBER,
             :COMMISSION:COMM-IND)
     END-EXEC.
```

The dimension of the indicator table must be greater than, or equal to, the dimension of the host table.

Host Group Item Containing Tables

Note: If you have a host group item containing tables, you cannot use a table of half-word integer variables for an indicator. You must use a corresponding group item of tables for an indicator. For example, if your group item is the following:

```
 01   DEPARTURE.
```

```
05 HOUR    PIC X(2) OCCURS 3 TIMES.
05 MINUTE  PIC X(2) OCCURS 3 TIMES.
```

the following indicator variable *cannot* be used:

```
01  DEPARTURE-IND PIC S9(4) COMP OCCURS 6 TIMES.
```

The indicator variable you use with the group item of tables must itself be a group item of tables such as the following:

```
01  DEPARTURE-IND.
    05 HOUR-IND   PIC S9(4) COMP OCCURS 3 TIMES.
    05 MINUTE-IND PIC S9(4) COMP OCCURS 3 TIMES.
```

VARCHAR Variables

COBOL string datatypes are fixed length. However, Pro*COBOL lets you declare a variable-length string pseudotype called VARCHAR.

Declaring VARCHAR Variables

You define a VARCHAR host variable by adding the keyword VARYING to its declaration, as shown in the following example:

```
01  ENAME  PIC X(15) VARYING.
```

The COBOL VARYING phrase is used in PERFORM and SEARCH statements to increment subscripts and indexes. Do not confuse this with the Pro*COBOL VARYING clause in the preceding example.

VARCHAR is an extended Pro*COBOL datatype or pre-declared group item. For example, Pro*COBOL expands the VARCHAR declaration

```
01  ENAME  PIC X(15) VARYING.
```

into a group item with length and string fields, as follows:

```
01  ENAME.
    05  ENAME-LEN  PIC S9(4) COMP.
    05  ENAME-ARR  PIC X(15).
```

The *length* field (suffixed with -LEN) holds the current length of the value stored in the *string* field (suffixed with -ARR). The maximum length in the VARCHAR host-variable declaration must be in the range of 1 to 65533 bytes.

The advantage of using VARCHAR variables is that you can explicitly set and reference the length field. With input host variables, Oracle8 reads the value of the length field and uses that many characters of the string field. With output host variables, Oracle8 sets the length value to the length of the character string stored in the string field.

Implicit VARCHAR Group Items

Pro*COBOL implicitly recognizes some group items as VARCHAR host variables when the precompiler option VARCHAR=YES is specified on the command line. For variable-length single-byte character types, use the following structure (*length* expressed in single-byte characters):

```
<nn>  DATA-NAME-1.
      49  DATA-NAME-2 PIC S9(4) COMP.
      49  DATA-NAME-3   PIC X(<length>).
```

nn must be 01 through 48. For variable-length multi-byte NCHAR character types, use these formats (*length* expressed in *double-byte* characters):

```
<nn>  DATA-NAME-1.
      49 DATA-NAME-2 PIC S9(4) COMP.
      49  DATA-NAME-3   PIC N(<length>).
```

or,

```
<nn>  DATA-NAME-1.
      49 DATA-NAME-2 PIC S9(4) COMP.
      49 DATA-NAME-3 PIC G(<length>).
```

The elementary items in these group-item structures *must* be declared as level 49 for Pro*COBOL to recognize them as VARCHAR host variables.

The VARCHAR=YES command line option must be specified for Pro*COBOL to recognize the extended form of the VARCHAR group items. If VARCHAR=NO, then any declarations that resemble the above formats will be interpreted as regular group items. If VARCHAR=YES and a group item declaration format looks similar (but not identical) to the extended VARCHAR format, then the item will be interpreted as a regular group item rather than a VARCHAR group item. For example, if VARCHAR=YES is specified and you write the following:

```
        01  lastname
            48 lastname-len  PIC S9(4) USAGE COMP.
            48 lastname-text PIC X(15).
```

then, since level 48 instead of 49 is used for the group item elements, the item is interpreted as a regular group item rather than a VARCHAR group item.

For more information about the Pro*COBOL VARCHAR option, see Chapter 7, "Running the Pro*COBOL Precompiler"

Referencing VARCHAR Variables

In SQL statements, you reference a VARCHAR variable using the group name pre-fixed with a colon, as the following example shows:

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        ...
        01  PART-NUMBER  PIC X(5).
        01  PART-DESC    PIC X(20) VARYING.
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...

    EXEC SQL
        SELECT PDESC INTO :PART-DESC FROM PARTS
        WHERE PNUM = :PART-NUMBER
    END-EXEC.
```

After the query executes, PART-DESC-LEN holds the actual length of the character string retrieved from the database and stored in PART-DESC-ARR.

In COBOL statements, you can reference VARCHAR variables using the group name or the elementary items, as this example shows:

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        ...
        01  EMP-TABLES.
            05  EMP-NAME  OCCURS 50 TIMES  PIC X(15) VARYING.
            ...
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    PERFORM DISPLAY-NAME
        VARYING J FROM 1 BY 1 UNTIL J > NAME-COUNT.
```

```
        ...
   DISPLAY-NAME.
       DISPLAY EMP-NAME-ARR OF EMP-NAME(J).
```

Handling Character Data

This section explains how Pro*COBOL handles character host variables. There are two kinds of single-byte character host variables and two kinds of multi-byte NLS character host variables:

- PIC X(*n*) (or PIC X...X)

- PIC X(*n*) VARYING (or PIC X...X VARYING)

- PIC N(*n*) (or PIC N...N) or PIC G(n) (or PIC G...G)

- PIC N(n) VARYING (or PIC N...N VARYING) or PIC G(n) VARYING (or PIC G...G VARYING)

  Attention: Before using multi-byte NCHAR datatypes, verify that the PIC N or PIC G datatype is supported by your COBOL compiler.

New Default for PIC X

Starting in Pro*COBOL 8.0, the default datatype of PIC X variables is changed from VARCHAR2 to CHARF. The new precompiler command line option, PICX, is provided for backward compatibility. PICX can be entered only on the command line or in a configuration file. See "PICX" on page 7-32 for more details.

Effects of the PICX Option

The PICX option determines how Pro*COBOL treats data in character strings. The PICX option allows your program to use ANSI fixed-length strings or to maintain compatibility with previous versions of the Oracle8 Server and Pro*COBOL.

You must use PICX=VARCHAR2 (not the default) to obtain the same results as releases of Pro*COBOL before 8.0. Or, use

```
     EXEC SQL <varname> IS VARCHAR@ END-EXEC
.
```

for each variable.

Fixed-Length Character Variables

Fixed-length character variables are declared using the PIC X(*n*) and PIC G(n) and PIC N(n) datatypes. These types of variables handle character data based on their roles as input or output variables.

On Input

When PICX=VARCHAR2, the program interface strips trailing blanks before sending the value to the database. If you insert into a fixed-length CHAR column, Oracle8 re-appends trailing blanks up to the length of the database column. However, if you insert into a variable-length VARCHAR2 column, Oracle8 never appends blanks.

When PICX=CHARF, trailing blanks are never stripped.

Make sure that the input value is not trailed by extraneous characters. For example, nulls are not stripped and are inserted into the database. Normally, this is not a problem because when a value is ACCEPTed or MOVEd into a PIC X(*n*) variable, COBOL appends blanks up to the length of the variable.

The following example illustrates the point:

```
 WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         01  EMPLOYEES.
             05  EMP-NAME     PIC X(10).
             05  DEPT-NUMBER  PIC S9(4) VALUE 20 COMP.
             05  EMP-NUMBER   PIC S9(9) VALUE 9999 COMP.
             05  JOB-NAME     PIC X(8).
             ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     ...
     DISPLAY "Employee name? " WITH NO ADVANCING.
     ACCEPT EMP-NAME.
*    Assume that the name MILLER was entered
*    EMP-NAME contains "MILLER    " (4 trailing blanks)
     MOVE "SALES" TO JOB-NAME.
*    JOB-NAME now contains "SALES   " (3 trailing blanks)
     EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO, JOB)
         VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER, :JOB-NAME
     END-EXEC.
     ...
```

If you precompile the last example with PICX=VARCHAR2 and the target database columns are VARCHAR2, the program interface strips the trailing blanks on input and inserts just the 6-character string "MILLER" and the 5-character string "SALES" into the database. However, if the target database columns are CHAR, the strings are blank-padded to the width of the columns.

If you precompile the last example with PICX=CHARF and the JOB column is defined as CHAR(10), the value inserted into that column is "SALES#####" (five trailing blanks). However, if the JOB column is defined as VARCHAR2(10), the value inserted is "SALES###" (three trailing blanks), because the host variable is declared as PIC X(8). This might not be what you want, so be careful.

On Output

The PICX option has no effect on output to fixed-length character variables. When you use a PIC X(n) variable as an output host variable, Oracle8 blank-pads it. In our example, when your program fetches the string "MILLER" from the database, EMP-NAME contains the value "MILLER####" (with four trailing blanks). This character string can be used without change as input to another SQL statement.

Restrictions When NLS_LOCAL=YES

Tables Disallowed. Host variables declared using the PIC N or PIC G datatype must not be tables.

No Odd-Byte Widths. Oracle8 CHAR columns should not be used to store multi-byte NCHAR characters. A run-time error is generated if data with an odd number of bytes is FETCHed from a single-byte column into a multi-byte NCHAR host variable.

No Host Variable Equivalencing. Multi-byte NCHAR character variables cannot be equivalenced using an EXEC SQL VAR statement.

No Dynamic SQL. Dynamic SQL is not available for NCHAR multi-byte character string host variables in Pro*COBOL.

Functions should not be used on columns that store multi-byte NLS data.

Variable-Length Variables

VARCHAR variables handle character data based on their roles as input or output variables.

On Input

When you use a VARCHAR variable as an input host variable, your program must assign values to the length and string fields of the expanded VARCHAR declaration, as shown in the following example:

```
IF ENAME-IND = -1
    MOVE "NOT AVAILABLE" TO ENAME-ARR
    MOVE 13 TO ENAME-LEN.
```

You need not blank-pad the string variable. In SQL operations, Oracle8 uses exactly the number of characters given by the length field, counting any spaces.

Host input variables for multi-byte NLS data are *not* stripped of trailing double-byte spaces. The length component is assumed to be the length of the data in characters, not bytes.

On Output

When you use a VARCHAR variable as an output host variable, Oracle8 sets the length field. An example follows:

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  EMPNO  PIC S9(4) COMP.
        01  ENAME  PIC X(15) VARYING.
        ...
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    EXEC SQL
        SELECT ENAME INTO :ENAME FROM EMP
        WHERE EMPNO = :EMPNO
    END-EXEC.
    IF ENAME-LEN = 0
        MOVE FALSE TO VALID-DATA.
```

An advantage of VARCHAR variables over fixed-length strings is that the length of the value returned by Oracle8 is available right away. With fixed-length strings, to get the length of the value, your program must count the number of characters.

Host output variables for multi-byte NCHAR data are *not* padded at all. The length of the buffer is set to the length in characters, not bytes..

Connecting to Oracle

Your Pro*COBOL program must log on to Oracle before querying or manipulating data. To log on, you use the CONNECT statement, as in

```
EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
```

where USERNAME and PASSWD are PIC X(*n*) or PIC X(*n*) VARYING host variables. Alternatively, you can use the statement

```
EXEC SQL
    CONNECT :USR-PWD
END-EXEC.
```

where the host variable USR-PWD contains your username and password separated by a slash (/).

The syntax for the CONNECT statement has an optional ALTER AUTHORIZATION clause. The syntax (Oracle**8** and later) for CONNECT is shown here:

```
EXEC SQL CONNECT { :user IDENTIFIED BY :oldpswd | :usr_psw }
[ [ AT { dbname | :host_variable }] USING :connect_string ]
[ ALTER AUTHORIZATION :newpswd ]
```

(The ALTER AUTHORIZATION clause is explained in "Changing Passwords at Runtime" on page 3-55.)

The CONNECT statement must be the first SQL statement executed by the program. That is, other executable SQL statements can positionally, but not logically, precede the CONNECT statement. If the precompiler option AUTO_CONNECT=YES, a CONNECT statement is not needed.)

To supply the Oracle username and password separately, you define two host variables as character strings or VARCHAR variables. 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. Your program can prompt for the values or you can hard-code them, as follows:

```
WORKING STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME  PIC X(10) VARYING.
        01  PASSWD    PIC X(10) VARYING.
```

```
              ...
        EXEC SQL END DECLARE SECTION END-EXEC.
        ...
 PROCEDURE DIVISION.
 LOGON.
        MOVE "SCOTT" TO USERNAME-ARR.
        MOVE 5 TO USERNAME-LEN.
        MOVE "TIGER" TO PASSWD-ARR.
        MOVE 5 TO PASSWD-LEN.
        EXEC SQL WHENEVER SQLERROR GOTO LOGON-ERROR END-EXEC.
        EXEC SQL
            CONNECT :USERNAME IDENTIFIED BY :PASSWD
        END-EXEC.
```

However, you cannot hard-code a username and password into the CONNECT statement or use quoted literals. For example, the following statements are *invalid*:

```
    EXEC SQL
        CONNECT SCOTT IDENTIFIED BY TIGER
    END-EXEC.

    EXEC SQL
        CONNECT "SCOTT" IDENTIFIED BY "TIGER"
    END-EXEC.
```

See "Sample Tables" on page 2-15

Connecting Using Net8

To connect using a Net8 driver, substitute a service name, as defined in your *tnsnames.ora* configuration file or in Oracle Names, in place of the SQL*Net V1 connect string.

If you are using Oracle Names, the name server obtains the service name from the network definition database.

Note: SQL*Net V1 does work with Oracle8.

See *Oracle Net8 Administrator's Guide* for more information about Net8.

Automatic Logons

You can log on to Oracle automatically with the userid:

```
<prefix><username>
```
where *prefix* is the value of the Oracle initialization parameter OS_AUTHENT_PREFIX (the default value is OPS$) and *username* is your operating system user or task name. For exam-

ple, if the prefix is OPS$, your user name is TBARNES, and OPS$TBARNES is a valid Oracle userid, you log on to Oracle as user OPS$TBARNES.

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

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 ORACLEID   PIC X.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE '/' TO ORACLEID.
EXEC SQL CONNECT :ORACLEID END-EXEC.
```

This automatically connects you as user OPS$*username*. For example, if your operating system username is RHILL, and OPS$RHILL is a valid Oracle username, connecting with a slash (/) automatically logs you on to Oracle as user OPS$RHILL.

You can also pass a character string to Pro*COBOL. However, the string cannot contain trailing blanks. For example, the following CONNECT statement will fail:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      01 ORACLEID   PIC X(5).
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE '/    ' TO ORACLEID.
EXEC SQL CONNECT :ORACLEID END-EXEC.
```

The AUTO_CONNECT Precompiler Option

Pro*COBOL lets your program log on to the default database without using the CONNECT statement. Simply specify the precompiler option AUTO_CONNECT on the command line.

Assume that the default value of OS_AUTHENT_PREFIX is OPS$, your username is TBARNES, and OPS$TBARNES is a valid Oracle userid. When AUTO_CONNECT=YES, as soon as Pro*COBOL encounters an executable SQL statement, your program logs on to Oracle automatically with the userid OPS$TBARNES.

When AUTO_CONNECT=NO (the default), you must use the CONNECT statement to log on to Oracle.

Concurrent Logons

Pro*COBOL supports distributed processing via Net8. Your application can concurrently access any combination of local and remote databases or make multiple connections to the same database. In Figure 3–1, an application program communicates with one local and three remote Oracle**8** databases. ORA2, ORA3, and ORA4 are simply logical names used in CONNECT statements.

Figure 3–1   Connecting via Net8
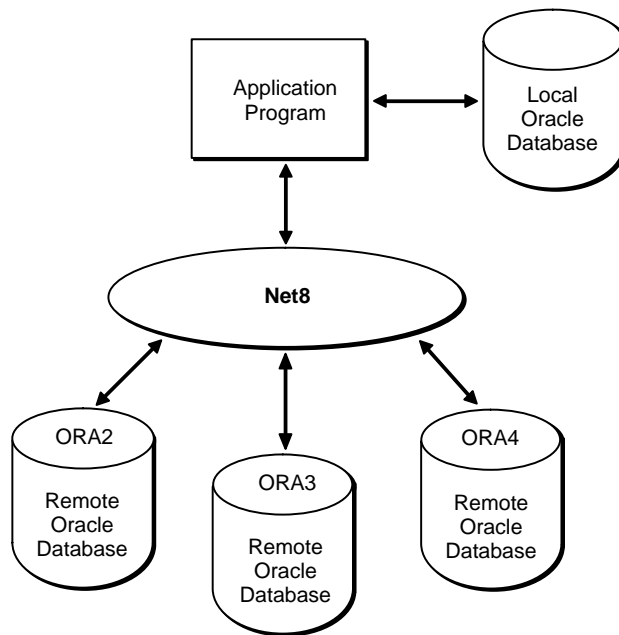
By eliminating the boundaries in a network between different machines and operating systems, Net8 provides a distributed processing environment for Oracle tools. This section shows you how the Pro*COBOL supports distributed processing via Net8. You learn how your application can

- access other databases directly or indirectly

- concurrently access any combination of local and remote databases

- make multiple connections to the same database

Some Preliminaries

The communicating points in a network are called *nodes*. Net8 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.

The Net8 syntax for connecting to the default database in the local domain is simply to use the service name for the database.

If the service name is not in the default (local) domain, you must use a global specification (all domains specified). For example:

```
HR.US.ORACLE.COM
```

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 using a CONNECT statement without an 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. A SQL statement with an AT clause is 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 END-EXEC.
```

Or, you can use:

```
EXEC SQL CONNECT :USR-PWD END-EXEC.
```

where *USR-PWD* contains *USERNAME/PASSWORD*.

You can also log on automatically as shown on "Automatic Logons" on page 3-44.

If you do not specify a database and node, you are connected to the default data-base 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 necessary host variables
 WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         01  USERNAME  PIC X(10) .
         01  PASSWORD  PIC X(10) .
         01  DB-STRING PIC X(20) .
        ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSSWORD.
     MOVE "nyremote" TO DB-STRING.
      ...
* --  Assign a unique name to the database connection.
     EXEC SQL DECLARE DBNAME DATABASE END-EXEC.
* --  Connect to the non-default database
     EXEC SQL
     CONNECT :USERNAME IDENTIFIED BY :PASSWORD
     AT DBNAME USING :DB-STRING
     END-EXEC.
```

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 Net8 syntax for logging on to a non-default database at a remote node using the DECnet protocol.

- The undeclared identifier *DBNAME* names a non-default connection; it is an identi-fier used by Oracle, *not* a host or program variable.

The USING clause specifies the network, machine, and database to be associated with *DBNAME*. Later, SQL statements using the AT clause (with *DBNAME*) 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 necessary host variables
 WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME  PIC X(10).
        01  PASSWORD  PIC X(10).
        01  DB-NAME   PIC X(10).
        01  DB-STRING PIC X(20).
        ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSSWORD.
     MOVE "oracle1" TO DB-NAME.
     MOVE "nyremote" TO DB-STRING.
     ...
* --  Connect to the non-default database
     EXEC SQL
     CONNECT :USERNAME IDENTIFIED BY :PASSWORD
     AT :DB-NAME USING :DB-STRING
     END-EXEC.
```

If *DB-NAME* is a host variable, the DECLARE DATABASE statement is not needed. Only if *DBNAME* is an undeclared identifier must you execute a DECLARE *DBNAME* DATABASE statement before executing a CONNECT ... AT *DBNAME* 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 DBNAME SELECT ...
     EXEC SQL AT DBNAME INSERT ...
     EXEC SQL AT DBNAME 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.

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 END-EXEC.
```

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, you must declare *DB-NAME* globally or pass it to each subprogram.

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 ...
MOVE "oracle1" TO DB-NAME.
EXEC SQL OPEN EMP-CURSOR END-EXEC.
EXEC SQL FETCH EMP-CURSOR INTO ...
MOVE "oracle2" TO DB-NAME.
* -- illegal, cursor still open
EXEC SQL OPEN EMP-CURSOR END-EXEC.
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:

```
* -- close cursor first
EXEC SQL CLOSE EMP-CURSOR END-EXEC.
MOVE "oracle2" TO DB-NAME.
EXEC SQL OPEN EMP-CUROR END-EXEC.
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 :SQL-STMT END-EXEC.
```

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 SQL-STMT STATEMENT END-EXEC.
EXEC SQL PREPARE SQL-STMT FROM :SQL-STRING END-EXEC.
EXEC SQL EXECUTE SQL-STMT END-EXEC.
```

The following example shows Method 3:

```
EXEC SQL AT :DB-NAME DECLARE SQL-STMT STATEMENT END-EXEC.
EXEC SQL PREPARE SQL-STMT FROM :SQL-STRING END-EXEC.
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR SQL-STMT END-EXEC.
EXEC SQL OPEN EMP-CURSOR ...
EXEC SQL FETCH EMP-CURSOR INTO ...
EXEC SQL CLOSE EMP-CURSOR END-EXEC.
```

You need not use the AT clause when connecting to a remote database unless you open two or more connections simultaneously (in which case the AT clause is needed to identify the active connection). To make the default connection to a remote database, use the following syntax:

```
EXEC SQL
CONNECT :USERNAME IDENTIFIED BY :PASSWORD USING :DB-STRING
END-EXEC.
```

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:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      01  USERNAME  PIC X(10) .
      01  PASSWORD  PIC X(10) .
      01  DB-STRING1 PIC X(20) .
```

```
      01  DB-STRING2 PIC X(20) .
 EXEC SQL END DECLARE SECTION END-EXEC.
 ...
 MOVE "scott" TO USERNAME.
 MOVE "tiger" TO PASSWORD.
 MOVE "New-York" TO DB-STRING1.
 MOVE "Boston" TO DB-STRING2.

* --  give each database connection a unique name
 EXEC SQL DECLARE DBNAME1 DATABASE END-EXEC.
 EXEC SQL DECLARE DBNAME2 DATABASE;
* --  connect to the two non-default databases
 EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
 AT DBNAME1 USING :DB-STRING1 END-EXEC.
 EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
 AT DBNAME2 USING :DB-STRING2 END-EXEC.
```

The undeclared identifiers *DBNAME1* and *DBNAME2* 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:

```
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      01  USERNAME  PIC X(10) .
      01  PASSWORD  PIC X(10) .
      01  DB-NAME   PIC X(10) .
      01  DB-STRING PIC X(20) .
 EXEC SQL END DECLARE SECTION END-EXEC.
 ...
 MOVE "scott" TO USERNAME.
 MOVE "tiger" TO PASSWORD.
 PERFORM GETDB 2 TIMES.
 ...
* --  get next database name and Net8 string
 GETDB.
 DISPLAY "Database Name? ".
 ACCEPT DB-NAME.
 DISPLAY "Net8 String? ".
 ACCEPT DB-STRING.
* --  connect to the non-default database
 EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
 AT :DB-NAME USING :DB-STRING
 END-EXEC.
```

```
      ...
```

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

```
      MOVE "scott" TO USERNAME.
      MOVE "tiger" TO PASSWORD.
      MOVE "nyremote" TO DB-STRING.
      PERFORM GETDB 2 TIMES
      ...
 GETDB.
* -- get next database name
      DISPLAY 'Database Name? '.
      ACCEPT DB-NAME.
* -- connect to the non-default database
      EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
          AT :DB-NAME USING :DB-STRING
      END-EXEC.
      ...
```

You must use different database names for the connections, even if they use the same Net8 string.

Implicit Logons

Implicit logons are supported through the Oracle8 distributed database option, which does not require explicit logons. For example, 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 Oracle8 Server, which connects implicitly 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 'nyremote'
      END-EXEC.
```

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
END-EXEC.
```

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 Oracle8 Server logs on to the non-default database via Net8 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 END-EXEC.
```

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
END-EXEC.
```

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 'nyremote'
END-EXEC.
EXEC SQL CREATE DATABASE LINK db_link2
    CONNECT TO username2 IDENTIFIED BY password2
    USING 'chiremote'
END-EXEC.
EXEC SQL CREATE SYNONYM emp FOR emp@db_link1 END-EXEC.
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2 END-EXEC.
```

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
END-EXEC.
```

Oracle8 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*.

Changing Passwords at Runtime

Pro*COBOL now provides client applications with a convenient way to change a user password at runtime through a simple extension to the EXEC SQL CONNECT statement.

The syntax for the CONNECT statement now has an optional ALTER AUTHORIZATION clause. The new syntax for CONNECT is shown here:

```
EXEC SQL CONNECT { :user IDENTIFIED BY :oldpswd | :usr_psw }
    [ [ AT { dbname | :host_variable }] USING :connect_string ]
    [ ALTER AUTHORIZATION :newpswd ]
```

Using the Connect Syntax

This section describes the possible outcomes of different variations of the new CONNECT statement.

Standard CONNECT

If an application issues the following statement

```
EXEC SQL CONNECT ..    /* No ALTER AUTHORIZATION clause */
```

it performs a normal connection attempt. The possible results include the following:

1. The application will connect without issue.

2. The application will connect, but will receive a password warning. The warning indicates that the password has expired but is in a grace period which will allow logins. At this point, the user is encouraged to change the password before the account becomes locked.

3. The application will fail to connect. Possible causes include the following:

The password is incorrect.

The account has expired, and is possibly in a locked state.

Change Password on CONNECT

The following CONNECT statement

```
EXEC SQL CONNECT .. ALTER AUTHORIZATION :newpswd END-EXEC
```

indicates that the application wants to change the account password to the value indicated by newpswd. After the change is made, an attempt is made to connect as user/newpswd. This can have the following results:

1. The application will connect without issue

2. The application will fail to connect. This could be due to either of the following:

   a. Password verification failed for some reason. In this case the password remains unchanged.

   b. The account is locked. Changes to the password are not permitted.

4

Advanced Pro*COBOL Programs

Advanced Pro*COBOL techniques are presented. Topics are:

- The Oracle8 Datatypes
- Datatype Conversion
- Explicit Control Over DATE String Format
- Datatype Equivalencing
- Embedding PL/SQL
- National Language Support
- Multi-Byte NLS Character Sets
- Embedding OCI (Oracle Call Interface) Calls
- Developing X/Open Applications

The Oracle8 Datatypes

Oracle8 recognizes two kinds of datatypes: *internal* and *external.* Internal datatypes specify how Oracle8 stores data in database columns. Oracle8 also uses internal datatypes to represent database pseudo-columns. An external datatype specifies how data is stored in a host variable.

Internal Datatypes

For values stored in database columns, Oracle8 uses the following internal datatypes:

Table 4–1   Internal Datatypes

Name	Code	Description
CHAR	96	<= 2000-byte, fixed-length string
NCHAR	96	<= 2000-byte, fixed-length single-byte or fixed-width multi-byte string
DATE	12	7-byte, fixed-length date/time value
LONG	8	<= 2147483647-byte, variable-length string
LONG RAW	24	<= 2147483647-byte, variable-length binary data
MLSLABEL	105	<= 5-byte, variable-length binary label
NUMBER	2	fixed or floating point number, represented in abinary coded decimal format
RAW	23	<= 255-byte, variable-length binary data
ROWID	11	fixed-length binary value
VARCHAR2	1	<= 4000-byte, variable-length string
NVARCHAR2	1	<= 4000-byte, variable-length single-byte or fixed-width multi-byte string

These internal datatypes can be quite different from COBOL datatypes. For example, COBOL has no equivalent to the NUMBER datatype, which was specially designed for portability and high precision.

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 2000 bytes. The syntax follows:

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

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 multi-byte (2-byte) characters, its maximum width is less than *n*/2 characters.

NCHAR

Use this datatype to store NLS (National Language Support) strings. See "National Language Support" on page 4-30. NCHAR values can not be converted to an internal datatype and are only used in the Declare Table when performing a semantics check with SQLCHECK=SEMANTICS (or FULL). See "Specifying SQLCHECK=SEMANTICS" on page E-3 for a discussion of semantics checking. See "DECLARE TABLE (Oracle Embedded SQL Directive)" on page F-20 for a discussion and syntax diagram of this embedded SQL directive. You can not insert CHAR values into an NCHAR column. You can not insert NCHAR values into a CHAR column. This datatype can not be used in VAR statements for datatype equivalences.

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.

Internally, DATEs are stored in a binary format. When converting a DATE column value to a character string in your program, Oracle8 uses the default format mask for your session. If you need other date/time information such as the date in Julian days, use the TO_CHAR function with a format mask. Always convert DATE column values to and from character strings using (external) character datatypes such as VARCHAR2 or STRING.

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 the maximum width of a LONG column is 2147483647 bytes or two gigabytes.

You can use LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, function calls, or 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.

LONG RAW

You use the LONG RAW datatype to store variable-length 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 Oracle8 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.

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 Oracle documentation.

You can use the MLSLABEL datatype to define a database column. However, with standard Oracle8, 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.

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, Oracle8 does extra integrity checks before storing the data. If a value exceeds the precision, Oracle8 issues an error message; if a value exceeds the scale, Oracle8 rounds the value.

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 Oracle8.

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 Oracle8 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.

ROWID

Internally, every table in an Oracle8 database has a pseudo-column named ROWID, which stores binary values called *rowids*. ROWIDs uniquely identify rows and provide the fastest way to access particular rows.

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 4000 bytes. To define a VARCHAR2 column, you use the syntax

```
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 multi-byte (2-byte) characters, its maximum width is less than $n/2$ characters.

NVARCHAR2

Use NVARCHAR2 to store variable-length NLS character data. For fixed-width character sets, specify the maximum length in characters. For variable-width character sets, specify the maximum length in bytes. See "National Language Support" on page 4-30. NVARCHAR2 values can not be converted to an internal datatype and are only used in the Declare Table when performing a semantics check with SQLCHECK=SEMANTICS (or FULL). See "Specifying SQLCHECK=SEMANTICS" on page E-3 for a discussion of semantics checking. See "DECLARE TABLE (Oracle Embedded SQL Directive)" on page F-20 for a discussion and syntax diagram of this embedded SQL directive. You can not insert VARCHAR2 values into an NVARCHAR2 column. You can not insert NVARCHAR2 values into a VARCHAR2 column.This datatype can not be used in VAR statements for datatype equivalences.

SQL Pseudo-columns and Functions

SQL recognizes the pseudo-columns in Table 4–2, which return specific data items:

Table 4–2   Pseudocolumns and Internal Datatypes

Pseudo-column	Internal Datatype
CURRVAL	NUMBER
LEVEL	NUMBER
NEXTVAL	NUMBER
ROWID	ROWID
ROWLABEL	MLSLABEL
ROWNUM	NUMBER

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 pseudo-column values from a dummy table.

In addition, SQL recognizes the parameterless functions in Table 4–3, which also return specific data items:

Table 4–3   Functions and Internal Datatypes

Function	Internal Datatype
SYSDATE	DATE
UID	NUMBER
USER	VARCHAR2

You can refer to 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
END EXEC.
```

Brief descriptions of the SQL pseudo-columns and functions follow. For details, see the *Oracle8 Server SQL Reference.*

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

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.

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
END EXEC.
```

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.

ROWID returns a row address in hexadecimal.

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 END-EXEC.
```

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 END-EXEC.
```

SYSDATE returns the current date and time.

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

USER returns the username of the current Oracle user.

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'
END-EXEC.
```

For more information about the ROWLABEL column, see the Trusted Oracle documentation.

External Datatypes

As the table below shows, the external datatypes include all the internal datatypes plus several datatypes found in other supported host languages. For example, the STRING external datatype refers to a C null-terminated string. You use the datatype names in datatype equivalencing, and you use the datatype codes in dynamic SQL Method 4.

Table 4–4   External Datatypes

Name	Code	Description
CHAR	1	<= 65535-byte, variable-length character string (1)
	96	<= 65535-byte, fixed-length character string (1)
CHARF	96	<= 65535-byte, fixed-length character string
CHARZ	97	<= 65535-byte, fixed-length, null-terminated string (2)
DATE	12	7-byte, fixed-length date/time value
DECIMAL	7	COBOL packed decimal
DISPLAY	91	COBOL numeric character string
DISPLAY TRAILING	152	COBOL numeric with trailing sign
FLOAT	4	4-byte or 8-byte floating-point number
INTEGER	3	2-byte or 4-byte signed integer
LONG	8	<= 2147483647-byte, fixed-length string
LONG RAW	24	<= 217483647-byte, fixed-length binary data
LONG VAR-CHAR	94	<= 217483643-byte, variable-length string
LONG VARRAW	95	<= 217483643-byte, variable-length binary data
MLSLABEL	106	2..5-byte, variable-length binary data
NUMBER	2	integer or floating-point number
OVER-PUNCH LEADING	172	numeric with embedded leading sign

Table 4–4   External Datatypes

Name	Code	Description
OVER-PUNCH TRAILING	154	numeric with embedded trailing sign
RAW	23	<= 65535-byte, fixed-length binary data (2)
ROWID	11	fixed-length binary value (system-specific)
STRING	5	<= 65535-byte, null-terminated character string (2)
UNSIGNED	68	2-byte or 4-byte unsigned integer
UNSIGNED DISPLAY	153	COBOL unsigned numeric
VARCHAR	9	<= 65533-byte, variable-length character string
VARCHAR2	1	<= 65535-byte, variable-length character string (2)
VARNUM	6	variable-length binary number
VARRAW	15	<= 65533-byte, variable-length binary data

Notes:

1. CHAR is datatype 1 when PICX=VARCHAR2 and datatype 96 when PICX=CHARF.

2. Maximum size is 32767 (32K) on some platforms.

CHAR

CHAR behavior depends on the settings of the option PICX. See "PICX" on page 7-32.

CHARF

By default, Oracle8 assigns the CHARF datatype to all non-varying character host variables. You use the CHARF datatype to store fixed-length character strings. On most platforms, the maximum length of a CHARF value is 65535 (64K) bytes. See "PICX" on page 7-32.

On Input. Oracle8 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, Oracle8 generates an error. If the input value is all-blank, Oracle8 treats it like a character value.

On Output. Oracle8 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, Oracle8 fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, Oracle8 truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle8 sets it to the original length of the output value.

CHARZ

Use the CHARZ datatype to store fixed-length, null-terminated character strings. On most platforms, the maximum length of a CHARZ value is 65,535 bytes. You should not need this external type in Pro*COBOL.

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. Oracle8 appends a null terminator to the output value, which is also blank-padded if necessary.

DATE

Use the DATE datatype to store dates and times in 7-byte, fixed-length fields. As Table 4–5 shows, the century, year, month, day, hour (in 24-hour format), minute, and second are stored in that order from left to right.

Table 4–5   DATE Format

Byte	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example 17-OCT-1994 at 1:23:12 PM	119	194	10	17	14	24	13

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 is January 1, 4712 B.C.E. 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).

DECIMAL

With Pro*COBOL, 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. If significant digits are lost during data conversion, Oracle8 fills the host variable with asterisks.

DISPLAY

With Pro*COBOL, use the DISPLAY datatype to store numeric character data. The DISPLAY datatype refers to a COBOL "DISPLAY SIGN LEADING SEPARATE" number, which requires $n + 1$ bytes of storage for PIC S9(n), and $n + d + 1$ bytes of storage for PIC S9(n)V9(d).

FLOAT

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.

Oracle8 can represent numbers with greater precision than floating point implementations because the internal format of Oracle8 numbers is decimal.

> **Note:** In SQL statements, when comparing FLOAT values, use the SQL function ROUND because FLOAT stores binary (not decimal) numbers; so, fractions do not convert exactly.

INTEGER

Use the INTEGER datatype to store numbers that have no fractional part. An integer is a signed, 2- 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, Oracle8 truncates the fractional part.

LONG

Use the LONG datatype to store fixed-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 2147483647 bytes (two gigabytes).

LONG RAW

Use the LONG RAW datatype to store fixed-length, binary data or byte strings. The maximum length of a LONG RAW value is 2147483647 bytes (two gigabytes).

LONG RAW data is like LONG data, except that Oracle8 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.

LONG VARCHAR

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 bytes. In an EXEC SQL VAR statement, do *not* include the 4-byte length field.

LONG VARRAW

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. In an EXEC SQL VAR statement, do *not* include the 4-byte length field.

MLSLABEL

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 your Trusted Oracle documentation, 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 the label is invalid, 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.

NUMBER

Use the NUMBER datatype to store fixed or floating point Oracle8 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 in variable-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 Oracle8. 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. Oracle8 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.

RAW

Use the RAW datatype to store fixed-length binary data or byte strings. On most platforms, the maximum length of a RAW value is 65535 bytes.

RAW data is like CHAR data, except that Oracle8 assumes nothing about the meaning of RAW data and does no character set conversions when you transmit RAW data from one system to another.

ROWID

Use the ROWID datatype to store binary rowids in 18-byte fixed-length fields. The field size is system-specific. So, check your system-specific Oracle8 manuals.

The ROWID in Oracle 8 has a format of 'OOOOOOFFFBBBBBBSSS' which is an

 18 character string where:

OOOOOO = is a base 64 encoding of the 32-bit data object number.

 (Data object number was introduced in 8.0 to track versions of the same segment because certain operations can change the version. It is used to discover stale ROWIDs and stale undo records)

FFF = is a base 64 encoding of the relative file number

BBBBBB = is a base 64 encoding of the block number

SSS = is a base 64 encoding of the slot (row) number

This format is called the extended ROWID character format.

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, Oracle8 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 "Mimicking the CURRENT OF Clause" on page 10-14.

> **Note:** If you need full portability or your application communicates with a non-Oracle database via Transparent Gateway, specify a maximum length of 256 (not 18) bytes when declaring the VARCHAR2 host variable. If your application communicates with a non-Oracle data source 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.

STRING

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

On Input. Oracle8 uses the specified length to limit the scan for a null terminator. If a null terminator is not found, Oracle8 generates an error. If you do not specify a length, Oracle8 assumes the maximum length, which is 65535 on most platforms.

The minimum length of a STRING value is 2 bytes. If the first character is a null terminator and the specified length is 2, Oracle8 inserts a null unless the column is defined as NOT NULL. An all-blank or null-terminated value is stored intact.

On Output. Oracle8 appends a null byte to the last character returned. If the string length exceeds the specified length, Oracle8 truncates the output value and appends a null byte.

UNSIGNED

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, Oracle8 truncates the fractional part.

VARCHAR

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. In an EXEC SQL VAR statement, do *not* include the 2-byte length field.

VARCHAR2

Use the VARCHAR2 datatype to store variable-length character strings. On most platforms, the maximum length of a VARCHAR2 value is 65535 bytes.

Specify the maximum length of a VARCHAR2(n) value in bytes, not characters. So, if a VARCHAR2(n) variable stores multi-byte characters, its maximum length is less than n characters.

On Input. Oracle8 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 un-initialized host variable can contain nulls. So, always blank-pad a character input host variable to its declared length. (COBOL PIC X(n) variables do this automatically.)

If the input value is longer than the defined width of the database column, Oracle8 generates an error. If the input value is all-blank, Oracle8 treats it like a null.

Oracle8 can convert a character value to a NUMBER column value if the character value represents a valid number. Otherwise, Oracle8 generates an error.

On Output. Oracle8 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, Oracle8 fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, Oracle8 truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle8 sets it to the original length of the output value.

Oracle8 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, scien-

tific notation is used. For example, if you select the column value 123456789 into a host variable of length 6, Oracle8 returns the value "1.2E08" to the host variable.

VARNUM

The VARNUM datatype is like the NUMBER datatype, except that the first byte of a VAR-NUM 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 Oracle8. 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.

VARRAW

Use the VARRAW datatype to store variable-length 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. In an EXEC SQL VAR statement, do *not* include the 2-byte length field. To get the length of a VARRAW variable, simply refer to its length field.

Datatype Conversion

At precompile time, an external datatype is assigned to each host variable. For example, Pro*COBOL assigns the INTEGER external datatype to host variables of type PIC S9(*n*) COMP. At run time, the datatype code of every host variable used in a SQL statement is passed to Oracle8. Oracle8 uses the codes to convert between internal and external datatypes.

Before assigning a SELECTed column value to an output host variable, Oracle8 must convert 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 column, Oracle8 must convert the external datatype of the host variable to the internal datatype of the target column.

Conversions between internal and external datatypes follow the usual data conversion rules. For example, you can convert a CHAR value of "1234" to a PIC S9(4) COMP value. You cannot, however, convert a CHAR value of "65543" (number too large) or "10F" (number not decimal) to a PIC S9(4) COMP value. Likewise, you cannot convert a PIC X(*n*) value that contains alphabetic characters to a NUMBER value.

The datatype of the host variable must be compatible with that of the 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 National Language Support (NLS) parameters in the Oracle8 initialization file. For example, your system might be configured to recognize a comma (,) instead of a period (.) as the decimal character. For more information about NLS, see the *Oracle8 Application Developer's Guide*.

The following table shows the supported conversions between internal and external datatypes.

Table 4–6   Conversions Between Internal and External Datatypes

	Internal								
External	CHAR	DATE	LONG	LONG RAW	MLSLABEL	NUMBER	RAW	ROWID	VARCHAR2
CHAR	I/O	I/O (2)	I/O	I(3)	I/O (7)	I/O	I/O (3)	I/O (1)	I/O
CHARF	I/O	I/O (2)	I/O	I (3)	I/O (7)	I/O	I/O (3)	I/O (1)	I/O
CHARZ	I/O	I/O (2)	I/O	I (3)	I/O(7)	I/O	I/O (3)	I/O (1)	I/O
DATE	I/O	I/O	I						I/O
DECIMAL	I/O (4)		I			I/O			I/O (4)
DISPLAY	I/O (4))		I			I/O			I/O (4)
FLOAT	I/O (4)		I			I/O			I/O (4)
INTEGER	I/O (4)		I			I/O			I/O (4)
LONG	I/O	I/O (2)	I/O	I (3.5)	I/O (7)	I/O	I/O (3)	I/O (1)	I/O
LONG RAW	O(6)		I (5,6)	I/O			I/O		O (6)
LONG VAR-CHAR	I/O	I/O(2)	I/O	I (3,5)	I/O (7)	I/O	I/O(3))	I/O (1)	I/O
LONG VARRAW	I/O (6)		I (5,6)	I/O			I/O		I/O (6)
MLSLABEL	I/O (8)		I/O (8)		I/O				I/O (8)
NUMBER	I/O (4)		I			I/O			I/O (4)
RAW	I/O (6)		I (5,6)	I/O			I/O		I/O (6)
ROWID	I		I					I/O	I

Table 4–6   Conversions Between Internal and External Datatypes

STRING	I/O	I/O (2)	I/O	I (3.5)	I/O (7)	I/O	I/O (3)	I/O (1)	I/O
UNSIGNED	I/O (4)		I			I/O			I/O (4)
VARCHAR	I/O	I/O (2)	I/O	I (3,5)	I/O (7)	I/O	II/O (3)		I/O
VARCHAR2	I/O	I/O (2)	I/O	I (3)	I/O (7)	I/O	I/O (3)	I/O (1)	I/O
VARNUM	I/O (4)		I			I/O			I/O (4)
VARRAW	I/O (6)		I (5,6)	I/O			I/O		I/O (6)

Notes:

1. On input, host string must be in Oracle'BBBBBBBB.RRRR.FFFF' format.

On output, column value is returned in same format.

2. On input, host string must be the default DATE character format.

On output, column value is returned in same format

3. On input, host string must be in hex format.

On output, column value is returned in same format.

4. On output, column value must represent a valid number.

5. On input, length must be less than or equal to 2000.

6. On input, column value is stored in hex format.

On output, column value must be in hex format.

Legend:

7. On input, host string must be a valid OS label in text format.

I = input only

On output, column value is returned in same format.

O = output only

8. On input, host string must be a valid OS label in raw format.

I/O = input or output

On output, column value is returned in same format.

Explicit Control Over DATE String Format

When you select a DATE column value into a character host variable, Oracle8 must convert the internal binary value to an external character value. So, Oracle8 implicitly calls the SQL function TO_CHAR, which returns a character string in the default date format. The default is set by the Oracle8 initialization parameter NLS_DATE_FORMAT. To get other information such as the time or Julian date, you must explicitly call TO_CHAR with a format mask.

A conversion is also necessary when you insert a character host value into a DATE column. Oracle8 implicitly calls the SQL function TO_DATE, which expects the default date format. To insert dates in other formats, you must explicitly call TO_DATE with a format mask.

For compatibility with other versions of SQL Pro*COBOL now provides the following pre-compiler option to specify date strings:

DATE_FORMAT={ISO | USA | EUR | JIS | LOCAL | '*fmt*' (default LOCAL)}

The DATE_FORMAT option must be used on the command line or in a configuration file. The date strings are shown in the following table:

Table 4–7   Formats for Date Strings

Format Name	Abbreviation	Date Format
International Standards Organization	ISO	yyyy-mm-dd
USA standard	USA	mm/dd/yyyy
European standard	EUR	dd.mm.yyyy
Japanese Industrial Standard	JIS	yyyy-mm-dd
installation-defined	LOCAL	Any installation-defined form.

'*fmt*' is a date format model, such as 'Month dd, yyyy'. See the Oracle8 SQL Reference Manual for the list of date format model elements.

Note: All separately compiled units to be linked together must use the same DATE_FORMAT value.

Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle8 interprets input data and the way Oracle8 formats output data. You can equivalence supported COBOL datatypes to Oracle8 external datatypes on a variable-by-variable basis.

Why Equivalence Datatypes?

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

You can use datatype equivalencing when you want Oracle8 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. Unless NLS parameters in the Oracle8 initialization file specify otherwise, if you select a DATE col-

umn value into a character host variable, Oracle8 returns a 9-byte string formatted as follows:

```
DD-MON-YY
```

However, if you equivalence the character host variable to the DATE external datatype, Oracle8 returns a 7-byte value in the internal format.

Host Variable Equivalencing

By default, Pro*COBOL assigns a specific external datatype to every host variable. You can override the default assignments by equivalencing host variables to Oracle8 external datatypes. This is called *host variable equivalencing*.

The syntax of the VAR embedded SQL statement is:

```
EXEC SQL
    VAR <host_variable> IS <datatype> [CONVBUFSZ [IS] (<size>)]
END-EXEC
```

or

```
EXEC SQL VAR <host_variable> [CONVBUFSZ [IS] (<size>)] END-EXEC
```
where <datatype> is:

```
<SQL datatype> [ ( {<length> | <precision>, <scale> } ) ]
```

There must be at least one of the two clauses, or both.

where:

host_variable	is an input or output host variable (or host table) declared earlier.
	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 LONG VARCHAR or LONG VARRAW, *host_variable* must be at least 5 bytes long.
SQL datatype	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 predefined. 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. Pro*COBOL accounts for the length field. If *type_name* is LONG VARCHAR or LONG VARRAW and the data field exceeds 65533 bytes, put "-1" in the *length* field.
precision and scale	are integer literals that represent, respectively, the number of significant digits and the point at which rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of -3 means the number is rounded to the nearest thousand (3456 becomes 3000).
	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*.
	Specify *precision* and *scale* only when *type_name* is DECIMAL or DISPLAY.
size	an integer which is the size, in bytes, of a buffer used to perform conversion of the specified host_variable to another character set.

Table 4–8 shows which parameters to use with each external datatype.

The CONVBUFSZ clause is explained in "CONVBUFSZ Clause in VAR Statement" on page 4-23.

You cannot use EXEC SQL VAR with NCHAR host variables (those containing PIC G or PIC N clauses).

If DECLARE_SECTION=TRUE then you must have a Declare Section and you must place EXEC SQL VAR statements in the Declare Section.

For a syntax diagram of this statement, see "VAR (Oracle Embedded SQL Directive)" on page F-53.

When *ext_type_name* is FLOAT, use *length*; when *ext_type_name* is DECIMAL, you must specify *precision* and *scale* instead of *length*.

Host variable equivalencing is useful in several ways. For example, you can use it when you want Oracle8 to store but not interpret data. Suppose you want to store a host table of

4-byte integers in a RAW database column. Simply equivalence the host table to the RAW external datatype, as follows:

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         01  EMP-TABLES.
             05  EMP-NUMBER  PIC S9(4) COMP OCCURS 50 TIMES.
             ...
*    Reset default datatype (INTEGER) to RAW.
     EXEC SQL VAR EMP-NUMBER IS RAW (200) END-EXEC.
     EXEC SQL END DECLARE SECTION END-EXEC.
```

With host tables, the length you specify must match the buffer size required to hold the table. In the last example, you specified a length of 200, which is the buffer size needed to hold 50 4-byte integers.

You can also declare a group item to be used as a LONG VARCHAR:

```
         01 MY-LONG-VARCHAR.
             05 UC-LEN PIC S9(9) COMP.
             05 UC-ARR PIC X(6000).
     EXEC SQL VAR MY-LONG-VARCHAR IS LONG VARCHAR(6000).
```

CONVBUFSZ Clause in VAR Statement

The EXEC SQL VAR statement can have an optional *CONVBUFSZ* clause. You specify the size, in bytes, of the buffer in the Oracle8 runtime library used to perform conversion of the specified host variable between character sets.

When you have not used the CONVBUFSZ clause, the Oracle8 runtime automatically determines a buffer size based on the ratio of the host variable character size (determined by NLS_LANG) and the character size of the database character set. This can sometimes result in the creation of a buffer of LONG size. Databases are allowed to have only one LONG column. An error is raised if there is more than one LONG value.

To avoid such errors, you use a length shorter than the size of a LONG. If a character set conversion results in a value longer than the length specified by CONVBUFSZ, then Pro*COBOL returns an error.

An Example

Suppose you want to select employee names from the EMP table, then pass them to a C-language 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 END-EXEC.
    ...
        01   EMP-NAME  PIC X(11).
     EXEC SQL VAR EMP-NAME IS STRING (11) END-EXEC.
     EXEC SQL END DECLARE SECTION END-EXEC.
```

The width of the ENAME column is 10 characters, so you allocate 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*, Oracle8 null-terminates the value for you.

Table 4–8   Parameters for Host Variable Equivalencing

External Datatype	Length	Precision	Scale	Default Length
CHAR	optional	n/a	n/a	declared length of variable
CHARZ	optional	n/a	n/a	declared length of variable
DATE	n/a	n/a	n/a	7 bytes
DECIMAL	n/a	required	required	none
DISPLAY	n/a	required	required	none
DISPLAY TRAIL-ING	n/a	required	required	none
UNSIGNED DIS-PLAY	n/a	required	required	none
OVERPUNCH TRAILING	n/a	required	required	none
OVERPUNCH LEADING	n/a	required	required	none
FLOAT	optional (4 or 8)	n/a	n/a	declared length of variable
INTEGER	optional (1, 2, or 4)	n/a	n/a	declared length of variable
LONG	optional	n/a	n/a	declared length of variable

Table 4–8   Parameters for Host Variable Equivalencing

External Datatype	Length	Precision	Scale	Default Length
LONG RAW	optional	n/a	n/a	declared length of variable
LONG VAR-CHAR	required (note 1)	n/a	n/a	none
LONG VARRAW	required (note 1)	n/a	n/a	none
MLSLABEL	required	n/a	n/a	none
NUMBER	n/a	n/a	n/a	not available
STRING	optional	n/a	n/a	declared length of variable
RAW	optional	n/a	n/a	declared length of variable
ROWID	n/a	n/a	n/a	18 bytes (see note 2)
UNSIGNED	optional (1, 2, or 4)	n/a	n/a	declared length of variable
VARCHAR	required	n/a	n/a	none
VARCHAR2	optional	n/a	n/a	declared length of variable
VARNUM	n/a	n/a	n/a	22 bytes
VARRAW	optional	n/a	n/a	none

1. If the data field exceeds 65,533 bytes, pass -1.

2. This length is typical but the default is port-specific.

Using the CHARF Datatype Specifier

You can use the datatype specifier CHARF in VAR statements to equivalence COBOL datatypes to the fixed-length ANSI datatype CHAR.

When PICX=CHARF, specifying the datatype CHAR in a VAR statement equivalences the host-language datatype to the fixed-length ANSI datatype CHAR (Oracle8 external datatype code 96). However, when PICX=VARCHAR2, the host-language datatype is equivalenced to the variable-length datatype VARCHAR2 (code 1).

However, you can always equivalence host-language datatypes to the fixed-length ANSI datatype CHAR. Simply specify the datatype CHARF in the VAR statement. If you use CHARF, the host-language datatype is equivalenced to the fixed-length ANSI datatype CHAR even when PICX=VARCHAR2.

Guidelines

To input VARNUM or DATE values, you must use the Oracle8 internal format. Keep in mind that Oracle8 uses the internal format to output VARNUM and DATE values.

After selecting a column value into a VARNUM host variable, you can check the first byte to get the length of the value. Table 4–9 gives some examples of returned VARNUM values.

Table 4–9   VARNUM Examples

	VARNUM Value			
Decimal Value	Length Byte	Exponent Byte	Mantissa Bytes	Terminator Byte
0	1	128	n/a	n/a
5	2	193	6	n/a
-5	3	62	96	102
2767	3	194	28, 68	n/a
-2767	4	61	74, 34	102
100000	2	195	11	n/a
1234567	5	196	2, 24, 46, 68	n/a

For converting DATE values, see "Explicit Control Over DATE String Format" on page 4-19.

If no Oracle8 external datatype suits your needs exactly, use a VARCHAR2-based or RAW-based external datatype.

RAW and LONG RAW Values

When you select a RAW or LONG RAW column value into a character host variable, Oracle8 must convert the internal binary value to an external character value. In this case, Oracle8 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,

Oracle8 returns the binary byte 11111111 as the pair of characters "FF". The SQL function RAWTOHEX performs 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, Oracle8 issues the following error message:

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

For more information about datatype conversion, see "Sample Program 4: Datatype Equivalencing" on page 5-19.

The default assignments of External and COBOL datatypes are shown in Table 4–10

Table 4–10   Host Variable Equivalencing

COBOL Datatype	External Datatype	Code
PIC X...X PIC X(*n*)	CHARF	96
PIC X...X VARYING PIC X(*n*) VARYING	VARCHAR	9
PIC S9...9 COMP PIC S9(*n*) COMP PIC S9...9 COMP-5 PIC S9(*n*) COMP-5 PIC S9...9 COMP-4 PIC S9(*n*) COMP-4 PIC S9...9 BINARY PIC S9(*n*) BINARY	INTEGER	3
COMP-1 COMP-2	FLOAT	4
PIC S9...9V9...9 COMP-3 PIC S9(*n*)V9(*n*) COMP-3 PIC S9...9V9...9 PACKED-DECIMAL PIC S9(*n*)V9(*n*) PACKED-DECIMAL	DECIMAL	7
PIC 9(n) COMP PIC 9...9 COMP	UNSIGNED	68
PIC S9...9V9...9 LEADING SEPARATE PIC S9(*n*)V9(*n*) LEADING SEPARATE	DISPLAY	91
PIC 9(n)V9(9) PIC 9...9V9...9	UNSIGNED DIS-PLAY	153
PIC S9...9V9...9 TRAILING PIC S9(n)V9(n) TRAILING	OVERPUNCH TRAILING	154

Table 4–10   Host Variable Equivalencing

COBOL Datatype	External Datatype	Code
PIC S9...9V9...9 LEADING PIC S9(n)V9(n) LEADING	OVERPUNCH LEADING	172
PIC S9...9V9...9 TRAILING SEPARATE PIC S9(n)V9(n) TRAILING SEPARATE	DISPLAY TRAIL-ING	152

Embedding PL/SQL

Pro*COBOL treats 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, declare the variables to be shared with PL/SQL and bracket the PL/SQL block with the EXEC SQL EXECUTE and END-EXEC keywords.

Host Variables

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

VARCHAR Variables

When entering a PL/SQL block, Oracle8 automatically checks the length fields of VAR-CHAR host variables, so you must set the length fields *before* the block is entered. For input variables, set the length field to the length of the value stored in the string field. For output variables, set the length field to the maximum length allowed by the string field.

Multi-Byte NCHAR Features When NLS_LOCAL=YES

When NLS_LOCAL=YES, multi-byte NCHAR features are not supported within a PL/SQL block. These features include N-quoted character literals (see Chapter 4, "Advanced Pro*COBOL Programs") and fixed-length character variables.

Indicator Variables

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.

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.

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.

SQLCHECK

You must specify SQLCHECK=SEMANTICS when precompiling a program with an embedded PL/SQL block. You must also use the USERID option. For more information, see Chapter 7, "Running the Pro*COBOL Precompiler".

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 or more, to represent each character. How does Oracle8 deal with such dissimilar languages?

Oracle8 provides National Language Support (NLS), which lets you process single-byte and multi-byte 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 Oracle8 in their native languages.

You control the operation of language-dependent features by specifying various NLS parameters. You can set default parameter values in the Oracle8 initialization file. Table 4–11 shows what each NLS parameter specifies.

Table 4–11   NLS Parameters

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_TERRITORY 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>
```

Pro*COBOL fully supports all the NLS features that allow your applications to process multilingual data stored in an Oracle8 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 *Oracle8 Application Developer's Guide.*

Multi-Byte NLS Character Sets

Pro*COBOL extends support for multi-byte NLS character sets through

- recognition of multi-byte character strings by Pro*COBOL in embedded SQL statements.

- the COBOL PIC N and PIC G datatype declaration clauses, that instruct Pro*COBOL to interpret host character variables as strings of multi-byte characters.

Character Strings in Embedded SQL

A multi-byte NLS character string in an embedded SQL statement consists of the letter *N*, followed by the string enclosed in single quotes.

For example,

```
EXEC SQL
    SELECT EMPNO INTO :EMP-NUM FROM EMP
```

```
        WHERE ENAME=N'<NLS_string>'
    END-EXEC.
```

Embedded DDL

When the precompiler option, NLS_LOCAL=YES, columns storing NCHAR data cannot be used in embedded data definition language (DDL) statements. This restriction cannot be enforced when precompiling, so the use of extended column types, such as NCHAR, within embedded DDL statements results in an execution error rather than a precompile error.

For more information about these options, see their entries in Chapter 7, "Running the Pro*COBOL Precompiler".

Blank Padding

When a Pro*COBOL character variable is defined as a multi-byte NLS variable, the following blank padding and blank stripping rules apply, depending on the external datatype of the variable. See the section "Handling Character Data" on page 3-39.

CHARF. Input data is stripped of any trailing double-byte spaces. However, if a string consists only of multi-byte spaces, a single multi-byte space is left in the buffer to act as a sentinel.

Output host variables are blank padded with multi-byte spaces.

VARCHAR. On input, host variables are *not* stripped of trailing double-byte spaces. The length component is assumed to be the length of the data in characters, not bytes.

On output, the host variable is not blank padded at all. The length of the buffer is set to the length of the data in characters, not bytes.

STRING/LONG VARCHAR. These host variables are not supported for NLS data, since they can only be specified using dynamic SQL or datatype equivalencing, neither of which is supported for NLS data.

Indicator Variables

You can use indicator variables with multi-byte NLS character variables as use you would with any other variable, except column length values are expressed in characters instead of bytes. For a list of possible values, see "Using Indicator Variables" on page 5-3.

Embedding OCI (Oracle Call Interface) Calls

Pro*COBOL let s you embed OCI calls in your program. Just take the following steps:

1. Declare the OCI Logan Data Area (LDA) outside the Declare Section, if it exists. For details, see the *Oracle Call Interface Programmer's Guide*.

2. Connect to Oracle using the embedded SQL statement CONNECT, *not* the OCI call OLOG.

3. Call the Oracle8 run-time library routine SQLLDA to store the connect information in the LDA.

That way, Pro*COBOL and the OCI "know" that they are working together. However, there is no sharing of Oracle8 cursors.

You need not worry about declaring the OCI Host Data Area (HDA) because the Oracle8 run-time library manages connections and maintains the HDA for you.

Setting Up the LDA

You set up the LDA by issuing the OCI call

```
CALL "SQLLDA" USING LDA.
```
where *LDA* identifies the LDA data structure. See the *Oracle Call Interface Programmer's Guide*. 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 set up 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 END-EXEC.
        01  USERNAME   PIC X(10) .
        01  PASSWORD   PIC X(10) .
        01  DB-STRING1 PIC X(20) .
        01  DB-STRING2 PIC X(20) .
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
* -- Field sizes in LDA are system-dependent.
        01 LDA1.
            02 LDA1-V2RC  PIC S9(4) COMP.
            02 FILLER     PIC X(10).
            02 LDA1-RC    PIC S9(4) COMP.
```

```
                02 FILLER      PIC X(50).
           01 LDA2.
                02 LDA2-V2RC   PIC S9(4) COMP.
                02 FILLER      PIC X(10).
                02 LDA2-RC     PIC S9(4) COMP.
                02 FILLER      PIC X(50).
     ...
     MOVE 'SCOTT' TO USERNAME.
     MOVE 'TIGER' TO PASSWORD.
     MOVE 'D:NEWYORK-NONDEF1' TO DB-STRING1.
     MOVE 'D:CHICAGO-NONDEF2' TO DB-STRING2.
     ...
* --  give each database connection a unique name
     EXEC SQL DECLARE db_name1 DATABASE END-EXEC.
     EXEC SQL DECLARE db_name2 DATABASE END-EXEC.
     ...
* -- connect to first non-default database
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
         AT db_name1 USING :DB-STRING1
     END-EXEC.
* --  set up first LDA for OCI use
     CALL 'SQLLDA' USING LDA1.
* -- connect to second non-default database
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
         AT db_name2 USING :DB-STRING2
     END-EXEC.
* --  set up second LDA for OCI use
     CALL 'SQLLDA' USING LDA2.
```

Remember, do not declare *db_name1* and *db_name2* 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 4–1   shows one way that components of the DTP model can interact to provide efficient access to data in an Oracle8 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 interface* between your application program and the resource managers.

Figure 4–1   Hypothetical DTP Model
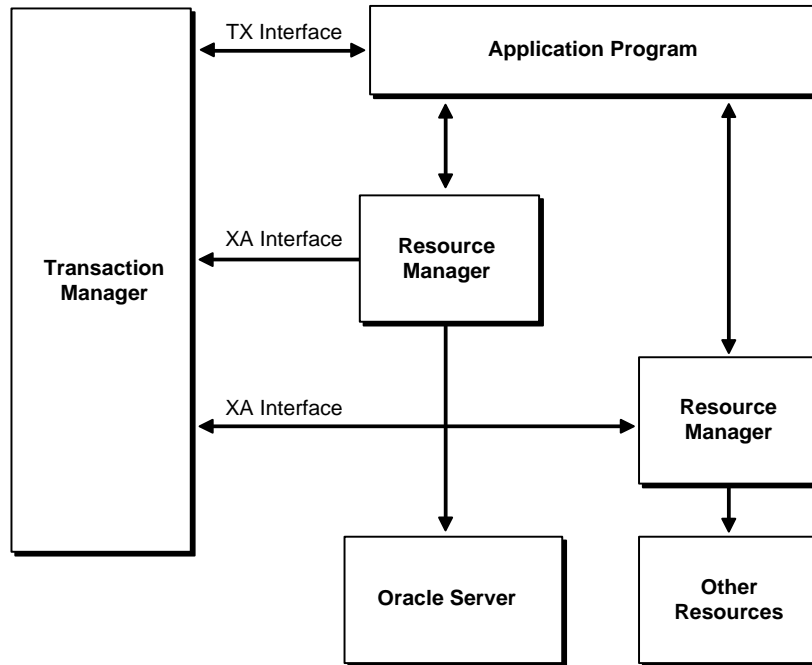
The DTP model that specifies how a transaction manager and resource managers interact with an application program is described in the X/Open guide *Distributed Transaction Processing Reference Model* and related publications, which you can obtain by writing to

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 Pro*COBOL 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 run-time 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 *Oracle Call Interface Programmer's Guide*. Note that OCOM, OCON, OCOF, ORLON, OLON, OLOG, and OLOGOF cannot be issued by an X/Open application.

Linking

To get XA functionality, you must link the XA library to your X/Open application object modules. For instructions, see your system-specific Oracle8 manuals.

5

Using Embedded SQL

This chapter helps you to understand and apply the basic techniques of embedded SQL programming. Topics are:

- Using Host Variables
- Using Indicator Variables
- The Basic SQL Statements
- Cursors
- Sample Program 2: Cursor Operations
- Sample Program 4: Datatype Equivalencing

Using Host Variables

Oracle uses host variables to pass data and status information to your 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. In fact, input host variables can appear in a SQL statement wherever a value or expression is allowed.

> **Attention:** In an ORDER BY clause, you *can* use a host variable, but it is treated as a constant or literal, and hence the contents of the host variable have no effect. For example, the SQL statement
>
> ```
> EXEC SQL SELECT ENAME, EMPNO INTO :NAME, :NUMBER
>     FROM EMP
>     ORDER BY :ORD
> END-EXEC.
> ```

appears to contain an input host variable, *ORD*. However, the host variable in this case is treated as a constant, and regardless of the value of *ord*, no ordering is done.

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 (sometimes called *DDL*) such as ALTER, CREATE, and DROP. In the following example, the DROP TABLE statement is *invalid*:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 TABLE-NAME    PIC X(30) VARYING.
     ...
EXEC SQL END DECLARE SECTION END-EXEC.
...
DISPLAY 'Table name? '.
ACCEPT TABLE-NAME.
EXEC SQL DROP TABLE :TABLE-NAME END-EXEC.
*  -- host variable not allowed
```

Before Oracle executes a SQL statement containing input host variables, your program must assign values to them. Consider the following example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01    EMP-NUMBER   PIC S9(4) COMP.
    01    EMP-NAME     PIC X(20) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
...
* -- get values for input host variables
DISPLAY 'Employee number? '.
ACCEPT EMP-NUMBER.
DISPLAY 'Employee name? '.
ACCEPT EMP-NAME.
EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
    VALUES (:EMP-NUMBER, :EMP-NAME)
END-EXEC.
```

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.

Input 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 assigns the value of the host variable to the column.

Output Variables

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

-2	Oracle assigned a truncated column value to the host variable, but could not assign the original length of the column value to the indicator variable because the number was too large.
-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, assigned the original column length (expressed in characters, instead of bytes, for multi-byte NLS host variables) to the indicator variable, and set SQLCODE in the SQLCA to zero.

Remember, an indicator variable must be declared as a 2-byte integer and, in SQL statements, must be prefixed with a colon and appended to its host variable (unless you use the keyword INDICATOR).

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:

```
MOVE -1 TO IND-COMM.
EXEC SQL INSERT INTO EMP (EMPNO, COMM)
    VALUES (:EMP-NUMBER, :cOMMISSION:IND-COMM)
END-EXEC.
```

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

You can hard-code the null instead, as follows:

```
EXEC SQL INSERT INTO EMP (EMPNO, COMM)
    VALUES (:EMP-NUMBER, NULL)
END-EXEC.
```

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: '
```

```
          WITH NO ADVANCING.
      ACCEPT EMP-NUMBER.
      IF EMP-NUMBER = 0
          MOVE -1 TO IND-EMPNUM
      ELSE
          MOVE 0 TO IND-EMPNUM
      END-IF.
      EXEC SQL INSERT INTO EMP (EMPNO, SAL)
          VALUES (:EMP-NUMBER:IND-EMPNUM, :SALARY)
      END-EXEC.
```

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
      END-EXEC.
      IF IND-COMM = -1
          MOVE SALARY TO PAY.
*   -- commission is null; ignore it
      ELSE
          ADD SALARY TO COMMISSIO GIVING PAY.
      END-IF.
```

Fetching Nulls

Using the precompiler option UNSAFE_NULL=YES, you can select or fetch nulls into a host variable that lacks 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
      END-EXEC.
```

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

There is no way to know whether or not a NULL was returned, or the value of the host variable if a NULL is returned. This is to be avoided, thus the name of the option. UNSAFE_NULL=YES should not be used in new applications. It is provided only for backward compatibility.

However, when UNSAFE_NULL=NO, if you select or fetch nulls into a host variable that lacks an indicator variable, Oracle issues the following error message:

```
ORA-01405: fetched column value is NULL
```
For more information, see "UNSAFE_NULL" on page 7-37.

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
END-EXEC.
```

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))
END-EXEC.
```

Fetching Truncated Values

If a value is truncated when fetched into a host variable, 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 data manipulation statements (sometimes called *DML*) and cursor control statements. The following 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 rows from a table.

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

Alternatively, when MODE={ANSI|ANSI14}, you can check the status variable SQLSTATE or SQLCODE. For more information, see "Using Status Variables when MODE={ANSI|ANSI14}" on page 9-4.

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 an explicitly declared cursor or cursor variable (or the use of host arrays, which are discussed in Chapter 10, "Using Host Tables"). The following embedded SQL statements let you define and control 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 multi-row SELECT statements. For a detailed discussion of each statement and its clauses, see the *Oracle8 SQL Reference.*

Selecting Rows

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
END-EXEC.
```

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 (in which EMPNO is a unique key). However, if a query can return more than one row, you must fetch the rows using a cursor or select them into a host array.

If a query is written to return only one row but might actually return several rows, the result depends on how you specify the option SELECT_ERROR. When SELECT_ERROR=YES (the default), Oracle issues the following error message if more than one row is returned:

```
ORA-01422: exact fetch returns more than requested number of rows
```
When SELECT_ERROR=NO, a row is returned and Oracle generates no error.

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, and FOR UPDATE OF.

Inserting Rows

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)
END-EXEC.
```

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 in the same order they were defined by CREATE TABLE.

Using Subqueries

A *subquery* is a nested SELECT statement. Subqueries let you conduct multi-part 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

For example, to copy rows from one table to another, replace the VALUES clause in an INSERT statement with a subquery, as follows:

```
EXEC SQL INSERT INTO EMP2 (EMPNO, ENAME, SAL, DEPTNO)
    SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP
    WHERE JOB = :JOB-TITLE
END-EXEC.
```

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

Updating Rows

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
END-EXEC.
```

You can use the optional WHERE clause to specify the conditions under which rows are updated. See "Using the WHERE Clause" on page 5-10.

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
END-EXEC.
```

Deleting Rows

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
END-EXEC.
```

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 SQL-WARN(5) in the SQLCA to 'W' to warn that all rows were processed.

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

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.

Declaring a Cursor

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 EMPCURSOR CURSOR FOR
    SELECT ENAME, EMPNO, SAL
    FROM EMP
    WHERE DEPTNO = :DEPT_NUMBER
END-EXEC.
```

The cursor name is an identifier used by the precompiler, *not* a host or program variable, and should not be declared in a COBOL statement. Therefore, cursor

names cannot be passed from one precompilation unit to another. 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 WITH HOLD clause can be used in a DECLARE CURSOR statement to hold the cursor open after a COMMIT or a ROLLBACK.

The precompiler option CLOSE_ON_COMMIT is provided for use in the command line or in a configuration file. Any cursor not declared with the WITH HOLD clause is closed after a COMMIT or ROLLBACK when CLOSE_ON_COMMIT=YES. See "WITH HOLD Clause in DECLARE CURSOR Statements" on page 8-5, and "CLOSE_ON_COMMIT" on page 7-14.

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 EMPCURSOR END-EXEC.
*     -- MISPLACED OPEN STATEMENT
      EXEC SQL DECLARE EMPCURSOR CURSOR FOR
          SELECT ENAME, EMPNO, SAL
          FROM EMP
          WHERE ENAME = :EMP-NAME
      END-EXEC.
```

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 "MAXOPENCURSORS" on page 7-28.

Opening a Cursor

Use the OPEN statement to execute the query and identify the active set. In the following example, a cursor named *EMPCURSOR* is opened.

```
EXEC SQL OPEN EMPCURSOR END-EXEC.
```

OPEN positions the cursor just before the first row of the active set. It also zeroes the rows-processed count kept by SQLERRD(3) 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 CLOSE_ON_COMMIT=YES, you need not close a cursor before reopening it. This can boost performance; for details, see Appendix D, "Performance Tuning"The amount of work done by OPEN depends on the values of three precompiler options: HOLD_CURSOR, RELEASE_CURSOR, and MAXOPENCURSORS. For more information, see "Using Pro*COBOL Options" on page 7-11.

Fetching from a Cursor

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 EMPCURSOR
    INTO :EMP-NAME, :EMP-NUMBER, :SALARY
END-EXEC.
```

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 CLOSE_ON_COMMIT=NO, 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 EMPCURSOr CURSOR FOR
        SELECT ENAME, SAL FROM EMP WHERE DEPTNO = 20
    END-EXEC.
    ...
    EXEC SQL OPEN EMPCURSOr END-EXEC.
    EXEC SQL WHENEVER NOT FOUND DO ...
 PERFORM
    EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME1, :SAL1 END-EXEC
    EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME2, :SAL2 END-EXEC
    EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME3, :SAL3 END-EXEC
    ...
 END-PERFORM.
```

If the active set is empty or contains no more rows, FETCH returns the "no data found" Oracle warning code to SQLCODE in the SQLCA (or when MODE=ANSI, to the status variable SQLSTATE). The status of the output host variables is indeterminate. (In a typical program, the WHENEVER NOT FOUND statement detects this error.) To reuse the cursor, you must reopen it.

Closing a Cursor

When finished 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 options HOLD_CURSOR and RELEASE_CURSOR. In the following example, you close the cursor named *EMPCURSOR*:

```
    EXEC SQL CLOSE EMPCURSOR END-EXEC.
```

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 CLOSE_ON_COMMIT=NO, issuing a commit or rollback closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by a commit or rollback and if open, remain open. However, when CLOSE_ON_COMMIT=YES, issuing a commit or rollback closes *all* explicit cursors.

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 refer-enced in the CURRENT OF clause of an UPDATE or DELETE statement. The CUR-RENT OF clause signals the precompiler to add a FOR UPDATE clause if necessary. For more information, see "Mimicking the CURRENT OF Clause" on page 10-14.

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

```
    EXEC SQL DECLARE EMPCURSOR CURSOR FOR
        SELECT ENAME, SAL FROM EMP WHERE JOB = 'CLERK'
        FOR UPDATE OF SAL
    END-EXEC.
    ...
    EXEC SQL OPEN EMPCURSOR END-EXEC.
    EXEC SQL WHENEVER NOT FOUND DO ...
 PERFORM
    EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME, :SALARY
    END-EXEC
    ...
    EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
        WHERE CURRENT OF EMPCURSOR
    END-EXEC
 END-PERFORM.
```

Restrictions

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, and are released when you commit or rollback. If you try to fetch from a FOR UPDATE cursor after a commit, Oracle generates the following error:

```
ORA-01002: fetch out of sequence
```

You cannot use host arrays with the CURRENT OF clause. For an alternative, see "Mimicking the CURRENT OF Clause" on page 10-14. Also, 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 the CUR-
RENT OF clause in dynamic SQL.

A Typical Sequence of Statements

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

```
* --  Define a cursor.
      EXEC SQL DECLARE EMPCURSOR CURSOR FOR
          SELECT ENAME, JOB FROM EMP
          WHERE EMPNO = :EMP-NUMBER
          FOR UPDATE OF JOB
      END-EXEC.
* --  Open the cursor and identify the active set.
      EXEC SQL OPEN EMPCURSOR END-EXEC.
* --  Exit if the last row was already fetched.
      EXEC SQL
          WHENEVER NOT FOUND DO PERFORM NO-MORE
      END-EXEC.
* --  Fetch and process data in a loop.
 PERFORM
        EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME, :JOB-TITLE
        END-EXEC
* --  host-language statements that operate on the fetched data
      EXEC SQL UPDATE EMP
          SET JOB = :NEW-JOB-TITLE
          WHERE CURRENT OF EMPCURSOR
      END-EXEC
 END-PERFORM.
      ...
 MO-MORE.
* --  Disable the cursor.
      EXEC SQL CLOSE EMPCURSOR END-EXEC.
      EXEC SQL COMMIT WORK RELEASE END-EXEC.
      STOP RUN.
```

Sample Program 2: Cursor Operations

This program logs on to Oracle, declares and opens a cursor, fetches the names, sal-
aries, and commissions of all salespeople, displays the results, then closes the cursor

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 warning code to SQLCODE in the SQLCA. The cumulative number of rows actually fetched is found in SQLERRD(3) in the SQLCA.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CURSOR-OPS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME          PIC X(10) VARYING.
        01  PASSWD            PIC X(10) VARYING.
        01  EMP-REC-VARS.
            05  EMP-NAME   PIC X(10) VARYING.
            05  SALARY     PIC S9(6)V99
                            DISPLAY SIGN LEADING SEPARATE.
            05  COMMISSIO  PIC S9(6)V99
                            DISPLAY SIGN LEADING SEPARATE.
        EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
        EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.
    EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL INCLUDE SQLCA END-EXEC.


        01  DISPLAY-VARIABLES.
            05  D-EMP-NAME   PIC X(10).
            05  D-SALARY     PIC Z(4)9.99.
            05  D-COMMISSION PIC Z(4)9.99.

PROCEDURE DIVISION.

BEGIN-PGM.
    EXEC SQL
        WHENEVER SQLERROR DO PERFORM SQL-ERROR
    END-EXEC.
    PERFORM LOGON.
    EXEC SQL
        DECLARE SALESPEOPLE CURSOR FOR
        SELECT ENAME, SAL, COMM FROM EMP
        WHERE JOB LIKE 'SALES%'
    END-EXEC.
    EXEC SQL
        OPEN SALESPEOPLE
    END-EXEC.
    DISPLAY "SALESPERSON    SALARY        COMMISSION".
    DISPLAY "-----------    ----------    ----------".
```

```
            FETCH-LOOP.
                EXEC SQL
                    WHENEVER NOT FOUND DO PERFORM SIGN-OFF
                END-EXEC.
                EXEC SQL
                    FETCH SALESPEOPLE
                    INTO :EMP-NAME, :SALARY, :COMMISSION
                END-EXEC.
                MOVE EMP-NAME-ARR TO D-EMP-NAME.
                MOVE SALARY TO D-SALARY.
                MOVE COMMISSION TO D-COMMISSION.
                DISPLAY D-EMP-NAME, "      ", D-SALARY, "      ",
        -           D-COMMISSION.
                MOVE SPACES TO EMP-NAME-ARR.
                GO TO FETCH-LOOP.

            LOGON.
                MOVE "SCOTT" TO USERNAME-ARR.
                MOVE 5 TO USERNAME-LEN.
                MOVE "TIGER" TO PASSWD-ARR.
                MOVE 5 TO PASSWD-LEN.
                EXEC SQL
                    CONNECT :USERNAME IDENTIFIED BY :PASSWD
                END-EXEC.
                DISPLAY " ".
                DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
                DISPLAY " ".

            SIGN-OFF.
                EXEC SQL
                    CLOSE SALESPEOPLE
                END-EXEC.
                DISPLAY " ".
                DISPLAY "HAVE A GOOD DAY.".
                DISPLAY " ".
                EXEC SQL
                    COMMIT WORK RELEASE
                END-EXEC.
                STOP RUN.

            SQL-ERROR.
                EXEC SQL
                    WHENEVER SQLERROR CONTINUE
                END-EXEC.
```

```
                DISPLAY " ".
                DISPLAY "ORACLE ERROR DETECTED:".
                DISPLAY " ".
                DISPLAY SQLERRMC.
                EXEC SQL
                    ROLLBACK WORK RELEASE
                END-EXEC.
                STOP RUN.
```

Sample Program 4: Datatype Equivalencing

After connecting to Oracle, this program creates a database table named IMAGE in the SCOTT account, then simulates the insertion of bitmap images of employee numbers into the table. Datatype equivalencing lets the program use the Oracle external datatype LONG RAW to represent the images. Later, when the user enters an employee number, the number's "bitmap" is selected from the IMAGE table and pseudo-displayed on the terminal screen.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DTY-EQUIV.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME          PIC X(10) VARYING.
        01  PASSWD            PIC X(10) VARYING.
        01  EMP-REC-VARS.
            05  EMP-NUMBER    PIC S9(4) COMP.
            05  EMP-NAME      PIC X(10) VARYING.
            05  SALARY        PIC S9(6)V99
                              DISPLAY SIGN LEADING SEPARATE.
            05  COMMISSION    PIC S9(6)V99
                              DISPLAY SIGN LEADING SEPARATE.
            05  COMM-IND      PIC S9(4) COMP.

    EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
    EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.
        01  BUFFER-VAR.
            05  BUFFER        PIC X(8192).
    EXEC SQL VAR BUFFER IS LONG RAW END-EXEC.
        01  SELECTION         PIC S9(4) COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
```

```
        01  DISPLAY-VARIABLES.
            05  D-EMP-NAME    PIC X(10).
            05  D-SALARY      PIC $Z(4)9.99.
            05  D-COMMISSION  PIC $Z(4)9.99.
        01  REPLY            PIC X(10).
        01  INDX             PIC S9(9) COMP.
        01  PRT-QUOT         PIC S9(9) COMP.
        01  PRT-MOD          PIC S9(9) COMP.

    PROCEDURE DIVISION.

    BEGIN-PGM.
        EXEC SQL
            WHENEVER SQLERROR DO PERFORM SQL-ERROR
        END-EXEC.
        PERFORM LOGON.
        DISPLAY "OK TO DROP THE IMAGE TABLE? (Y/N) "
            WITH NO ADVANCING.
        ACCEPT REPLY.
        IF (REPLY NOT = "Y") AND (REPLY NOT = "Y")
            PERFORM SIGN-OFF.
        EXEC SQL
            WHENEVER SQLERROR CONTINUE
        END-EXEC.
        EXEC SQL
            DROP TABLE IMAGE
        END-EXEC.
        DISPLAY " ".
        IF (SQLCODE = 0) DISPLAY
            "TABLE IMAGE DROPPED - CREATING NEW TABLE."
        ELSE IF (SQLCODE = -942) DISPLAY
            "TABLE IMAGE DOES NOT EXIST - CREATING NEW TABLE."
        ELSE PERFORM SQL-ERROR.
        EXEC SQL
            WHENEVER SQLERROR DO PERFORM SQL-ERROR
        END-EXEC.
        EXEC SQL
            CREATE TABLE IMAGE
            (EMPNO NUMBER(4) NOT NULL, BITMAP LONG RAW)
        END-EXEC.
        EXEC SQL
            DECLARE EMPCUR CURSOR FOR
            SELECT EMPNO, ENAME FROM EMP
        END-EXEC.
        EXEC SQL
```

```
              OPEN EMPCUR
         END-EXEC.
         DISPLAY " ".
         DISPLAY "INSERTING BITMAPS INTO IMAGE FOR ALL EMPLOYEES.".
         DISPLAY " ".

     INSERT-LOOP.
         EXEC SQL
             WHENEVER NOT FOUND GOTO NOT-FOUND
         END-EXEC.
         EXEC SQL
             FETCH EMPCUR INTO :EMP-NUMBER, :EMP-NAME
         END-EXEC.
         MOVE EMP-NAME-ARR TO D-EMP-NAME.
         DISPLAY "EMPLOYEE ", D-EMP-NAME WITH NO ADVANCING.
         PERFORM GET-IMAGE.
         EXEC SQL
             INSERT INTO IMAGE VALUES (:EMP-NUMBER, :BUFFER)
         END-EXEC.
         DISPLAY " IS DONE!".
         MOVE SPACES TO EMP-NAME-ARR.
         GO TO INSERT-LOOP.

     NOT-FOUND.
         EXEC SQL
             CLOSE EMPCUR
         END-EXEC.
         EXEC SQL
             COMMIT WORK
         END-EXEC.
         DISPLAY " ".
         DISPLAY "DONE INSERTING BITMAPS.  NEXT, DISPLAY SOME.".

     DISP-LOOP.
         MOVE 0 TO SELECTION.
         DISPLAY " ".
         DISPLAY "ENTER EMP NUMBER (0 TO QUIT): "
-            WITH NO ADVANCING.
         ACCEPT SELECTION.
         IF (SELECTION = 0) PERFORM SIGN-OFF.
         EXEC SQL
             WHENEVER NOT FOUND GOTO NO-EMP
         END-EXEC.
         EXEC SQL
             SELECT EMP.EMPNO, ENAME, SAL, COMM, BITMAP
```

```
                    INTO :EMP-NUMBER, :EMP-NAME, :SALARY,
                        :COMMISSION:COMM-IND, :BUFFER
                FROM EMP, IMAGE
            WHERE EMP.EMPNO = :SELECTION AND EMP.EMPNO = IMAGE.EMPNO
            END-EXEC.
            DISPLAY " ".
            PERFORM SHOW-IMAGE.
            MOVE EMP-NAME-ARR TO D-EMP-NAME.
            MOVE SALARY TO D-SALARY.
            MOVE COMMISSION TO D-COMMISSION.
            DISPLAY "EMPLOYEE ", D-EMP-NAME, " HAS SALARY ", D-SALARY
                WITH NO ADVANCING.
            IF COMM-IND = -1
                DISPLAY " AND NO COMMISSION."
            ELSE
                DISPLAY " AND COMMISSION ", D-COMMISSION
            END-IF.
            MOVE SPACES TO EMP-NAME-ARR.
            GO TO DISP-LOOP.

    NO-EMP.
            DISPLAY "NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.".
            GO TO DISP-LOOP.

    LOGON.
            MOVE "SCOTT" TO USERNAME-ARR.
            MOVE 5 TO USERNAME-LEN.
            MOVE "TIGER" TO PASSWD-ARR.
            MOVE 5 TO PASSWD-LEN.
            EXEC SQL
                CONNECT :USERNAME IDENTIFIED BY :PASSWD
            END-EXEC.
            DISPLAY " ".
            DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
            DISPLAY " ".

    GET-IMAGE.
            PERFORM MOVE-IMAGE
                VARYING INDX FROM 1 BY 1 UNTIL INDX > 8192.

    MOVE-IMAGE.
            STRING '*' DELIMITED BY SIZE INTO BUFFER WITH POINTER
    -           INDX.
            DIVIDE 256 INTO INDX GIVING PRT-QUOT REMAINDER
    -           PRT-MOD.
```

```
        IF (PRT-MOD = 0) DISPLAY "." WITH NO ADVANCING.

SHOW-IMAGE.
    PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > 10
        DISPLAY "            *************************"
    END-PERFORM.
    DISPLAY " ".

SIGN-OFF.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL
        COMMIT WORK RELEASE
    END-EXEC.
    STOP RUN.

SQL-ERROR.
    EXEC SQL
        WHENEVER SQLERROR CONTINUE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL
        ROLLBACK WORK RELEASE
    END-EXEC.
    STOP RUN.
```

6

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:

- Advantages of PL/SQL

- Embedding PL/SQL Blocks

- Using Host Variables

- Using Indicator Variables

- Using Host Tables

- Using Cursors

- Stored Subprograms

- Stored Subprograms

- Sample Program 9: Calling a Stored Procedure

- Using Dynamic PL/SQL

- Cursor Variables

Advantages of PL/SQL

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

- better performance
- integration with Oracle8
- cursor FOR loops
- procedures and functions
- packages
- PL/SQL tables
- user-defined records

For more information about PL/SQL, see the *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, Oracle8 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 Oracle8.

Integration with Oracle8

PL/SQL is tightly integrated with the Oracle8 Server. For example, most PL/SQL datatypes are native to the Oracle8 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.

Subprograms

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 a new department, just write your own as follows:

```
PROCEDURE create_dept
    (new_dname  IN CHAR(14),
     new_loc    IN CHAR(13),
     new_deptno 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 a new 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 can store subprograms in the database (using CREATE FUNCTION and CREATE PROCEDURE) that can be called from multiple applications without needing to be re-compiled each time.

Parameter Modes

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 4–11 shows the legal conversions between datatypes.

Packages

PL/SQL lets you bundle logically related types, program objects, and subprograms into a *package*. Packages can be compiled and stored in an Oracle8 database, where their contents can be shared by multiple 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 modeled 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*:

```
DECLARE
    TYPE NumTabTyp IS TABLE OF NUMBER
        INDEX BY BINARY_INTEGER;
    ...
BEGIN
    ...
END;
```

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 database 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 := 10,  -- must initialize
         dname   CHAR(9),
         loc     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. However, you must initialize NOT NULL fields.

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

Pro*COBOL treats 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.
```

When your program embeds PL/SQL blocks, you must specify the precompiler option SQLCHECK=SEMANTICS because PL/SQL must be parsed by Oracle8. To

connect to Oracle8, you must also specify the option USERID. For more information, see "Using Pro*COBOL Options" on page 7-11.

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. However, character host variables cannot exceed 255 characters in length. 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 END-EXEC.
          01 USERNAME   PIC X(20) VARYING.
          01 PASSWORD   PIC X(20) VARYING.
          01 EMP-NUMBER PIC S9(4) COMP.
          01 JOB-TITLE  PIC X(20) VARYING.
          01 HIRE-DATE  PIC X(9) VARYING.
          01 SALARY     PIC S9(6)V99
                            DISPLAY SIGN LEADING SEPARATE.
      EXEC SQL END DECLARE SECTION END-EXEC.
      EXEC SQL INCLUDE SQLCA END-EXEC.
      ...
      DISPLAY 'Username? ' WITH NO ADVANCING.
      ACCEPT USERNAME.
      DISPLAY 'Password? ' WITH NO ADVANCING.
      ACCEPT PASSWORD.
      EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR.
      EXEC SQL
          CONNECT :USERNAME IDENTIFIED BY :PASSWORD
      END-EXEC.
      DISPLAY 'Connected to Oracle'.
```

```
      PERFORM
          DISPLAY 'Employee Number (0 to end)? 'WITH NO ADVANCING
          ACCEPTd EMP-NUMBER
          IF EMP-NUMBER = 0
              EXEC SQL COMMIT WORK RELEASE END-EXEC
              DISPLAY 'Exiting program'
              STOP RUN
          END-IF.
*         --------------- 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.
       END-PERFORM.
          ...
       SQL-ERROR.
          EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
          EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
          DISPLAY 'Processing error'.
          STOP RUN.
```

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 END-EXEC.
          01 USERNAME   PIC X(20) VARYING.
          01 ACCT-NUM   PIC S9(4) COMP.
          01 TRANS-TYPE PIC X(1).
          01 TRANS-AMT  PIC PIC S9(6)V99
```

```
                        DISPLAY SIGN LEADING SEPARATE.
             01 STATUS      PIC X(80) VARYING.
         EXEC SQL END DECLARE SECTION END-EXEC.
         EXEC SQL INCLUDE SQLCA END-EXEC.
         DISPLAY 'Username? 'WITH NO ADVANCING.
         ACCEPT USERNAME.
         DISPLAY 'Password? '.
         ACCEPT PASSWORD.
         EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR.
         EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD.
         PERFORM
         DISPLAY 'Account Number (0 to end)? '
             WITH NO ADVANCING
         ACCEPT ACCT_NUM
         IF ACCT-NUM = 0
             EXEC SQL COMMIT WORK RELEASE END-EXEC
             DISPLAY 'Exiting program' WITH NO ADVANCING
             STOP RUN
         END-IF.
         DISPLAY 'Transaction Type - D)ebit or C)redit? '
             WITH NO ADVANCING
         ACCEPT TRANS-TYPE
         DISPLAY 'Transaction Amount? '
         ACCEPT trans_amt
*    -------------------- begin PL/SQL block ------------------
         EXEC SQL EXECUTE
            DECLARE
                old_bal      NUMBER(9,2);
                err_msg      CHAR(70);
                nonexistent  EXCEPTION;
            BEGIN
               :TRANS-TYP-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';
                  WHEN OTHERS THEN
                        err_msg := SUBSTR(SQLERRM, 1, 70);
                        :STATUS := 'Error: ' || err_msg;
          END;
        END-EXEC.
*       ------------------ end PL/SQL block ----------------------
        DISPLAY 'Status: ', STATUS
    END-PERFORM.
        ...
    SQL-ERROR.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
        DISPLAY 'Processing error'.
        STOP RUN.
```

VARCHAR Pseudotype

Recall from Chapter 4, "Advanced Pro*COBOL Programs", that you can use the
VARCHAR pseudotype to declare variable-length character strings. If the
VARCHAR is an input host variable, you must tell Oracle8 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, Oracle8 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 END-EXEC.
    01 EMP-NUM   PIC S9(4) COMP.
    01 EMP-NAME  PIC X(10) VARYING.
    01 SALARY    PIC S9(6)V99
                 DISPLAY SIGN LEADING SEPARATE.
```

```
        ...
        EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
        ...
* -- initialize length field
        MOVE 10 TO EMP-NAME-LEN.
        EXEC SQL EXECUTE
        BEGIN
            SELECT ename, sal INTO :EMP-NAME, :SALARY
                FROM emp
                WHERE empno = :EMP-NUM;
         ...
        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 ...
```

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 an 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-NUM;
    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 Tables

You can pass input host tables and indicator tables 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 table is passed to PL/SQL, but you can use the ARRAYLEN statement (discussed later) to specify a smaller table dimension.

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

> **Note:** Pro*COBOL does not check your usage of host tables. For instance, no index range checking is done.

In the example below, you pass a host table named *salary* to a PL/SQL block, which uses the host table 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 END-EXEC.
    ...
        01 SALARY OCCURS 100 TIMES PIC S9(6)V99
        DISPLAY SIGN LEADING SEPARATE.
        01 MEDIAN-SALARY  PIC S9(6)V99
        DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL END DECLARE SECTION END-EXEC.
* -- populate the host table
    EXEC SQL EXECUTE
    DECLARE
        TYPE NumTabTyp IS TABLE OF REAL
            INDEX BY BINARY_INTEGER;
        n  BINARY_INTEGER;
```

```
          ...
          FUNCTION median (num_tab NumTabTyp, n INTEGER)
              RETURN REAL IS
          BEGIN
* -- compute median
          END;
      BEGIN
          n := 100;
          :MEDIAN-SALARY := median(:SALARY    END;
      END-EXEC.
```

You can also use a subprogram call to assign all row values in a PL/SQL table to corresponding elements in a host table. For an example, see "Stored Subprograms" on page 6-21.

Table 6–1   shows the legal conversions between row values in a PL/SQL table and elements in a host table. For example, a host table 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 6–1   Legal Datatype Conversions

PL/SQL Table							
Host table	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	X							
CHARZ	X							
DATE		X						
DECIMAL					X			
DISPLAY					X			
FLOAT					X			
INTEGER					X			
LONG	X		X					
LONG VAR-CHAR			X	X		X		X
LONG VARRAW				X		X		
NUMBER					X			
RAW				X		X		
ROWID							X	
STRING			X	X		X		X
UNSIGNED					X			
VARCHAR			X	X		X		X
VARCHAR2			X	X		X		X
VARNUM					X			
VARRAW				X		X		

ARRAYLEN Statement

Suppose you must pass an input host table to a PL/SQL block for processing. By default, when binding such a host table, Pro*COBOL use its declared dimension. However, you might not want to process the entire table. In that case, you can use

the ARRAYLEN statement to specify a smaller table dimension. ARRAYLEN associates the host table with a host variable, which stores the smaller dimension. The statement syntax is:

```
EXEC SQL ARRAYLEN host_array (dimension) EXECUTE END-EXEC.
```

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

The ARRAYLEN statement must appear somewhere after the declarations of *host_array* and *dimension*. You cannot specify an offset into the host table. However, you might be able to use COBOL features for that purpose.

In the following example, you use ARRAYLEN to override the default dimension of a host table named *bonus*:

```
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
          01 BONUS OCCURS 100 TIMES PIC S9(6)V99
          DISPLAY SIGN LEADING SEPARATE.
          01 MY-DIM  PIC S9(4) COMP.
      ...
      EXEC SQL ARRAYLEN BONUS (MY-DIM) END-EXEC.
      EXEC SQL END DECLARE SECTION END-EXEC.
* -- populate the host table
      ...
* -- set smaller table dimension
      MOVE 25 TO MY-DIM.
      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 table elements are passed to the PL/SQL block because ARRAYLEN reduces the host table from 100 to 25 elements. As a result, when the PL/SQL block is sent to Oracle8 for execution, a much smaller host table is sent along. This saves time and, in a networked environment, reduces network traffic.

Optional Keyword EXECUTE

Host tables used in a dynamic SQL method 2 EXEC SQL EXECUTE statement may have two different interpretations based on the presence or absence of the optional keyword EXECUTE. See "Using Method 2" on page 11 - 13.

By default (if the EXECUTE keyword is absent):

- The host array is considered when determining the number of times a PL/SQL block will be executed. The minimum array dimension is used.

- The host array must not be bound to a PL/SQL index table.

If the keyword EXECUTE is present:

- The host table must be bound to an index table.

- The PL/SQL block will be executed one time.

- All host variables specified in the EXEC SQL EXECUTE statement must either

  - be specified in an ARRAYLEN ... EXECUTE statement, or

  - be a scalar.

For example, given the following PL/SQL procedure:

```
CREATE OR REPLACE PACKAGE pkg AS
   TYPE tab IS TABLE OF NUMBER(5) INDEX BY BINARY_INTEGER;
   PROCEDURE proc1 (parm1 tab, parm2 NUMBER, parm3 tab);
END;
```

The following Pro*COBOL example demonstrates how host tables can be used to determine how many times a given PL/SQL block is executed. In this case, the PL/SQL block will be execute 3 times resulting in 3 new rows in the *emp* table.

```
    ...
    01  DYNSTMT   PIC X(80) VARYING.
    01  EMPNOTAB  PIC S9(4) COMPUTATIONAL OCCURS 5 TIMES.
    01  ENAMETAB  PIC X(10) OCCURS 3 TIMES.
    01  DIM       PIC S9(9) COMP VALUE 2.
    ...
        MOVE 1111 TO EMPNOTAB(1).
        MOVE 2222 TO EMPNOTAB(2).
        MOVE 3333 TO EMPNOTAB(3).
        MOVE 4444 TO EMPNOTAB(4).
        MOVE 5555 TO EMPNOTAB(5).

        MOVE "MICKEY" TO ENAMETAB(1).
```

```
                        MOVE "MINNIE" TO ENAMETAB(2).
                        MOVE "GOOFY" TO ENAMETAB(3).

                        MOVE "BEGIN INSERT INTO emp(empno, ename) VALUES :b1, :b2; END;"
                          TO DYNSTMT-ARR.
                        MOVE 57 TO DYNSTMT-LEN.

                        EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.
                        EXEC SQL EXECUTE s1 USING :EMPNOTAB, :ENAMETAB END-EXEC.
                 ...
```

The following Pro*COBOL example demonstrates how to bind a host table to a
PL/SQL index table through dynamic method 2. Note the presence of the
ARRAYLEN...EXECUTE statement for all host arrays specified in the EXEC SQL
EXECUTE statement.

```
                 ...
                 01 DYNSTMT   PIC X(80) VARYING.
                 01 II        PIC S9(4) COMP VALUE 2.
                 01 INTTAB    PIC S9(9) COMP OCCURS 3 TIMES.
                 01 DIM       PIC S9(9) COMP VALUE 3.

                        EXEC SQL ARRAYLEN INTTAB (DIM) EXECUTE END-EXEC.
                 ...
                        MOVE 1 TO INTTAB(1).
                        MOVE 2 TO INTTAB(2).
                        MOVE 3 TO INTTAB(3).

                        MOVE "BEGIN pkg.proc1 (:v1, :v2, :v3); end;";
                          TO DYNSTMT-ARR.
                        MOVE 37 TO DYNSTMT-LEN.

                        EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.
                        EXEC SQL EXECUTE s1 USING :INTTAB, :II, :INTTAB END-EXEC.
                 ...
```

However, the following Pro*COBOL example will result in a precompile-time error
because there is no ARRAYLEN...EXECUTE statement for INTTAB2.

```
                 ...
                 01 DYNSTMT   PIC X(80) VARYING.
                 01 INTTAB    PIC S9(9) COMP OCCURS 3 TIMES.
                 01 INTTAB2   PIC S9(9) COMP OCCURS 3 TIMES.
                 01 DIM       PIC S9(9) COMP VALUE 3.
```

```
      EXEC SQL ARRAYLEN INTTAB (DIM) EXECUTE END-EXEC.
...
    MOVE 1 TO INTTAB(1).
    MOVE 2 TO INTTAB(2).
    MOVE 3 TO INTTAB(3).

    MOVE "BEGIN pkg.proc1 (:v1, :v2, :v3); end;";
      TO DYNSTMT-ARR.
    MOVE 37 TO DYNSTMT-LEN.

    EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.
    EXEC SQL EXECUTE s1 USING :INTTAB, :INTTAB2, :INTTAB END-EXEC.
...
```

Using Cursors

Every embedded SQL statement is assigned a cursor, either explicitly by you in a
DECLARE CURSOR statement or implicitly by Pro*COBOL. Internally,
Pro*COBOL maintains 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 Oracle8.

Various precompiler 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 Oracle8 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 Oracle8 initialization parameter OPEN_CURSORS. Figure 6–1   shows you
how to calculate the maximum number of cursors in use.

Figure 6–1   Maximum Cursors in Use

```
        SQL statement cursors
        PL/SQL parent cursors
        PL/SQL child cursors
   +    6 cursors for overhead
   ─────────────────────────────
        Sum of cursors in use

Must not exceed OPEN_CURSORS
```
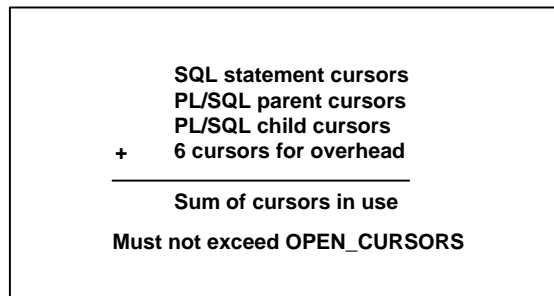
If your program exceeds the limit imposed by OPEN_CURSORS, you get the following Oracle8 error:

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

You can avoid this error by specifying the RELEASE_CURSOR=YES and HOLD_CURSOR=NO options. If you do not want to precompile 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) END-EXEC.
* --  first embedded PL/SQL block
    EXEC ORACLE OPTION (RELEASE_CURSOR=NO)END-EXEC.
* --  embedded SQL statements
    EXEC ORACLE OPTION (RELEASE_CURSOR=YES)END-EXEC.
* --  second embedded PL/SQL block
    EXEC ORACLE OPTION (RELEASE_CURSOR=NO)END-EXEC.
* --  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, there are no free cache entries, and HOLD_CURSOR=NO, then Pro*COBOL tries to reuse an entry. If you specify a very low value for MAXOPENCURSORS, then Pro*COBOL 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 Oracle8 database, and invoked. A subprogram explicitly created using an Oracle8 tool such as SQL*Plus or Server Manager is called a *stored* subprogram. Once compiled and stored in the data dictionary, it is a database object, which can be re-executed without being re-compiled.

When a subprogram within a PL/SQL block or stored subprogram is sent to Oracle8 by your application, it is called an *inline* subprogram. Oracle8 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 COBOL 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 | 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 tables. When the procedure finishes, it automatically assigns all row values in the PL/SQL tables to corresponding elements in the host tables.

```
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 re-grant privileges on it. For the full syntax of the CREATE statement see the *Oracle8 SQL Reference*.

If an embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement fails, Oracle8 generates a warning, not an error.

Calling a Stored Subprogram

To invoke (call) a stored subprogram from your COBOL 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 table, PL/SQL constant or variable, PL/SQL table, PL/SQL user-defined record, subprogram call, or expression. However, an actual OUT parameter cannot be a literal, subprogram call, or expression.

Sample Program 9: Calling a Stored Procedure

Before trying the sample program, you must create a PL/SQL package named *calldemo*, by running a script named CALLDEMO.SQL, which is supplied with Pro*COBOL and shown below. The script can be found in the Pro*COBOL demo library. Check your system-specific Oracle8 documentation for exact spelling of the script.

```
CREATE OR REPLACE PACKAGE calldemo AS

    TYPE name_array IS TABLE OF emp.ename%type
        INDEX BY BINARY_INTEGER;
    TYPE job_array IS TABLE OF emp.job%type
        INDEX BY BINARY_INTEGER;
    TYPE sal_array IS TABLE OF emp.sal%type
        INDEX BY BINARY_INTEGER;

    PROCEDURE get_employees(
      dept_number IN     number,    -- department to query
      batch_size  IN     INTEGER,   -- rows at a time
      found       IN OUT INTEGER,   -- rows actually returned
      done_fetch  OUT    INTEGER,   -- all done flag
      emp_name    OUT    name_array,
      job         OUT    job_array,
      sal         OUT    sal_array);

END calldemo;
/

CREATE OR REPLACE PACKAGE BODY calldemo AS

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


    -- Procedure "get_employees" fetches a batch of employee
    -- rows (batch size is determined by the client/caller
    -- of the procedure).  It can be called from other
    -- stored procedures or client application programs.
    -- The procedure opens the cursor if it is not
    -- already open, fetches a batch of rows, and
    -- returns the number of rows actually retrieved. At
    -- end of fetch, the procedure closes the cursor.
```

```
    PROCEDURE get_employees(
      dept_number IN      number,
      batch_size  IN      INTEGER,
      found       IN OUT INTEGER,
      done_fetch  OUT     INTEGER,
      emp_name    OUT     name_array,
      job         OUT     job_array,
      sal         OUT     sal_array) IS

  BEGIN
      IF NOT get_emp%ISOPEN THEN       -- open the cursor if
          OPEN get_emp(dept_number);  -- not already open
      END IF;

      -- Fetch up to "batch_size" rows into PL/SQL table,
      -- tallying rows found as they are retrieved. When all
      -- rows have been fetched, close the cursor and exit
      -- the loop, returning only the last set of rows found.

      done_fetch := 0;  -- set the done flag FALSE
      found := 0;

      FOR i IN 1..batch_size LOOP
          FETCH get_emp INTO emp_name(i), job(i), sal(i);
          IF get_emp%NOTFOUND THEN     -- if no row was found
              CLOSE get_emp;
              done_fetch := 1;   -- indicate all done
              EXIT;
          ELSE
              found := found + 1;  -- count row
          END IF;
      END LOOP;
    END;
END;
/
```

The following sample program connects to Oracle8, prompts the user for a
department number, then calls a PL/SQL procedure named *get_employees*, which is
stored in package *calldemo*. 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 tables. When the procedure finishes, row
values in the PL/SQL tables are automatically assigned to the corresponding
elements in the host tables. The program calls the procedure repeatedly, displaying
each batch of employee data, until no more data is found.

```
                IDENTIFICATION DIVISION.
                PROGRAM-ID. CALL-STORED-PROC.
                ENVIRONMENT DIVISION.
                DATA DIVISION.
                WORKING-STORAGE SECTION.

                    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
                        01  USERNAME        PIC X(15) VARYING.
                        01  PASSWD          PIC X(15) VARYING.
                        01  DEPT-NUM        PIC S9(9) COMP.
                        01  EMP-TABLES.
                            05  EMP-NAME    OCCURS 10 TIMES PIC X(10).
                            05  JOB-TITLE   OCCURS 10 TIMES PIC X(10).
                            05  SALARY      OCCURS 10 TIMES COMP-2.
                        01  DONE-FLAG       PIC S9(9) COMP.
                        01  TABLE-SIZE      PIC S9(9) COMP VALUE 10.
                        01  NUM-RET         PIC S9(9) COMP.
                        01  SQLCODE         PIC S9(9) COMP.
                    EXEC SQL END DECLARE SECTION END-EXEC.

                        01  COUNTER         PIC S9(9) COMP.
                        01  DISPLAY-VARIABLES.
                            05  D-EMP-NAME  PIC X(10).
                            05  D-JOB-TITLE PIC X(10).
                            05  D-SALARY    PIC Z(5)9.

                    EXEC SQL INCLUDE SQLCA END-EXEC.

                PROCEDURE DIVISION.

                BEGIN-PGM.
                    EXEC SQL
                        WHENEVER SQLERROR DO PERFORM SQL-ERROR
                    END-EXEC.
                    PERFORM LOGON.
                    PERFORM INIT-TABLES VARYING COUNTER FROM 1 BY 1
                        UNTIL COUNTER > 10.
                    PERFORM GET-DEPT-NUM.
                    PERFORM DISPLAY-HEADER.
                    MOVE ZERO TO DONE-FLAG.
                    MOVE ZERO TO NUM-RET.
                    PERFORM FETCH-BATCH UNTIL DONE-FLAG = 1.
                    PERFORM LOGOFF.

                INIT-TABLES.
```

```
            MOVE SPACE TO EMP-NAME(COUNTER).
            MOVE SPACE TO JOB-TITLE(COUNTER).
            MOVE ZERO TO SALARY(COUNTER).

    GET-DEPT-NUM.
            MOVE ZERO TO DEPT-NUM.
            DISPLAY " ".
            DISPLAY "ENTER DEPARTMENT NUMBER: " WITH NO ADVANCING.
            ACCEPT DEPT-NUM.

    DISPLAY-HEADER.
            DISPLAY " ".
            DISPLAY "EMPLOYEE    JOB TITLE    SALARY".
            DISPLAY "--------    ---------    ------".

    FETCH-BATCH.
            EXEC SQL EXECUTE
                BEGIN
                    CALLDEMO.GET_EMPLOYEES
                        (:DEPT-NUM, :TABLE-SIZE,
                         :NUM-RET,  :DONE-FLAG,
                         :EMP-NAME, :JOB-TITLE, :SALARY);
                END;
            END-EXEC.
            PERFORM PRINT-ROWS VARYING COUNTER FROM 1 BY 1
                UNTIL COUNTER > NUM-RET.

    PRINT-ROWS.
            MOVE EMP-NAME(COUNTER) TO D-EMP-NAME.
            MOVE JOB-TITLE(COUNTER) TO D-JOB-TITLE.
            MOVE SALARY(COUNTER) TO D-SALARY.
            DISPLAY D-EMP-NAME, "   ",
                    D-JOB-TITLE, "    ",
                    D-SALARY.

    LOGON.
            MOVE "SCOTT" TO USERNAME-ARR.
            MOVE 5 TO USERNAME-LEN.
            MOVE "TIGER" TO PASSWD-ARR.
            MOVE 5 TO PASSWD-LEN.
            EXEC SQL
                CONNECT :USERNAME IDENTIFIED BY :PASSWD
            END-EXEC.
            DISPLAY " ".
            DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
```

```
LOGOFF.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL
        COMMIT WORK RELEASE
    END-EXEC.
    STOP RUN.

SQL-ERROR.
    EXEC SQL
        WHENEVER SQLERROR CONTINUE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL
        ROLLBACK WORK RELEASE
    END-EXEC.
    STOP RUN.
```

Remember, the datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Also, before a stored subprogram exits, 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 Oracle8 data dictionary. A database link tells Oracle8 where the remote database is located, the path to it, and what Oracle8 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 4, 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 <Title>Programmer's Guide to the Oracle Call Interface, Volume II: OCI Reference.

You can also use the procedure *describe_procedure* in package DBMS_DESCRIBE, which is supplied with Oracle8. For more information, see the *Oracle8 Application Developer's Guide.*

Using Dynamic PL/SQL

Recall that Pro*COBOL treats 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 12, "Using Dynamic SQL: Advanced Concepts"."

Subprograms Restriction

In dynamic SQL Method 4, a host table cannot be bound to a PL/SQL procedure with a parameter of type "table."

Cursor Variables

Starting with Release 1.7 of Pro*COBOL, you can use *cursor variables* in your Pro*COBOL programs to process multi-row queries using static embedded SQL. A cursor variable identifies a *cursor reference* that is defined and opened on the Oracle7 Server, Release 7.2 or later, using PL/SQL. See the *PL/SQL User's Guide and Reference* for complete information about cursor variables.

Like a cursor, a cursor variable points to the current row in the active set of a multi-row query. Cursors differ from cursor variables the way constants differ from

variables. While a cursor is static, a cursor variable is dynamic, because it is not tied to a specific query. You can open a cursor variable for any type-compatible query.

You can assign new values to a cursor variable and pass it as a parameter to subprograms, including subprograms stored in an Oracle8 database. This gives you a convenient way to centralize data retrieval.

First, you declare the cursor variable. After declaring the variable, you use four statements to control a cursor variable:

- ALLOCATE
- OPEN ... FOR
- FETCH
- CLOSE

After you declare the cursor variable and allocate memory for it, you must pass it as an input host variable (bind variable) to PL/SQL, OPEN it FOR a multi-row query on the server side, FETCH from it on the client side, then CLOSE it on either side.

The advantages of cursor variables are

- *Ease of maintenance*: queries are centralized, in the stored procedure that opens the cursor variable. If you need to change the cursor, you only need to make the change in one place: the stored procedure. There is no need to change each application.
- *Security*: the user of the application (the username when the Pro*COBOL application connected to the database) must have execute permission on the stored procedure that opens the cursor. This user, however, does not need to have read permission on the tables used in the query. This capability can be used to limit access to the columns in the table.

Declaring a Cursor Variable

You declare a Pro*COBOL cursor variable using the SQL-CURSOR pseudotype. For example:

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        ...
        01  CUR-VAR  SQL-CURSOR.
```

```
        ...
    EXEC SQL END DECLARE SECTION END-EXEC.
```

A SQL-CURSOR variable is implemented as a COBOL group item in the code that Pro*COBOL generates. A cursor variable is just like any other Pro*COBOL host variable.

Allocating a Cursor Variable

Before you can OPEN or FETCH from a cursor variable, you must initialize it using the Pro*COBOL ALLOCATE command. For example, to initialize the cursor variable CUR-VAR that was declared in the previous section, write the following statement:

```
    EXEC SQL ALLOCATE :CUR-VAR END-EXEC.
```

Allocating a cursor variable does *not* require a call to the server, either at precompile time or at run time.

> **Warning:** Allocating a cursor variable *does* cause heap memory to be used. For this reason, avoid allocating a cursor variable in a program loop.

Opening a Cursor Variable

You must use an embedded anonymous PL/SQL block to open a cursor variable on the Oracle8 Server. The anonymous PL/SQL block may open the cursor either indirectly by calling a PL/SQL stored procedure that opens the cursor (and defines it in the same statement) or directly from the Pro*COBOL program.

Opening Indirectly through a Stored PL/SQL Procedure

Consider the following PL/SQL package stored in the database:

```
CREATE PACKAGE demo_cur_pkg AS
    TYPE EmpName IS RECORD (name VARCHAR2(10));
    TYPE cur_type IS REF CURSOR RETURN EmpName;
    PROCEDURE open_emp_cur (
            curs    IN OUT curtype,
            dept_num IN     number);
END;

CREATE PACKAGE BODY demo_cur_pkg AS
    CREATE PROCEDURE open_emp_cur (
            curs    IN OUT curtype,
            dept_num IN     number) IS
```

```
        BEGIN
            OPEN curs FOR
                SELECT ename FROM emp
                    WHERE deptno = dept_num
                    ORDER BY ename ASC;
        END;
END;
```

After this package has been stored, you can open the cursor *curs* by calling the *open_emp_cur* stored procedure from your Pro*COBOL program, and FETCH from the cursor variable EMP-CURSOR in the program. For example:

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         01  EMP-CURSOR       SQL-CURSOR.
         01  DEPT-NUM      PIC S9(4).
         01  EMP-NAME      PIC X(10) VARYING.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...

 PROCEDURE DIVISION.
     ...
*    Allocate the cursor variable.
     EXEC SQL
         ALLOCATE :EMP-CURSOR
     END-EXEC.
     ...
     MOVE 30 TO DEPT_NUM.
*    Open the cursor on the Oracle Server.
     EXEC SQL EXECUTE
         BEGIN
             demo_cur_pkg.open_emp_cur(:EMP-CURSOR, :DEPT-NUM);
         END;
     END-EXEC.
     EXEC SQL
         WHENEVER NOT FOUND DO PERFORM SIGN-OFF
     END-EXEC.
 FETCH-LOOP.
     EXEC SQL
         FETCH :EMP-CURSOR INTO :EMP-NAME
     END-EXEC.
     DISPLAY "Employee Name: ",:EMP-NAME.
     GO TO FETCH-LOOP.
     ...
 SIGN-OFF.
```

...

Opening Directly from Your Pro*COBOL Application

To open a cursor using a PL/SQL anonymous block in a Pro*COBOL program, define the cursor in the anonymous block. Consider the following example:

```
PROCEDURE DIVISION.
    ...
    EXEC SQL EXECUTE
        BEGIN
            OPEN :EMP-CURSOR FOR SELECT ENAME FROM EMP
                WHERE deptno = :DEPT-NUM;
        end;
    END-EXEC.
    ...
```

Fetching from a Cursor Variable

After opening a cursor variable for a multi-row query, you use the FETCH statement to retrieve rows from the active set one at a time. The syntax follows:

```
EXEC SQL FETCH cursor_variable_name
    INTO {record_name | variable_name[, variable_name, ...]}
END-EXEC.
```

Each column value returned by the cursor variable is assigned to a corresponding field or variable in the INTO clause, providing their datatypes are compatible.

The FETCH statement must be executed on the client side. In the following example, you fetch rows into a host record named *EMP-REC*:

```
* -- exit loop when done fetching
    EXEC SQL
        WHENEVER NOT FOUND DO PERFORM NO-MORE
    END-EXEC.
    PERFORM
* -- fetch row into record
    EXEC SQL FETCH :EMP-CUR INTO :EMP-REC END-EXEC
* -- test for transfer out of loop
    ...
* -- process the data
    ...
    END-PERFORM.
    ...
```

```
      NO-MORE.
         ...
```

Use the embedded SQL FETCH .... INTO command to retrieve the rows SELECTed when you opened the cursor variable. For example:

```
EXEC SQL
    FETCH :EMP-CURSOR INTO :EMP-INFO:EMP-INFO-IND
END-EXEC.
```

Before you can FETCH from a cursor variable, the variable must be initialized and opened. You cannot FETCH from an unopened cursor variable.

Closing a Cursor Variable

Use the embedded SQL CLOSE statement to close a cursor variable, at which point its active set becomes undefined. The syntax follows:

```
      EXEC SQL CLOSE cursor_variable_name END-EXEC.
```

The CLOSE statement can be executed on the client side or the server side. In the following example, when the last row is processed, you close the cursor variable *CUR-VAR*:

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*    Declare the cursor variable.
         01  CUR-VAR       SQL-CURSOR.
         ...
     EXEC SQL END DECLARE SECTION END-EXEC.

 PROCEDURE DIVISION.
*    Allocate and open the cursor variable, then
*    Fetch one or more rows.
     ...
*    Close the cursor variable.
     EXEC SQL
         CLOSE :CUR-VAR
     END-EXEC.
```

Restrictions

The following restrictions apply to the use of cursor variables:

- Cursor variables are not supported in dynamic SQL.

- You can only use cursor variables with the ALLOCATE, FETCH, and CLOSE commands. The DECLARE CURSOR command does *not* apply to cursor variables.

- You cannot FETCH from a CLOSEd or un-allocated cursor variable.

- If you precompile with CLOSE_ON_COMMIT=NO, it is an error to close a cursor variable that is already closed.

- You cannot use the AT clause with the ALLOCATE command.

Error Conditions

Do *not* perform any of the following operations:

- FETCH from a closed cursor variable

- use a cursor variable that is not ALLOCATEd

- CLOSE a cursor variable that is not open

These operations on cursor variables result in errors.

Sample Programs

The following sample programs — a SQL script (SAMPLE11.SQL) and a Pro*COBOL program (SAMPLE11.PCO) — demonstrate how you can use cursor variables in Pro*COBOL.

SAMPLE11.SQL

Following is the PL/SQL source code for a creating a package that declares and opens a cursor variable:

```
CONNECT SCOTT/TIGER
CREATE OR REPLACE PACKAGE emp_demo_pkg AS
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_cur (
        cursor   IN OUT emp_cur_type,
        dept_num IN     number);
END emp_demo_pkg;
/
CREATE OR REPLACE PACKAGE BODY emp_demo_pkg AS

    PROCEDURE open_cur (
        cursor   IN OUT emp_cur_type,
        dept_num IN     number) IS
```

```
      BEGIN
          OPEN cursor FOR SELECT * FROM emp
          WHERE deptno = dept_num
          ORDER BY ename ASC;
      END;
END emp_demo_pkg;
/
```

SAMPLE11.PCO

Following is a Pro*COBOL sample program that uses the cursor declared in the
SAMPLE11.SQL example to fetch employee names, salaries, and commissions from the
EMP table.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CURSOR-VARIABLES.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC ORACLE OPTION (SQLCHECK=FULL) END-EXEC.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         01  USERNAME          PIC X(15) VARYING.
         01  PASSWD            PIC X(15) VARYING.
         01  HOST              PIC X(15) VARYING.
*    Declare the cursor variable.
         01  EMP-CUR           SQL-CURSOR.

         01  EMP-INFO.
             05  EMP-NUM       PIC S9(4) COMP.
             05  EMP-NAM       PIC X(10) VARYING.
             05  EMP-JOB       PIC X(10) VARYING.
             05  EMP-MGR       PIC S9(4) COMP.
             05  EMP-DAT       PIC X(10) VARYING.
             05  EMP-SAL       PIC S9(6)V99
                               DISPLAY SIGN LEADING SEPARATE.
             05  EMP-COM       PIC S9(6)V99
                               DISPLAY SIGN LEADING SEPARATE.
             05  EMP-DEP       PIC S9(4) COMP.
         01  EMP-INFO-IND.
             05  EMP-NUM-IND   PIC S9(2) COMP.
             05  EMP-NAM-IND   PIC S9(2) COMP.
             05  EMP-JOB-IND   PIC S9(2) COMP.
             05  EMP-MGR-IND   PIC S9(2) COMP.
             05  EMP-DAT-IND   PIC S9(2) COMP.
             05  EMP-SAL-IND   PIC S9(2) COMP.
```

```
                    05  EMP-COM-IND   PIC S9(2) COMP.
                    05  EMP-DEP-IND   PIC S9(2) COMP.

           EXEC SQL END DECLARE SECTION END-EXEC.

           EXEC SQL INCLUDE SQLCA END-EXEC.
                01  DISPLAY-VARIABLES.
                    05  D-DEP-NUM    PIC Z(3)9.
                    05  D-EMP-NAM    PIC X(10).
                    05  D-EMP-SAL    PIC Z(4)9.99.
                    05  D-EMP-COM    PIC Z(4)9.99.


      PROCEDURE DIVISION.
      BEGIN-PGM.
           EXEC SQL
               WHENEVER SQLERROR DO PERFORM SQL-ERROR
           END-EXEC.
           PERFORM LOGON.

  *    Initialize the cursor variable.
           EXEC SQL
               ALLOCATE :EMP-CUR
           END-EXEC.
           DISPLAY "Enter department number (0 to exit):  "
               WITH NO ADVANCING.
           ACCEPT EMP-DEP.
           IF EMP-DEP <= 0
               PERFORM SIGN-OFF
           END-IF.
           MOVE EMP-DEP TO D-DEP-NUM.

  *    Open the cursor by calling a PL/SQL stored procedure.
           EXEC SQL EXECUTE
               BEGIN
                   emp_demo_pkg.open_cur(:EMP-CUR, :EMP-DEP);
               END;
           END-EXEC.
           DISPLAY " ".
           DISPLAY "For department ", D-DEP-NUM, ":".
           DISPLAY " ".
           DISPLAY "EMPLOYEE   SALARY     COMMISSION".
           DISPLAY "---------- ---------- ----------".

      FETCH-LOOP.
```

```
                EXEC SQL
                    WHENEVER NOT FOUND DO PERFORM SIGN-OFF
                END-EXEC.
                MOVE SPACES TO EMP-NAM-ARR.
      *     Fetch data from the cursor into the host variables.
                EXEC SQL FETCH :EMP-CUR
                    INTO :EMP-NUM:EMP-NUM-IND,
                         :EMP-NAM:EMP-NAM-IND,
                         :EMP-JOB:EMP-JOB-IND,
                         :EMP-MGR:EMP-MGR-IND,
                         :EMP-DAT:EMP-DAT-IND,
                         :EMP-SAL:EMP-SAL-IND,
                         :EMP-COM:EMP-COM-IND,
                         :EMP-DEP:EMP-DEP-IND
                END-EXEC.
                MOVE EMP-SAL TO D-EMP-SAL.
                MOVE EMP-COM TO D-EMP-COM.
      *     Check for commission and print results.
                IF EMP-COM-IND = 0
                    DISPLAY EMP-NAM-ARR, "    ", D-EMP-SAL,
                            "    ", D-EMP-COM
                ELSE
                    DISPLAY EMP-NAM-ARR, "    ", D-EMP-SAL,
                            "          N/A"
                END-IF.
                GO TO FETCH-LOOP.

       LOGON.
                MOVE "SCOTT" TO USERNAME-ARR.
                MOVE 5 TO USERNAME-LEN.
                MOVE "TIGER" TO PASSWD-ARR.
                MOVE 5 TO PASSWD-LEN.
                MOVE "INST1_ALIAS" TO HOST-ARR.
                MOVE 11 TO HOST-LEN.
                EXEC SQL
                    CONNECT :USERNAME IDENTIFIED BY :PASSWD
                END-EXEC.
                DISPLAY " ".
                DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

       SIGN-OFF.
      *     Close the cursor variable.
                EXEC SQL
                    CLOSE :EMP-CUR
                END-EXEC.
```

```
        DISPLAY " ".
        DISPLAY "HAVE A GOOD DAY.".
        DISPLAY " ".
        EXEC SQL
            COMMIT WORK RELEASE
        END-EXEC.
        STOP RUN.

    SQL-ERROR.
        EXEC SQL
            WHENEVER SQLERROR CONTINUE
        END-EXEC.
        DISPLAY " ".
        DISPLAY "ORACLE ERROR DETECTED:".
        DISPLAY " ".
        DISPLAY SQLERRMC.
        EXEC SQL
            ROLLBACK WORK RELEASE
        END-EXEC.
        STOP RUN.
```

7

Running the Pro*COBOL Precompiler

This chapter details the requirements for running the Pro*COBOL. You learn

- The Pro*COBOL Command
- What Occurs during Precompilation?
- Precompiler Options
- Entering Options
- Scope of Options
- Quick Reference
- Using Pro*COBOL Options
- Conditional Precompilations
- Separate Precompilations
- Compiling and Linking

The Pro*COBOL Command

To run the Oracle Pro*COBOL Precompiler, you issue the command

```
procob
```

The location of Pro*COBOL differs from system to system. Typically, your system manager or DBA defines environment variables, logicals, or aliases or uses other operating system-specific means to make the Pro*COBOL executable accessible.

The INAME option specifies the source file to be precompiled. For example, the command

```
procob INAME=test
```

precompiles the file *test.pco* in the current directory, since Pro*COBOL assumes that the filename extension is *.pco*. You need not use a file extension when specifying INAME unless the extension is nonstandard.

Input and output filenames need not be accompanied by their respective option names, INAME and ONAME. When the option names are not specified, Pro*COBOL assumes that the first filename specified on the command line is the input filename and that the second filename is the output filename.

Thus, the command

```
procob MODE=ANSI myfile.pco myfile.cob
```

is equivalent to

```
procob MODE=ANSI INAME=myfile.pco ONAME=myfile.cob
```

> **Note:** Option names and option values that do not name specific operating system objects, such as filenames, are not case-sensitive. In the examples in this guide, option names are written in upper case, and option values are usually in lower case. Filenames, including the name of the Pro*COBOL executable itself, always follow the case conventions used by the operating system on which it is executed.

What Occurs during Precompilation?

During precompilation, Pro*COBOL generates COBOL code that replaces the SQL statements embedded in your host program. The generated code includes data structures that contain the datatype, length, and address of each host variable, as well as other information required by the Oracle runtime library, SQLLIB. The gen-

erated code also contains the calls to SQLLIB routines that perform the embedded SQL operations.

> **Note:** Pro*COBOL does *not* generate calls to Oracle Call Interface (OCI) routines.

Pro*COBOL can issue warnings and error messages. These messages have the prefix PCC-, and are described in *Oracle8 Error Messages.*

Precompiler Options

Many useful options are available at precompile time. They let you control how resources are used, how errors are reported, how input and output are formatted, and how cursors are managed. To specify a precompiler option, use the following syntax:

```
<option_name>=<value>
```

The *value* of an option is a string literal, which represents text or numeric values. For example, for the option

```
... INAME=my_test
```

the value is a string literal that specifies a filename, but for the option

```
... MAXOPENCURSORS=20
```

the value is numeric.

Some options take Boolean values, which you can represent with the strings YES or NO, TRUE or FALSE, or with the integer literals 1 or 0, respectively. For example, the option

```
... SELECT_ERROR=YES
```

is equivalent to

```
... SELECT_ERROR=TRUE
```

or

```
... SELECT_ERROR=1
```

The option value is always separated from the option name by an equal sign, leave no whitespace around the equal sign, because spaces delimit individual options. For example, you might specify the option AUTO_CONNECT on the command line as follows:

```
...  AUTO_CONNECT=YES
```

You can abbreviate the names of options if the abbreviation is unambiguous. For example, you cannot use the abbreviation MAX because it might stand for MAX-LITERAL or MAXOPENCURSORS.

A handy reference to the Pro*COBOL options is available online. To see the online display, enter the Pro*COBOL command, with no arguments, at your operating system prompt:

```
procob
```

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.

Precedence of Option Values

Option values are determined by the following, in order of increasing precedence:

- a default built in to Pro*COBOL
- a value set in the *system* configuration file
- a value set in a *user* configuration file
- a value entered in the command line
- a value set in an inline specification

For example, the option MAXOPENCURSORS specifies the maximum number of cached open cursors. The built-in Pro*COBOL default value for this option is 10. However, if MAXOPENCURSORS=32 is specified in the system configuration file, the value becomes 32. The user configuration file could set it to yet another value, which then overrides the system configuration value.

Then, if this option is set on the command line, the new command-line value takes precedence. Finally, an inline specification takes precedence over all preceding defaults. For more information, see "Configuration Files" on page 7-6 and "Entering Options" on page 7-7.

Macro and Micro Options

Pro*COBOL has two options, DBMS and MODE, that existed before release 8.0, and that also control several functions at once. These are known as *macro* options. Some newer options, such as END_OF_FETCH, control only one function and are known as *micro* options. When setting a macro and a micro option, you must

remember that macro options have precedence over micro options, if, and only if, the macro option is at a higher level of precedence than the micro option, as listed in the section "Precedence of Option Values" on page 7-4. This behavior is a change from releases of Pro*COBOL prior to 8.0.

For example, the default for MODE is ORACLE, and for END_OF_FETCH is 1403. If you specify MODE=ANSI in the user configuration file, Pro*COBOL will return a value of 100 at the end of fetch, overriding the default END_OF_FETCH value of 1403. If you specify both MODE=ANSI and END_OF_FETCH=1403 in the configuration file, then 1403 will be returned. If you specify MODE=ANSI in your configuration file and END_OF_FETCH=1403 on the command line, 1403 will be returned.

The following table lists the values of micro options set by the macro option values:

Table 7–1 How Macro Option Values Set Micro Oprion Values

Macro Option	Micro Option
MODE=ANSI	ISO	CLOSE_ON_COMMIT=YES
	DECLARE_SECTION=YES
	END_OF_FETCH=100
MODE=ANSI14	ANSI13	ISO14	ISO13	CLOSE_ON_COMMIT=NO
	DECLARE_SECTION=YES
	END_OF_FETCH=100
MODE=ORACLE	CLOSE_ON_COMMIT=NO
	DECLARE_SECTION=NO
	END_OF_FETCH=1403
DBMS=V6	UNSAFE_NULL=YES
DBMS=NATIVE	V7	V8	UNSAFE_NULL=NO

Determining Current Values

You can interactively determine the current value for one or more options by using a question mark on the command line. For example, if you issue the command

```
procob?
```

the complete option set, along with current values, is displayed on your terminal. In this case, the values are those built into Pro*COBOL, overridden by any values in the system configuration file. But if you issue the following command

```
procob CONFIG=my_config_file.cfg?
```

and there is a file named *my_config_file.cfg* in the current directory, the options from the *my_config_file.cfg* file are listed with the other default values. Values in the user configuration file supply missing values, and they supersede values built into Pro*COBOL and values specified in the system configuration file.

You can also determine the current value of a single option by simply specifying the option name followed by "=?" as in

```
procob MAXOPENCURSORS=?
```

> **Note:** With some operating systems and user shells, such as UNIX C shell, the "?" may need to be preceded by an "escape" character, such as a back-slash (\). For example, instead of "procob?," you might need to use "procob \?" to list the Pro*COBOL option settings.

Case Sensitivity

In general, you can use either uppercase or lowercase for command-line option names and values. However, if your operating system is case-sensitive, (UNIX for example) you must specify filename values, including the name of Pro*COBOL executable, using the correct combination of upper and lowercase letters.

Configuration Files

A configuration file is a text file that contains precompiler options. Each record (line) in the file contains one option, with its associated value or values. For example, a configuration file might contain the lines

```
FIPS=YES
MODE=ANSI
```

to set values for the FIPS and MODE options.

There is a single system configuration file for each system. The name of the system configuration file is

```
pcbcfg.cfg
```

The location of the file is operating system-specific. On most UNIX systems, the Pro*COBOL configuration file is usually located in the *$ORACLE_HOME/pre-*

comp/admin directory, where *$ORACLE_HOME* is the environment variable for the database software.

Note that before release 8.0 of Pro*COBOL, the configuration file was called *pccob.cfg*.

The Pro*COBOL user can have one or more user configuration files. The name of the configuration file must be specified using the CONFIG command-line option. For more information, see "Determining Current Values" on page 7-5.

Note: You cannot nest configuration files. This means that CONFIG is not a valid option inside a configuration file.

Entering Options

All Pro*COBOL options can be entered on the command line or (except CONFIG) from a configuration file. Many options can also be entered inline. During a given run, Pro*COBOL can accept options from all three sources.

On the Command Line

You enter precompiler options on the command line using the following syntax:

```
... [option_name=value] [option_name=value] ...
```

Separate each option with one or more spaces. For example, you might enter the following options:

```
... ERRORS=no LTYPE=short
```

Inline

You enter options inline by coding EXEC ORACLE statements, using the following syntax:

```
EXEC ORACLE OPTION (option_name=value) END-EXEC.
```

For example, you might code the following statement:

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.
```

An option entered inline overrides the same option entered on the command line.

Advantages

The EXEC ORACLE feature is especially useful for changing option values during precompilation. For example, you might want to change the HOLD_CURSOR and RELEASE_CURSOR values on a statement-by-statement basis. Appendix D shows you how to use inline options to optimize runtime performance.

Specifying options inline is also helpful if your operating system limits the number of characters you can enter on the command line, and you can store inline options in configuration files, which are discussed in the next section.

Scope of EXEC ORACLE

An EXEC ORACLE statement stays in effect until textually 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 END-EXEC.
    01 EMP-NAME    PIC X(20) VARYING.
    01 EMP-NUMBER  PIC S9(4) COMP VALUE ZERO.
    01 SALARY      PIC S9(5)V99 COMP-3 VALUE ZERO.
    01 DEPT-NUMBER PIC S9(4) COMP VALUE ZERO.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL WHENEVER NOT FOUND GOTO NO-MORE END-EXEC.
...
EXEC ORACLE OPTION (HOLD_CURSOR=NO)END-EXEC.
...
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
     SELECT EMPNO, DEPTNO FROM EMP
END-EXEC.
EXEC SQL OPEN EMP-CURSOR END-EXEC.

DISPLAY 'Employee Number  Dept'.
DISPLAY '--------------  ----'.
PERFORM
    EXEC SQL
      FETCH EMP-CURSOR INTO :EMP-NUMBER, :DEPT-NUMBER
    END-EXEC
    DISPLAY EMP-NUMBER, DEPT-NUMBER END-EXEC
END-PERFORM.

NO-MORE.
     EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
PERFORM
    DISPLAY 'Employee number? '
```

```
      ACCEPT EMP-NUMBER
      IF EMP-NUMBER IS NOT = 0
          EXEC ORACLE OPTION (HOLD_CURSOR=YES) END-EXEC
          EXEC SQL SELECT ENAME, SAL
              INTO :EMP-NAME, :SALARY
              FROM EMP
              WHERE EMPNO = :EMP-NUMBER
              DISPLAY 'Salary for ', EMP-NAME, ' is ', SALARY
          END-EXEC
      END-IF
   END-PERFORM.
NEXT-PARA.
   ...
```

Scope of Options

A precompilation unit is a file containing COBOL code and one or more embedded SQL statements. 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=YES and RELEASE_CURSOR=YES for unit A but not unit B, SQL statements in unit A run with these HOLD_CURSOR and RELEASE_CURSOR values, but SQL statements in unit B run with the default values. However, the MAXOPENCURSORS setting that is in effect when you connect to Oracle stays in effect for the life of that connection.

The scope of an inline option is positional, not logical. That is, an inline option affects SQL statements that follow it in the source file, not in the flow of program logic. An option setting stays in effect until the end-of-file unless you re-specify the option.

Quick Reference

Table 7–2 is a quick reference to the Pro*COBOL options. Options marked with an asterisk can be entered inline.

Another handy reference is available online. To see the online display, just enter the Pro*COBOL command without options at your operating system prompt. The display gives the name, syntax, default value, and purpose of each option.

Note: There are some platform-specific options. For example, on byte-swapped platforms the option COMP5 governs the use of certain COMPUTATIONAL items. Check your system-specific Oracle manuals.

Table 7–2   Option List

Syntax	Default	Specifies ...
ASACC={YES|NO}	NO	If YES, use ASA carriage control for listing
ASSUME_SQLCODE={YES|NO}	NO	If YES, assume SQLCODE variable exists
AUTO_CONNECT={YES|NO}	NO	If YES, allow automatic connect to ops$ accounts
CLOSE_ON_COMMIT	NO	If YES, close all cursors on COMMIT
CONFIG=*filename*		Specifies name of user-defined configuration file
DATE_FORMAT	LOCAL	Specifies date string format
DBMS={NATIVE|V7|V8}	NATIVE	Version-specific behavior of Oracle at precompile time
DECLARE_SECTION	NO	If YES, DECLARE SECTION is required.
DEFINE=*symbol* *		Define a symbol used in conditional precompilation
END_OF_FETCH	1403	End-of-fetch SQLCODE value
ERRORS={YES|NO} *	YES	If YES, display errors on the terminal
FIPS={YES|NO}	NO	If YES, ANSI/ISO extensions are flagged
FORMAT={ANSI|TERMINAL}	ANSI	Format of input file COBOL statements
HOLD_CURSOR={YES|NO}*	NO	If YES, hold OraCursor (do not re-assign)
HOST={COBOL|COB74}	COBOL	COBOL version used in input file (COBOL 85 or COBOL 74)
[INAME=]*filename*		Name of input file
INCLUDE=*path**		Pathname for EXEC SQL INCLUDE files
IRECLEN=*integer*	80	Record length of input file
LITDELIM={APOST|QUOTE}	QUOTE	Delimiters for COBOL strings
LNAME=*filename*		Name of listing file
LRECLEN=*integer*	132	Record length of listing file
LTYPE={LONG|SHORT|NONE} *	LONG	Type of listing

Table 7–2   Option List

Syntax	Default	Specifies ...
MAXLITERAL=*integer* *	platform-specific	Maximum length of strings
MAXOPENCURSORS=*integer* *	10	Maximum number of OraCursors cached (1)
MODE={ORA-CLE | ANSI | ANSI14 | ANSI13}	ORACLE	If ANSI, follow the ANSI/ISO SQL standard
NLS_LOCAL={YES | NO}	NO	If YES, use NCHAR semantics of previous Pro*COBOL releases
[ONAME=]*filename*		Name of output file
ORACA={YES | NO}*	NO	If YES, use ORACA communications area
ORECLEN=*integer*	80	Record length of output file
PAGELEN=*integer*	66	Lines per page in listing
PICX	CHARF	Datatype of PIC X COBOL variables
RELEASE_CURSOR={YES | NO} *	NO	If YES, release OraCursor after execute
SELECT_ERROR={YES | NO}*	YES	If YES, generate FOUND error on SELECT
SQLCHECK={FULL | SYNTAX | LIM-ITED | NONE}*	SYNTAX	SQL checking level
UNSAFE_NULL={YES | NO}	NO	If YES, unsafe null fetches are allowed (disables the ORA-01405 message)
USERID=*username/ password*		Oracle username and password
VARCHAR={YES | NO}	NO	If YES, accept user-defined VARCHAR group items
XREF={YES | NO}*	YES	If YES, generate symbol cross references in listing

Using Pro*COBOL Options

This section is organized for easy reference. It lists Pro*COBOL options alphabetically, and for each option gives its purpose, syntax, and default value. Usage notes that help you understand how the option works are also provided. Unless the usage notes say otherwise, the option can be entered on the command line, inline, or from a configuration file.

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
Cannot be entered inline.

ASSUME_SQLCODE

Purpose
Instructs Pro*COBOL to presume that SQLCODE is declared whether or not it is declared in the program, or of the proper type.

Syntax
ASSUME_SQLCODE={YES | NO}

Default
NO

Usage Notes
Cannot be entered inline.

When DECLARE_SECTION=YES and ASSUME_SQLCODE=YES, SQLCODE can be declared outside a Declare Section.

When DECLARE_SECTION=YES and ASSUME_SQLCODE=NO, SQLCODE is recognized as the status variable if and only if at least one of the following criteria is satisfied:

- It is declared with *exactly* the right datatype.

- Pro*COBOL finds no other status variable. If Pro*COBOL finds a SQLSTATE declaration (of *exactly* the right type of course), or finds an include of a SQLCA, then it will *not* presume SQLCODE is declared.

When ASSUME_SQLCODE=YES,and when SQLSTATE and/or SQLCA are declared as status variables, Pro*COBOL presumes SQLCODE is declared whether or not it is declared or of the proper type.

AUTO_CONNECT

Purpose
Specifies whether your program connects automatically to the default user account.

Syntax
AUTO_CONNECT={YES | NO}

Default
NO

Usage Notes
Cannot be entered inline.

When AUTO_CONNECT=YES, as soon as Pro*COBOL encounters an executable SQL statement, your program tries to log on to Oracle automatically with the userid

```
<prefix><username>
```

where *prefix* is the value of the Oracle initialization parameter OS_AUTHENT_PREFIX (the default value is OPS$) and *username* is your operating system user or task name. In this case, you cannot override the default value for MAXOPENCURORS (10), even if you specify a different value on the command line.

When AUTO_CONNECT=NO (the default), you must use the CONNECT statement to logon to Oracle.

CLOSE_ON_COMMIT

Purpose
Specifies whether or not all cursors declared without the WITH HOLD clause are closed on commit.

Syntax
CLOSE_ON_COMMIT={YES | NO}

Default
NO

Usage Notes
Can be used only on the command line or in a configuration file.

This option will only have an effect when a cursor is not coded using the WITH HOLD clause in a DECLARE CURSOR statement, since that will override both the new option and the existing behavior which is associated with the MODE option. If MODE is specified at a higher level than CLOSE_ON_COMMIT, then MODE takes precedence. For example, the defaults are MODE=ORACLE and CLOSE_ON_COMMIT=NO. If the user specifies MODE=ANSI on the command line, then any cursors not using the WITH HOLD clause will be closed on commit.

Issuing a COMMIT or ROLLBACK closes all explicit cursors. (When MODE={ANSI13|ORACLE}, a commit or rollback closes only cursors referenced in a CURRENT OF clause.)

For a further discussion of the precedence of this option see "Macro and Micro Options" on page 7-4.

CONFIG

Purpose
Specifies the name of a user configuration file.

Syntax
CONFIG=*filename*

Default

None

Usage Notes

Can be entered only on the command line.

Pro*COBOL can use a configuration file containing preset command-line options. However, you can specify any of several alternative files, called *user configuration files*. For more information, see "Configuration Files" on page 7-6.

You cannot nest configuration files. Therefore, you cannot specify the option CON-FIG in a configuration file.

DATE_FORMAT

Purpose

Species the string format in which dates are returned.

Syntax

DATE_FORMAT={ISO | USA | EUR | JIS | LOCAL | '*fmt*' (default LOCAL)

Default

LOCAL

Usage Notes

Can only be entered on the command line or in a configuration file. The date strings are shown in the following table:

Table 7–3   Formats for Date Strings

Format Name	Abbreviation	Date Format
International Standards Organization	ISO	yyyy-mm-dd
USA standard	USA	mm/dd/yyyy
European standard	EUR	dd.mm.yyyy
Japanese Industrial Standard	JIS	yyyy-mm-dd

Table 7–3   Formats for Date Strings

Format Name	Abbreviation	Date Format
installation-defined	LOCAL	Any installation-defined form.

'*fmt*' is a date format model, such as 'Month dd, yyyy'. See the *Oracle8 SQL Reference* for the list of date format model elements.

There is one restriction on the use of the DATE_FORMAT option: All compilation units to be linked together must use the same DATE_FORMAT value. An error occurs when there is a mismatch in the values of DATE_FORMAT across compilation units

DBMS

Purpose
Specifies whether Oracle follows the semantic and syntactic rules of Oracle7, Oracle8, or the native version of Oracle (that is, the version to which your application is connected).

Syntax
DBMS={NATIVE | V7 | V8}

Default
NATIVE

Usage Notes
Cannot be entered inline.

With the DBMS option 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.

Table 7–4 shows how the compatible DBMS and MODE settings interact. All other combinations are incompatible or not recommended.

Table 7–4   How DBMS and MODE Interact

Situation	DBMS=V7 or V8 MODE=ANSI	DBMS=V7 or V8 MODE=ORACLE
fetch truncated values without using indicator variables	no error but SQLWARN(2) is set	no error but SQLWARN(2) is set
open an already OPENed cursor	error -2117	no error
close an already CLOSEd cursor	error -2114	no error
SQL group function ignores nulls	no warning	no warning
when SQL group function in multi-row query is called	FETCH time	FETCH time
declare SQLCA structure	optional	required
declare SQLCODE or SQLSTATE status variable	required	optional but Oracle ignores
external datatype code DESCRIBE returns (dynamic SQL Method 4)	96	96
integrity constraints	enabled	enabled
PCTINCREASE for rollback segments	not allowed	not allowed
MAXEXTENTS storage parameters	not allowed	not allowed

Notes:

1. Includes ANSI13.

2. Includes ANSI14 and ANSI13.

DECLARE_SECTION

Purpose
Specifies whether or not *only* declarations in a Declare Section are allowed as host variables.

Syntax
DECLARE_SECTION={YES | NO}

Default
NO

Usage Notes
Can be entered only on the command line or in a configuration file.

When MODE=ORACLE, use of the BEGIN DECLARE SECTION and END DECLARE SECTION statements are optional, starting with release 8.0 of Pro*COBOL. The DECLARE_SECTION option is provided for backward compatibility with previous releases. DECLARE_SECTION is a micro option of MODE. For a discussion of precedence of this option, see "Macro and Micro Options" on page 7-4.

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 "Conditional Precompilations" on page 7-39.

Syntax
DEFINE=*symbol*

Default
None

Usage Notes
If you enter DEFINE inline, the EXEC ORACLE statement takes the following form:

```
EXEC ORACLE DEFINE <symbol> END-EXEC.
```

END_OF_FETCH

Purpose
Specifies which SQLCODE value is returned when an END-OF-FETCH condition occurs after execution of a SQL statement.

Syntax
END_OF_FETCH={100 | 1403}

Default
1403

Usage Notes
Can be entered only on the command line or in a configuration file.

If you specify MODE=ANSI in a configuration file, Pro*COBOL returns the SQL-CODE value 100 at the END_OF_FETCH, overriding the default END_OF_FETCH=1403. If you specify MODE=ANSI and END_OF_FETCH=1403 in the configuration file, then Pro*COBOL will return the SQLCODE value 1403 at the END_OF_FETCH. If you specify MODE=ANSI in the configuration file and END_OF_FETCH=1403 on the command line, Pro*COBOL will again return the SQLCODE value 1403 at the END_OF_FETCH.

END_OF_FETCH is a micro option of MODE. For further discussion, see "Macro and Micro Options" on page 7-4.

ERRORS

Purpose
Specifies whether Pro*COBOL error messages are sent to the terminal and listing file or only to the listing file.

Syntax
ERRORS={YES | NO}

Default
YES

Usage Notes

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/ISO SQL are flagged (by the FIPS Flagger). An extension is any SQL element that violates ANSI/ISO format or syntax rules, except privilege enforcement rules.

Syntax

FIPS={YES | NO}

Default

NO

Usage Notes

When FIPS=YES, the FIPS Flagger issues warning (not error) messages if you use an Oracle extension to the ANSI/ISO embedded SQL standard (SQL92) or use a SQL92 feature in a nonconforming manner.

The following extensions to ANSI/ISO SQL are flagged at precompile 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, NUMBER, RAW, LONG RAW, VARRAW, ROWID, and VAR-CHAR datatypes
- ORACLE OPTION statement for specifying runtime options
- EXEC IAF and EXEC TOOLS 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 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...DO labels and of host variables in the INTO clause

FORMAT

Purpose
Specifies the format of COBOL input lines.

Syntax
FORMAT={ANSI | TERMINAL}

Default
ANSI

Usage Notes
Cannot be entered inline.

The format of input lines is system-dependent. Check your system-specific Oracle manuals.

When FORMAT=ANSI, the format of input lines conforms as much as possible to the current ANSI standard for COBOL. When FORMAT=TERMINAL, input lines start with column 7. Example code in this book is in TERMINAL format. See "Coding Area" on page 3-2 for a more complete description.

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
You can use HOLD_CURSOR to improve the performance of your program. For more information, see Appendix D.

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, Pro*COBOL 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 and RELEASE_CURSOR=NO, the link is maintained; Pro*COBOL 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 re-parse 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 that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For information showing how these two options interact, see Table 4–11.

HOST

Purpose
Specifies the host language to be used.

Syntax
HOST={COB74 | COBOL}

Default
COBOL

Usage Notes
Cannot be entered inline.

COB74 refers to the 1974 version of ANSI-approved COBOL. COBOL refers to the 1985 version. Other values might be available on your platform.

INAME

Purpose
Specifies the name of the input file.

Syntax
INAME=*filename*

Default
None

Usage Notes
Cannot be entered inline.

When specifying the name of your input file on the command line, the keyword INAME is optional. For example, in Pro*COBOL, you can specify *myprog.pco* instead of INAME=*myprog.pco.*

Pro*COBOL assumes the standard input file extension, *pco.* So, you need not use a file extension when specifying INAME unless the extension is nonstandard. For Pro*COBOL, if you use a nonstandard 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
Typically, you use INCLUDE to specify a directory path for the SQLCA and ORACA files. Pro*COBOL 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> ...
```

Pro*COBOL 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, Pro*COBOL 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. Follow the conventions of your operating system.

IRECLEN

Purpose
Specifies the record length of the input file.

Syntax
IRECLEN=*integer*

Default
80

Usage Notes
Cannot be entered inline.

The value you specify for IRECLEN should not exceed the value of ORECLEN. The maximum value allowed is system-dependent.

LITDELIM

Purpose
The LITDELIM option specifies the delimiters for string constants and literals in the COBOL code generated by Pro*COBOL.

Syntax
LITDELIM={APOST | QUOTE}

Default
QUOTE

Usage Notes
When LITDELIM=APOST, Pro*COBOL uses apostrophes when generating COBOL code. If you specify LITDELIM=QUOTE, quotation marks are used, as in

```
CALL "SQLROL" USING SQL-TMP0.
```

In SQL statements, you must use quotation marks to delimit identifiers containing special or lowercase characters, as in

```
EXEC SQL CREATE TABLE "Emp2" END-EXEC.
```

but you must use apostrophes to delimit string constants, as in

```
EXEC SQL SELECT ENAME FROM EMP WHERE JOB = 'CLERK' END-EXEC.
```

Regardless of which delimiters are used in the Pro*COBOL source file, Pro*COBOL generates the delimiters specified by the LITDELIM value.

LNAME

Purpose
Specifies a non-default name for the listing file.

Syntax
LNAME=*filename*

Default
input.LIS, where *input* is the base name of the input file.

Usage Notes
Cannot be entered inline.

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
Cannot be entered inline.

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
Cannot be entered inline.

Table 7–5   Listing Types

LTYPE=LONG	input lines appear in the listing file.
LTYPE=SHORT	input lines do *not* appear in the listing file.
LTYPE=NONE	no listing file is created.

MAXLITERAL

Purpose
Specifies the maximum length of string literals generated by Pro*COBOL so that compiler limits are not exceeded. For example, if your compiler cannot handle string literals longer than 132 characters, you can specify MAXLITERAL=132 on the command line.

Syntax
MAXLITERAL=*integer*

Default
The default is 256.

Usage Notes

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, some COBOL compilers cannot handle string literals longer than 132 characters, so you would specify MAXLITERAL=132.

Strings that exceed the length specified by MAXLITERAL are divided during precompilation, then recombined (concatenated) at run time.

You can enter MAXLITERAL inline but your program can set its value just once, and the EXEC ORACLE statement must precede the first EXEC SQL statement. Otherwise, Pro*COBOL issues a warning message, ignores the extra or misplaced EXEC ORACLE statement, and continues processing.

MAXOPENCURSORS

Purpose

Specifies the number of concurrently open cursors that Pro*COBOL tries to keep cached.

Syntax

MAXOPENCURSORS=*integer*

Default

10

Usage Notes

You can use MAXOPENCURSORS to improve the performance of your program. For more information, see Appendix D.

When precompiling separately, use MAXOPENCURSORS as described in "Separate Precompilations" on page 7-41.

MAXOPENCURSORS specifies the *initial* size of the SQLLIB 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. To avoid a "maximum open cursors exceeded" Oracle error, MAXOPENCURSORS must be lower than OPEN_CURSORS by at least 6.

As your program's need for concurrently open cursors grows, you might want to re-specify 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
Cannot be entered inline.

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:

- You cannot OPEN a cursor that is already open or CLOSE a cursor that is already closed. (When MODE=ORACLE, you can reOPEN an open cursor to avoid re-parsing.).

- No error message is issued if Oracle assigns a truncated column value to an output host variable.

When MODE={ANSI | ANSI14}, a 4-byte integer variable named SQLCODE or a 5-byte character variable named SQLSTATE must be declared. For more information, see "Error Handling Alternatives" on page 9-2.

Table 7–4   shows how the MODE and DBMS settings interact. Other combinations are incompatible or are not recommended.

NLS_LOCAL

Purpose
The NLS_LOCAL option determines whether NLS character conversions are performed by the Pro*COBOL runtime library or by the Oracle Server.

Syntax
NLS_LOCAL={YES | NO}

Default
NO

Usage Notes
Cannot be entered inline.

When NLS_LOCAL=YES, the runtime library (SQLLIB) locally performs blank-padding and blank-stripping for host variables that have multi-byte NLS datatypes. Continue to use this value only for Pro*COBOL applications written for previous releases that have not been updated for Oracle8.

When NLS_LOCAL=NO, blank-padding and blank-stripping operations are performed by the Oracle Server for host variables that have multi-byte NLS datatypes. Use for all new Oracle8, or later, applications.

ONAME

Purpose
Specifies the name of the output file.

Syntax
ONAME=*filename*

Default

System-dependent

Usage Notes

Cannot be entered inline.

Use this option to specify the name of the output file, where the name differs from that of the input file. For example, if you issue

```
procob INAME=my_test
```

the default output filename is *my_test.cob*. If you want the output filename to be *my_test_1.cob*, issue the command

```
procob INAME=my_test ONAME=my_test_1.cob
```

Note that you should add the *.cob* extension to files specified using ONAME. There is no default extension with the ONAME option.

> **Attention:** Oracle recommends that you not let the output filename default, but rather name it explicitly using ONAME.

ORACA

Purpose

Specifies whether a program can use the Oracle Communications Area (ORACA).

Syntax ORACA={YES | NO}

Default

NO

Usage Notes

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
Cannot be entered inline.

The value you specify for ORECLEN should equal or exceed the value of IRE-CLEN. 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
Cannot be entered inline. The maximum value allowed is system-dependent.

PICX

Purpose
Specifies the default datatype of PIC X variables.

Syntax
PICX={CHARF | VARCHAR2}

Default
CHARF

Usage Notes

Can be entered only on the command line or in a configuration file.

Starting in Pro*COBOL 8.0, the default datatype of PIC X, N, or G variables is changed from VARCHAR2 to CHARF. PICX is provided for backward compatibility.

This new default behavior is consistent with the normal COBOL move semantics. Note that this is a change in behavior for the case where you are inserting a PIC X variable (with MODE=ORACLE) into a VARCHAR2 column. Any trailing blanks which had formerly been trimmed will be preserved. Note also, that the new default lessens the occurrence of the following anomaly: Using a PIC X bind variable initialized with trailing blanks in a WHERE clause would never match a value with the same number of trailing blanks which was stored in a char column because the bind variable's trailing blanks were stripped before the comparison.

When PICX=VARCHAR2, Oracle treats local CHAR variables in a PL/SQL block like variable-length character values. When PICX=CHARF, however, Oracle treats the CHAR variables like ANSI-compliant, fixed-length character values.

RELEASE_CURSOR

Purpose

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

Syntax

RELEASE_CURSOR={YES|NO}

Default

NO

Usage Notes

You can use RELEASE_CURSOR to improve the performance of your program. For more information, see Appendix D.

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, Pro*COBOL 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. Pro*COBOL 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 re-parse 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 that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For information showing how these two options interact, see Table 4–11.

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
When SELECT_ERROR=YES, an error is generated if a single-row select returns too many rows or an array select returns more rows than the host array can accommodate.

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. To ensure a specific ordering of rows, 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 if you are selecting into an array. When SELECT_ERROR=YES, whether or not you use ORDER BY, an error is generated if 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
The values SEMANTICS and FULL are equivalent, as are the values SYNTAX and LIMITED.

Pro*COBOL can help you debug a program by checking the syntax and semantics of embedded SQL statements and PL/SQL blocks. Any errors found are reported at precompile time.

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=NONE on the command line, you cannot specify SQLCHECK=SYNTAX inline.

If SQLCHECK=SYNTAX | SEMANTICS, Pro*COBOL generates an error when PL/SQL reserved words are used in SQL statements, even though the SQL statements are not themselves PL/SQL. If a PL/SQL reserved word must be used as an identifier, you can enclose it in double-quotes.

When SQLCHECK=SEMANTICS, Pro*COBOL checks the syntax and semantics of

- data manipulation statements such as INSERT and UPDATE
- PL/SQL blocks

However, Pro*COBOL checks only the syntax of remote data manipulation statements (those using the AT *db_name* clause).

Pro*COBOL gets the information for a semantic check from embedded DECLARE TABLE statements or, if you specify the option USERID, 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 information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. During precompilation, a DECLARE TABLE definition overrides a data dictionary definition if they conflict.

Specify SQLCHECK=SEMANTICS when precompiling new programs. If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK=SEMANTICS and the option USERID.

When SQLCHECK=SYNTAX, Pro*COBOL checks the syntax of

- data manipulation statements
- PL/SQL blocks

No semantic checking is done. DECLARE TABLE statements are ignored and PL/SQL blocks are not allowed. When checking data manipulation statements, Pro*COBOL uses Oracle8 syntax rules, which are downwardly compatible. Specify SQLCHECK=SYNTAX when migrating your precompiled programs.

When SQLCHECK=NONE, no syntactic or semantic checking is done. DECLARE TABLE statements are ignored and PL/SQL blocks are not allowed. Specify SQLCHECK=NONE if your program

- contains non-Oracle SQL (for example, because it will connect to a non-Oracle server via Open Gateway)
- references tables not yet created and lacks DECLARE TABLE statements for them

Table 7–6 summarizes the checking done by SQLCHECK. For more information about syntactic and semantic checking, see Appendix E, "Syntactic and Semantic Checking".

Table 7–6   Checking Done by SQLCHECK

	SQLCHECK=SEMANTICS		SQLCHECK=SYNTAX		SQLCHECK=NONE	
	Syntax	Semantics	Syntax	Semantics	Syntax	Semantics
DML	X	X	X			
Remote DML	X		X			
PL/SQL	X	X				

UNSAFE_NULL

Purpose
Specifying UNSAFE_NULL=YES prevents generation of ORA-01405 messages when fetching NULLs without using indicator variables.

Syntax
UNSAFE_NULL={YES | NO}

Default
NO

Usage Notes
Cannot be entered inline.

The UNSAFE_NULL=YES is allowed only when MODE=ORACLE and DBMS=V7 or V8.

The UNSAFE_NULL option has no effect on host variables in an embedded PL/SQL block. You *must* use indicator variables to avoid ORA-01405 errors.

When UNSAFE_NULL=YES, no error is returned if a SELECT or FETCH statement selects a null, and there is no indicator variable associated with the output host variable. When UNSAFE_NULL=NO, SELECTing or FETCHing a null column or expression into a host variable that has no associated indicator variable causes an error (SQLSTATE is "22002"; SQLCODE is ORA-01405).

USERID

Purpose
Specifies an Oracle username and password.

Syntax
USERID=*username/password*

Default
None

Usage Notes
Cannot be entered inline.

Do not specify this option when using the automatic logon feature, which accepts your Oracle username prefixed with the value of the Oracle initialization parameter OS_AUTHENT_PREFIX.

When SQLCHECK=SEMANTICS, if you want Pro*COBOL to get needed information by connecting to Oracle and accessing the data dictionary, you must also specify USERID.

VARCHAR

Purpose
The VARCHAR option instructs Pro*COBOL to treat the COBOL group item described in Chapter 5, "Using Embedded SQL" as a VARCHAR datatype.

Syntax
VARCHAR={YES | NO}

Default
NO

Usage Notes
Cannot be entered inline.

When VARCHAR=YES, the implicit group item described in Chapter 5, "Using Embedded SQL" is accepted as an Oracle8 VARCHAR external datatype with a length field and a string field.

When VARCHAR=NO, Pro*COBOL does not accept the implicit group items as VARCHAR external datatypes.

XREF

Purpose
Specifies whether a cross-reference section is included in the listing file.

Syntax
XREF={YES | NO}

Default
YES

Usage Notes
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.

Conditional Precompilations

Conditional precompilation includes (or excludes) sections of code in your host pro-gram based on certain conditions. For example, you might want to include one sec-tion 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:

```
*    -- define a symbol
     EXEC ORACLE DEFINE symbol
*    -- if symbol is defined
```

```
      EXEC ORACLE IFDEF symbol
*   -- if symbol is not defined
      EXEC ORACLE IFNDEF symbol
*           -- otherwise
      EXEC ORACLE ELSE
*           -- end this control block
      EXEC ORACLE ENDIF
```

All EXEC ORACLE statements must be terminated with the statement terminator for your host language. For example, in Pro*COBOL, a conditional statement must be terminated with *END-EXEC.*

An Example

In the following example, the SELECT statement is precompiled only when the symbol *SITE2* is defined:

```
EXEC ORACLE IFDEF SITE2 END-EXEC.
EXEC SQL SELECT DNAME
    INTO :DEPT-NAME
    FROM DEPT
    WHERE DEPTNO = :DEPT-NUMBER
EXEC ORACLE ENDIF END-EXEC.
```

Blocks of conditions can be nested as shown in the following example:

```
EXEC ORACLE IFDEF OUTER END-EXEC.
EXEC ORACLE IFDEF INNER END-EXEC.
...
EXEC ORACLE ENDIF END-EXEC.
EXEC ORACLE ENDIF END-EXEC.
```

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 END-EXEC.
```

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. For example, predefined operating system symbols include CMS, MVS, MS-DOS, UNIX, and VMS.

Separate Precompilations

You can precompile several COBOL program modules separately, then link them into one executable program. This supports modular programming, which is required when the functional components of a program are written and debugged by different programmers. 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 names are SQL identifiers, whose scope is the precompilation unit. Hence, 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 precompile 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 globally in one of the program modules.

Using a Single DATE_FORMAT

You must use the same format string for DATE in each program module.

Restrictions

All references to an explicit cursor must be in the same program file. You cannot perform operations on a cursor that was DECLAREd in a different module. See Chapter 4 for more information about cursors.

Also, any program file that contains SQL statements must have a SQLCA that is in the scope of the local SQL statements.

Compiling and Linking

To get an executable program, you must compile the source file(s) produced by Pro*COBOL, then link the resulting object module with any modules needed from SQLLIB and system-specific Oracle libraries. Also, if you are embedding OCI calls, make sure to link in the OCI runtime library (OCILIB).

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 you might have problems. Check with Oracle Customer Support to see if the software is supported.

Compiling and linking are system-dependent. For example, on some systems, you must turn off compiler optimization when compiling a host language program. For instructions, see your system-specific Oracle manuals.

8

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 Oracle8 data are made permanent or undone. The following topics are discussed:

- Some Terms You Should Know

- How Transactions Guard Your Database

- How to Begin and End Transactions

- Using the COMMIT Statement

- Using the ROLLBACK Statement

- Using the SAVEPOINT Statement

- Using the RELEASE Option

- Using the SET TRANSACTION Statement

- Overriding Default Locking

- Fetching Across Commits

- 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 Oracle8 manages are called *sessions*. A *user session* is started when you run an application program or a tool such as Oracle Forms and connect to Oracle8. Oracle8 allows user sessions to work "simultaneously" and share computer resources. To do this, Oracle8 must control *concurrence*, the accessing of the same data by many users. Without adequate concurrence controls, there might be a loss of *data integrity*. That is, changes to data or structures might be made in the wrong order.

Oracle8 uses *locks* 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 Oracle8 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 Oracle8 breaks the deadlock. Oracle8 signals an error to the participating transaction that had completed the least amount of work, and the "deadlock detected while waiting for resource" Oracle8 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, Oracle8 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, Oracle8 takes *snapshots* of the table's data and records changes in a *rollback segment*. Oracle8 uses information in the rollback segment to build read-consistent query results and to undo changes if necessary.

How Transactions Guard Your Database

Oracle8 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. Oracle8 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 *Oracle8 SQL Reference.*

How to Begin and End Transactions

You begin a transaction with the first executable SQL statement (other than CON-NECT) in your program. When one transaction ends, the next executable SQL state-ment automatically begins another transaction. Thus, every executable statement is part of a transaction. Because they cannot be rolled back and need not be commit-ted, 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) that 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 inter-rupt. Oracle8 rolls back the transaction.

If your program fails in the middle of a transaction, Oracle8 detects the error and rolls back the transaction. If your operating system fails, Oracle8 restores the database to its former (pre-transaction) state.

Using the COMMIT Statement

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. The COMMIT statement has no effect on the values of host variables or on the flow of control in your program. 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

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 "Fetching Across Commits" on page 8-12.

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, Oracle8 rolls them back. In the following example, you commit your transaction and disconnect from Oracle8:

```
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all Oracle8 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.

WITH HOLD Clause in DECLARE CURSOR Statements

Starting with Pro*COBOL 8.0, any cursor that has been declared with the clause *WITH HOLD* after the word CURSOR, remains open after a COMMIT or a ROLLBACK. The following example shows how to use this clause:

```
EXEC SQL
    DECLARE C1 CURSOR WITH HOLD
    FOR SELECT ENAME FROM EMP
    WHERE EMPNO BETWEEN 7600 AND 7700
END-EXEC.
```

The cursor must not be declared for UPDATE.The WITH HOLD clause is used in DB2 to override the default, which is to close all cursors on commit. Pro*COBOL provides this clause in order to ease migrations of applications from DB2 to Oracle8. When MODE=ANSI, Oracle8 uses the DB2 default, but all host variables must be declared in a Declare Section. To avoid having a Declare Section, use the precompiler option *CLOSE_ON_COMMIT* described next. See *DECLARE CURSOR(Embedded SQL Directive)* on page "DECLARE CURSOR (Embedded SQL Directive)" on page F-14.

CLOSE_ON_COMMIT Precompiler Option

The precompiler option CLOSE_ON_COMMIT is available for DB2 compatibility:

CLOSE_ON_COMMIT = YES | NO

The default is NO and this option must be entered on the command line or in a configuration file. This option will only be in effect when the cursor is declared using the WITH HOLD clause. If you specify MODE=ORACLE on the command line, any cursors not declared with the WITH HOLD clause are closed on commit. See "CLOSE_ON_COMMIT" on page 7-14.

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 ROLLBACK statement has no effect on the values of host variables or on the flow of control in your program. 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

When MODE={ANSI13 | ORACLE}, explicit cursors not referenced in a CURRENT OF clause remain open across rollbacks.

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 roll back your transaction and disconnect from Oracle8:

```
EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
```

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.

For example, consider the following:

```
    EXEC SQL
        WHENEVER SQLERROR GOTO SQL-ERROR
    END-EXEC.
    ...
    DISPLAY 'Employee number? '.
    ACCEPT EMP-NUMBER.
    DISPLAY 'Employee name? '.
    ACCEPT EMP-NAME.
    EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
        VALUES (:EMP-NUMBER, :EMP-NAME)
    END-EXEC.
    ...
 SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    DISPLAY 'Processing error'.
* -- exit program with an error.
    STOP RUN.
```

Oracle8 rolls back transactions if your program terminates abnormally.

Statement-Level Rollbacks

Before executing any SQL statement, Oracle8 marks an implicit savepoint (not available to you). Then, if the statement fails, Oracle8 rolls it back automatically 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.

Only work started by the failed SQL statement is lost; work done before that statement in the current transaction is kept. Thus, if a data definition statement fails, the automatic commit that precedes it is not undone.

> **Note:** Before executing a SQL statement, Oracle8 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.

Oracle8 can also roll back single SQL statements to break deadlocks. Oracle8 signals an error to one of the participating transactions and rolls back the current statement in that transaction.

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_delete*:

```
EXEC SQL SAVEPOINT start_delete END-EXEC.
```

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 Oracle8 data to its former state, recover, then re-execute the function.

To undo part of a transaction, you use savepoints with the ROLLBACK statement and its TO SAVEPOINT clause. The TO SAVEPOINT clause lets you roll back to an intermediate statement in the current transaction, so you do not have to undo all your changes. 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

In the example below, you access the table MAIL_LIST to insert new listings, update old listings, and delete (a few) inactive listings. After the delete, you check SQLERRD(3) in the SQLCA for the number of rows deleted. If the number is unexpectedly large, you roll back to the savepoint *start_delete*, undoing just the delete.

```
* -- For each new customer
     DISPLAY 'New customer number? '.
     ACCEPT CUST-NUMBER.
     IF CUST-NUMBER = 0
          GO TO REV-STATUS
     END-IF.
     DISPLAY 'New customer name? '.
          ACCEPT  CUST-NAME.
     EXEC SQL INSERT INTO MAIL-LIST (CUSTNO, CNAME, STAT)
          VALUES (:CUST-NUMBER, :CUST-NAME, 'ACTIVE').
     END-EXEC.
     ...
* -- For each revised status
 REV-STATUS.
     DISPLAY 'Customer number to revise status? '.
     ACCEPT CUST-NUMBER.
     IF CUST-NUMBER = 0
          GO TO SAVE-POINT
     END-IF.
     DISPLAY 'New status? '.
     ACCEPT NEW-STATUS.
     EXEC SQL UPDATE MAIL-LIST
          SET STAT = :NEW-STATUS WHERE CUSTNO = :CUST-NUMBER
     END-EXEC.
     ...
* -- mark savepoint
 SAVE-POINT.
     EXEC SQL SAVEPOINT START-DELETE END-EXEC.
     EXEC SQL DELETE FROM MAIL-LIST WHERE STAT = 'INACTIVE'
     END-EXEC.
     IF SQLERRD(3) < 25
* -- check number of rows deleted
          DISPLAY 'Number of rows deleted is ', SQLERRD(3)
     ELSE
          DISPLAY 'Undoing deletion of ', SQLERRD(3), ' rows'
          EXEC SQL
              WHENEVER SQLERROR GOTO SQL-ERROR
          END-EXEC
          EXEC SQL
              ROLLBACK TO SAVEPOINT START-DELETE
```

```
          END-EXEC
      END-IF.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      EXEC SQL COMMIT WORK RELEASE END-EXEC.
      STOP RUN.
* -- exit program.
      ...
 SQL-ERROR.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
      DISPLAY 'Processing error'.
* -- exit program with an error.
      STOP RUN.
```

Note that you cannot specify the RELEASE option in a ROLLBACK TO SAVE-POINT statement.

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 roll back to the third, only the fourth and fifth are erased. A COMMIT or ROLLBACK statement erases all savepoints.

By default, the number of active savepoints per user 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 by increasing the value of the Oracle8 initialization parameter SAVEPOINTS. If you give two savepoints the same name, the earlier savepoint is erased.

Using the RELEASE Option

Oracle8 rolls back changes automatically if your program terminates abnormally. Abnormal termination occurs when your program does not explicitly commit or roll back work and disconnect from Oracle8 using the RELEASE option.

Normal termination occurs when your program runs its course, closes open cursors, explicitly commits or rolls back work, disconnects from Oracle8, and returns control to the user. Your program will exit gracefully if the last SQL statement it executes is either

```
      EXEC SQL COMMIT RELEASE END-EXEC.
```

or

```
      EXEC SQL ROLLBACK RELEASE END-EXEC.
```

Otherwise, locks and cursors acquired by your user session are held after program termination until Oracle8 recognizes that the user session is no longer active. This might cause other users in a multi-user 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 or read-write transaction, or to assign your current transaction to a specified rollback segment. A COMMIT, ROLLBACK, or data definition statement ends 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. During a read-only transaction, all queries refer to the same snapshot of the database, providing a multi-table, multi-query, read-consistent view. Other users can continue to query or update data as usual. An example of the SET TRANSACTION statement follows:

```
EXEC SQL SET TRANSACTION READ ONLY END-EXEC.
```

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 (without FOR UPDATE), LOCK TABLE, SET ROLE, ALTER SESSION, ALTER SYSTEM, COMMIT, and ROLLBACK statements are allowed in a read-only transaction.

In the example below, 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 END-EXEC.
EXEC SQL SELECT SUM(SALEAMT) INTO :DAILY FROM SALES
    WHERE SALEDATE = SYSDATE END-EXEC.
EXEC SQL SELECT SUM(SALEAMT) INTO :WEEKLY FROM SALES
    WHERE SALEDATE > SYSDATE - 7 END-EXEC.
EXEC SQL SELECT SUM(SALEAMT) INTO :MONTHLY FROM SALES
    WHERE SALEDATE > SYSDATE - 30 END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.
* --  simply ends the transaction since there are no changes
* --  to make permanent
* --  format and print report
```

Overriding Default Locking

By default, Oracle8 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 multi-table and multi-query 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, Oracle8 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 the FOR UPDATE OF Clause

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. (All rows are locked at the open, not as they are fetched.) 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 instance, instead of

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
    SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20
    FOR UPDATE OF SAL
END-EXEC.
```

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
END-EXEC.
```

The CURRENT OF clause signals the precompiler 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 "Using the CURRENT OF Clause" on page 5-15.

Restrictions

If you use the FOR UPDATE OF clause, you cannot reference multiple tables. Also, an explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. Row locks are released when you commit or rollback (except when you rollback to a savepoint). If you try to fetch from a FOR UPDATE cursor after a commit, Oracle8 generates the following error:

```
ORA-01002: fetch out of sequence
```

Using the LOCK TABLE Statement

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
END-EXEC.
```

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 *Oracle8 Application Developer's Guide*.

The optional keyword NOWAIT tells Oracle8 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 table lock failed.) If you omit NOWAIT, Oracle8 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. Consider the following example:

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
    SELECT ENAME, SAL, ROWID FROM EMP WHERE JOB = 'CLERK'
END-EXEC.
...
EXEC SQL OPEN EMP-CURSOR END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO ...
PERFORM
EXEC SQL
    FETCH EMP-CURSOR INTO :EMP_NAME, :SALARY, :ROW-ID
END-EXEC
...
    EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
        WHERE ROWID = :ROW-ID
    END-EXEC
    EXEC SQL COMMIT END-EXEC
END-PERFORM.
```

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 roll back 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' END-EXEC.
...
EXEC SQL ROLLBACK FORCE '25.33.86' END-EXEC.
```

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, Oracle8 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 of no more than 50 characters in length. An example follows:

```
EXEC SQL
    COMMIT COMMENT 'In-doubt trans; notify Order Entry'
END-EXEC.
```

For more information about distributed transactions, see *Oracle8 Concepts*.

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 Oracle8 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.
...
```

9

Error Handling and Diagnostics

An application program must anticipate runtime errors and attempt to recover from them. This chapter provides an in-depth discussion of error reporting and recovery. You learn how to handle warnings and errors using the status variables SQLCODE, SQLSTATE, and SQLCA (SQL Communications Area), and the WHENEVER statement. You also learn how to diagnose problems using the status variable ORACA (Oracle Communications Area). The following topics are discussed:

- The Need for Error Handling
- Error Handling Alternatives
- Using Status Variables when MODE={ANSI | ANSI14}
- Using the SQL Communications Area
- Using the Oracle Communications Area

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 Pro*COBOL, 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

Pro*COBOL supports four status variables that serve as error handling mechanisms:

- SQLCODE

- SQLSTATE

- SQLCA (using the WHENEVER statement)

- ORACA

The precompiler MODE option governs ANSI/ISO compliance. The availability of the SQLCODE, SQLSTATE, and SQLCA variables depends on the MODE setting. You can declare and use the ORACA variable regardless of the MODE setting. For more information, see "Using the Oracle Communications Area" on page 9-35.

When MODE={ORACLE|ANSI13}, you must declare the SQLCA status variable. SQLCODE and SQLSTATE declarations are accepted (not recommended) but are not recognized as status variables. For more information, see "Using the SQL Communications Area" on page 9-19.

When MODE={ANSI|ANSI14}, you can use any one, two, or all three of the SQLCODE, SQLSTATE, and SQLCA variables. To determine which variable (or variable combination) is best for your application, see "Using Status Variables when MODE={ANSI|ANSI14}" on page 9-4.

SQLCODE and SQLSTATE

With Release 1.5 of Pro*COBOL, the SQLCODE status variable was introduced as the SQL89 standard ANSI/ISO error reporting mechanism. The SQL92 standard listed SQLCODE as a deprecated feature and defined a new status variable, SQLSTATE (introduced with Release 1.6 of Pro*COBOL), as the preferred ANSI/ISO error reporting mechanism.

SQLCODE stores error codes and the "not found" condition. It is retained only for compatibility with SQL89 and is likely to be removed from future versions of the standard.

Unlike SQLCODE, SQLSTATE stores error and warning codes and uses a standardized coding scheme. After executing a SQL statement, the Oracle8 server returns a status code to the SQLSTATE variable currently in scope. The status code indicates whether a SQL statement executed successfully or raised an exception (error or warning condition). To promote *interpretability* (the ability of systems to exchange information easily), SQL92 pre-defines all the common SQL exceptions.

SQLCA

The SQLCA is a record-like, host-language data structure. Oracle8 updates the SQLCA after every *executable* SQL statement. (SQLCA values are undefined after a declarative statement.) By checking Oracle8 return codes stored in the SQLCA, your program can determine the outcome of a SQL statement. This can be done in 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.

Nested Programs

In nested programs, the included SQLCA definition provided will be declared as global, so the declaration of SQLCA will only be required within the higher-level program. SQLCA can change every time a new SQL statement is executed. The SQLCA provided can always be modified to remove the global specification by the user if the user wishes to declare additional SQLCAs in the nested programs. This applies to SQLDA and ORACA.

ORACA

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA, which contains cursor statistics, SQL statement data, option settings, and system statistics.

The ORACA is optional and can be declared regardless of the MODE setting. For more information about the ORACA status variable, see "Using the Oracle Communications Area" on page 9-35.

Using Status Variables when MODE={ANSI|ANSI14}

When MODE={ANSI | ANSI14}, you must declare at least one — you may declare two or all three — of the following status variables:

- SQLCODE
- SQLSTATE
- SQLCA

You cannot declare SQLCODE if SQLCA is declared. Likewise, you cannot declare SQLCA if SQLCODE is declared. The field in the SQLCA data structure that stores the error code for is also called SQLCODE, so errors will occur if both status variables are declared.

Your program can get the outcome of the most recent executable SQL statement by checking SQLCODE and/or SQLSTATE explicitly with your own code after executable SQL and PL/SQL statements. Your program can also check SQLCA implicitly (with the WHENEVER SQLERROR and WHENEVER SQLWARNING statements) or it can check the SQLCA variables explicitly.

Note: When MODE={ORACLE | ANSI13 | ANSI14}, you must declare the SQLCA status variable. For more information, see "Using the SQL Communications Area" on page 9-19.

Some Historical Information

The treatment of status variables and variable combinations by Pro*COBOL has evolved beginning with Release 1.5.

Release 1.5
Pro*COBOL, Release 1.5, presumed there was a status variable SQLCODE whether or not it was declared; in fact, Pro*COBOL never noted whether SQLCODE was

declared or not — it just presumed it was. SQLCA would be used as a status variable if and only if there was an INCLUDE of the SQLCA.

Release 1.6

Beginning with Pro*COBOL, Release 1.6, the precompiler no longer presumes that there is a SQLCODE status variable and it is not required. Pro*COBOL requires that *at least* one of SQLCODE or SQLSTATE be declared.

SQLCODE is recognized as a status variable if and only if at least one of the following criteria is satisfied:

- It is declared with *exactly* the right datatype.

- Pro*COBOL finds no other status variable.

If Pro*COBOL finds a SQLSTATE declaration (of *exactly* the right type of course) or finds an INCLUDE of the SQLCA, it will *not* presume SQLCODE is declared.

Release 1.7

Because Pro*COBOL, Release 1.5, allowed the SQLCODE variable to be declared outside of a Declare Section while also declaring SQLCA, Pro*COBOL, Release 1.6 and greater, is presented with a compatibility problem. A new option, ASSUME_SQLCODE={YES | NO} (default NO), was added to fix this in Release 1.6.7 and is documented as a new feature in Release 1.7.

Release 8.0

Beginning with release 8.0, the Declare Section is now optional. For details of the ASSUME_SQLCODE option, see "ASSUME_SQLCODE" on page 7-12.

Declaring Status Variables

This section describes how to declare SQLCODE and SQLSTATE. For information about declaring the SQLCA status variable, see "Declaring the SQLCA" on page 9-20.

Declaring SQLCODE

SQLCODE must be declared as a 4-byte integer variable either *inside* or *outside* the Declare Section, as shown in the following example:

```
*     Declare host and indicator variables.
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      ...
      EXEC SQL END DECLARE SECTION END-EXEC.
```

```
*     Declare the SQLCODE status variable.
        01  SQLCODE  PIC S9(9) COMP.
```

If declared outside the Declare Section, SQLCODE is recognized as a status variable if and only if ASSUME_SQLCODE=YES. When MODE={ORACLE | ANSI13 | ANSI14}, declarations of the SQLCODE variable are ignored.

> **Warning:** *Do not* declare SQLCODE if SQLCA is declared. Likewise, *do not* declare SQLCA if SQLCODE is declared. The status variable declared by the SQLCA structure is also called SQLCODE, so errors will occur if both error-reporting mechanisms are used.

After every SQL operation, Oracle8 returns a status code to the SQLCODE variable. 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, Pro*COBOL allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA.

Declaring SQLSTATE

SQLSTATE must be declared as a five-character alphanumeric string, as shown in the following example:

```
*     Declare the SQLSTATE status variable.
        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
          ...
          01 SQLSTATE PIC X(5).
          ...
        EXEC SQL END DECLARE SECTION END-EXEC.
```

When MODE={ORACLE | ANSI13 | ANSI14}, SQLSTATE declarations are ignored. Declaring the SQLCA is optional.

Status Variable Combinations

When MODE={ANSI | ANSI14}, the behavior of the status variables depends on the following:

- which variables are declared

- declaration placement (*inside* or *outside* the Declare Section)

- ASSUME_SQLCODE setting

Table 9–1 and Table 9–2 describe the resulting behavior of each status variable combination when ASSUME_SQLCODE=NO and when ASSUME_SQLCODE=YES, respectively.

For both Tables 9-1 and 9-2: when DECLARE_SECTION=NO, any declaration of a status variable is treated as IN as far as these tables are concerned.

Do not use ASSUME_SQLCODE=YES with DECLARE_SECTION=NO.

Table 9–1   Status Variable Behavior with ASSUME_SQLCODE=NO and MODE=ANSI | ANSI14 and DECLARE_SECTION=YES

Declare Section (IN/OUT/—)			Behavior
SQLCODE	SQLSTATE	SQLCA	
OUT	—	—	SQLCODE is declared and is presumed to be a status variable.
OUT	—	OUT	This status variable configuration is not supported.
OUT	—	IN	This status variable configuration is not supported.
OUT	OUT	—	SQLCODE is declared and is presumed to be a status variable, and SQLSTATE is declared but is not recognized as a status variable.
OUT	OUT	OUT	This status variable configuration is not supported.
OUT	OUT	IN	This status variable configuration is not supported.
OUT	IN	—	SQLSTATE is declared as a status variable, and SQLCODE is declared but is not recognized as a status variable.
OUT	IN	OUT	This status variable configuration is not supported.
OUT	IN	IN	This status variable configuration is not supported.
IN	—	—	SQLCODE is declared as a status variable.
IN	—	OUT	This status variable configuration is not supported.
IN	—	IN	This status variable configuration is not supported.
IN	OUT	—	SQLCODE is declared as a status variable, and SQLSTATE is declared but is not recognized as a status variable.
IN	OUT	OUT	This status variable configuration is not supported.
IN	OUT	IN	This status variable configuration is not supported.
IN	IN	—	SQLCODE and SQLSTATE are declared as a status variables.
IN	IN	OUT	This status variable configuration is not supported.
IN	IN	IN	This status variable configuration is not supported.
—	—	—	This status variable configuration is not supported.
—	—	OUT	SQLCA is declared as a status variable.

Table 9–1 Status Variable Behavior with ASSUME_SQLCODE=NO and MODE=ANSI | ANSI14 and DECLARE_SECTION=YES

Declare Section (IN/OUT/—)			Behavior
SQLCODE	SQLSTATE	SQLCA	
—	—	IN	SQLCA is declared as a status host variable.
—	OUT	—	This status variable configuration is not supported.
—	OUT	OUT	SQLCA is declared as a status variable, and SQLSTATE is declared but is not recognized as a status variable.
—	OUT	IN	SQLCA is declared as a status host variable, and SQLSTATE is declared but is not recognized as a status variable.
—	IN	—	SQLSTATE is declared as a status variable.
—	IN	OUT	SQLSTATE and SQLCA are declared as status variables.
—	IN	IN	SQLSTATE and SQLCA are declared as status host variables.

Table 9–2 Status Variable Behavior with ASSUME_SQLCODE=YES and MODE=ANSI | ANSI14 and DECLARE_SECTION=YES

Declare Section (IN/OUT/—)			Behavior
SQLCODE	SQLSTATE	SQLCA	
OUT	—	—	SQLCODE is declared and is presumed to be a status variable.
OUT	—	OUT	This status variable configuration is not supported.
OUT	—	IN	This status variable configuration is not supported.
OUT	OUT	—	SQLCODE is declared and is presumed to be a status variable, and SQLSTATE is declared but is not recognized as a status variable.
OUT	OUT	OUT	This status variable configuration is not supported.
OUT	OUT	IN	This status variable configuration is not supported.
OUT	IN	—	SQLSTATE is declared as a status variable, and SQLCODE is declared and is presumed to be a status variable.
OUT	IN	OUT	This status variable configuration is not supported.
OUT	IN	IN	This status variable configuration is not supported.
IN	—	—	SQLCODE is declared as a status variable.
IN	—	OUT	This status variable configuration is not supported.
IN	—	IN	This status variable configuration is not supported.
IN	OUT	—	SQLCODE is declared as a status variable, and SQLSTATE is declared but not as a status variable.
IN	OUT	OUT	This status variable configuration is not supported.

Table 9–2   Status Variable Behavior with ASSUME_SQLCODE=YES and MODE=ANSI | ANSI14 and DECLARE_SECTION=YES

Declare Section (IN/OUT/—)			Behavior
SQLCODE	SQLSTATE	SQLCA	
IN	OUT	IN	This status variable configuration is not supported.
IN	IN	—	SQLCODE and SQLSTATE are declared as a status variables.
IN	IN	OUT	This status variable configuration is not supported.
IN	IN	IN	This status variable configuration is not supported.
—	—	—	These status variable configurations are not supported. SQLCODE must be declared when ASSUME_SQLCODE=YES.
—	—	OUT	
—	—	IN	
—	OUT	—	
—	OUT	OUT	
—	OUT	IN	
—	IN	—	
—	IN	OUT	
—	IN	IN	

Status Variable Values

This section describes the values for the SQLCODE and SQLSTATE status variables. For information about the SQLCA status variable, see "Key Components of Error Reporting" on page 9-21.

SQLCODE Values

After every SQL operation, Oracle8 returns a status code to the SQLCODE variable currently in scope. The status code, which indicates the outcome of the SQL operation, can be any of the following numbers:'

You can learn the outcome of the most recent SQL operation by checking SQLCODE explicitly with your own code or implicitly with the WHENEVER statement.

When you declare SQLCODE instead of the SQLCA in a particular precompilation unit, Pro*COBOL allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA.

Note: When MODE={ORACLE | ANSI13}, declarations of SQLCODE are ignored.

SQLSTATE Values

SQLSTATE status codes consist of a two-character *class code* followed by a three-character *subclass code*. Aside from class code 00 (successful completion), the class code denotes a category of exceptions. Aside from subclass code 000 (not applicable), the subclass code denotes a specific exception within that category. For example, the SQLSTATE value '22012' consists of class code 22 (data exception) and subclass code 012 (division by zero).

Each of the five characters in a SQLSTATE value is a digit (0..9) or an uppercase Latin letter (A..Z). Class codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined conditions (those defined in SQL92). All other class codes are reserved for implementation-defined conditions. Within predefined classes, subclass codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined sub-conditions. All other subclass codes are reserved for implementation-defined sub-conditions. Figure 9–1 shows the coding scheme.

Figure 9–1   SQLSTATE Coding Scheme

Table 9–3 shows the classes predefined by SQL92

Table 9–3   Predifined Classes

Class	Condition
00	successful completion
01	warning
02	no data
07	dynamic SQL error
08	connection exception
0A	feature not supported
21	cardinality violation
22	data exception
23	integrity constraint violation
24	invalid cursor state
25	invalid transaction state
26	invalid SQL statement name
27	triggered data change violation
28	invalid authorization specification
2A	direct SQL syntax error or access rule violation
2B	dependent privilege descriptors still exist
2C	invalid character set name
2D	invalid transaction termination
2E	invalid connection name
33	invalid SQL descriptor name
34	invalid cursor name
35	invalid condition number
37	dynamic SQL syntax error or access rule violation
3C	ambiguous cursor name

Table 9–3   Predifined Classes

Class	Condition
3D	invalid catalog name
3F	invalid schema name
40	transaction rollback
42	syntax error or access rule violation
44	with check option violation
HZ	remote database access

Note: The class code HZ is reserved for conditions defined in International Standard ISO/IEC DIS 9579-2, *Remote Database Access.*

Table 9–4 shows how Oracle8 errors map to SQLSTATE status codes. In some cases, several Oracle8 errors map to the status code. In other cases, no Oracle8 error maps to the status code (so the last column is empty). Status codes in the range 60000 .. 99999 are implementation-defined.

Table 9–4   SQLSTATE Codes

Code	Condition	Oracle8 Error
00000	successful completion	ORA-00000
01000	warning	
01001	cursor operation conflict	
01002	disconnect error	
01003	null value eliminated in set function	
01004	string data - right truncation	
01005	insufficient item descriptor areas	
01006	privilege not revoked	
01007	privilege not granted	
01008	implicit zero-bit padding	
01009	search condition too long for info schema	

Table 9–4   SQLSTATE Codes

Code	Condition	Oracle8 Error
0100A	query expression too long for info schema	
02000	no data	ORA-01095 ORA-01403
07000	dynamic SQL error	
07001	using clause does not match parameter specs	
07002	using clause does not match target specs	
07003	cursor specification cannot be executed	
07004	using clause required for dynamic parameters	
07005	prepared statement not a cursor specification	
07006	restricted datatype attribute violation	
07007	using clause required for result fields	
07008	invalid descriptor count	SQL-02126
07009	invalid descriptor index	
08000	connection exception	
08001	SQL client unable to establish SQL connection	
08002	connection name in use	
08003	connection does not exist	SQL-02121
08004	SQL server rejected SQL connection	
08006	connection failure	
08007	transaction resolution unknown	
0A000	feature not supported	ORA-03000 .. 03099
0A001	multiple server transactions	
21000	cardinality violation	ORA-01427 SQL-02112
22000	data exception	
22001	string data - right truncation	ORA-01401 ORA-01406

Table 9–4   SQLSTATE Codes

Code	Condition	Oracle8 Error
22002	null value - no indicator parameter	ORA-01405 SQL-02124
22003	numeric value out of range	ORA-01426 ORA-01438 ORA-01455 ORA-01457
22005	error in assignment	
22007	invalid datetime format	
22008	datetime field overflow	ORA-01800 .. 01899
22009	invalid time zone displacement value	
22011	substring error	
22012	division by zero	ORA-01476
22015	interval field overflow	
22018	invalid character value for cast	
22019	invalid escape character	ORA-00911 ORA-01425
22021	character not in repertoire	
22022	indicator overflow	ORA-01411
22023	invalid parameter value	ORA-01025 ORA-01488 ORA-04000 .. 04019
22024	unterminated C string	ORA-01479 .. 01480
22025	invalid escape sequence	ORA-01424
22026	string data - length mismatch	
22027	trim error	
23000	integrity constraint violation	ORA-00001 ORA-02290 .. 02299

Table 9–4   SQLSTATE Codes

Code	Condition	Oracle8 Error
24000	invalid cursor state	ORA-01001 .. 01003
		ORA-01410
		ORA-08006
		SQL-02114
		SQL-02117
		SQL-02118
		SQL-02122
25000	invalid transaction state	
26000	invalid SQL statement name	
27000	triggered data change violation	
28000	invalid authorization specification	
2A000	direct SQL syntax error or access rule violation	
2B000	dependent privilege descriptors still exist	
2C000	invalid character set name	
2D000	invalid transaction termination	
2E000	invalid connection name	
33000	invalid SQL descriptor name	
34000	invalid cursor name	
35000	invalid condition number	
37000	dynamic SQL syntax error or access rule violation	
3C000	ambiguous cursor name	
3D000	invalid catalog name	
3F000	invalid schema name	
40000	transaction rollback	ORA-02091 .. 02092
40001	serialization failure	
40002	integrity constraint violation	

Table 9–4 SQLSTATE Codes

Code	Condition	Oracle8 Error
40003	statement completion unknown	
42000	syntax error or access rule violation	ORA-00022
		ORA-00251
		ORA-00900 .. 00999
		ORA-01031
		ORA-01490 .. 01493
		ORA-01700 .. 01799
		ORA-01900 .. 02099
		ORA-02140 .. 02289
		ORA-02420 .. 02424
		ORA-02450 .. 02499
		ORA-03276 .. 03299
		ORA-04040 .. 04059
		ORA-04070 .. 04099
44000	with check option violation	ORA-01402
60000	system errors	ORA-00370 .. 00429
		ORA-00600 .. 00899
		ORA-06430 .. 06449
		ORA-07200 .. 07999
		ORA-09700 .. 09999
61000	resource error	ORA-00018 .. 00035
		ORA-00050 .. 00068
		ORA-02376 .. 02399
		ORA-04020 .. 04039
62000	multi-threaded server and detached process errors	ORA-00100 .. 00120
		ORA-00440 .. 00569

Table 9–4   SQLSTATE Codes

Code	Condition	Oracle8 Error
63000	Oracle*XA and two-task interface errors	ORA-00150 .. 00159
		SQL-02128
		ORA-02700 .. 02899
		ORA-03100 .. 03199
		ORA-06200 .. 06249 SQL-02128
64000	control file, database file, and redo file errors;	ORA-00200 .. 00369
	archival and media recovery errors	ORA-01100 .. 01250
65000	PL/SQL errors	ORA-06500 .. 06599
66000	SQL*Net driver errors	ORA-06000 .. 06149
		ORA-06250 .. 06429
		ORA-06600 .. 06999
		ORA-12100 .. 12299
		ORA-12500 .. 12599
67000	licensing errors	ORA-00430 .. 00439
69000	SQL*Connect errors	ORA-00570 .. 00599
		ORA-07000 .. 07199
72000	SQL execute phase errors	ORA-01000 .. 01099
		ORA-01400 .. 01489
		ORA-01495 .. 01499
		ORA-01500 .. 01699
		ORA-02400 .. 02419
		ORA-02425 .. 02449
		ORA-04060 .. 04069
		ORA-08000 .. 08190
		ORA-12000 .. 12019
		ORA-12300 .. 12499
		ORA-12700 .. 21999
82100	out of memory (could not allocate)	SQL-02100

Table 9–4   SQLSTATE Codes

Code	Condition	Oracle8 Error
82101	inconsistent cursor cache: unit cursor/global cursor mismatch	SQL-02101
82102	inconsistent cursor cache: no global cursor entry	SQL-02102
82103	inconsistent cursor cache: out of range cursor cache reference	SQL-02103
82104	inconsistent host cache: no cursor cache available	SQL-02104
82105	inconsistent cursor cache: global cursor not found	SQL-02105
82106	inconsistent cursor cache: invalid Oracle8 cursor number	SQL-02106
82107	program too old for runtime library	SQL-02107
82108	invalid descriptor passed to runtime library	SQL-02108
82109	inconsistent host cache: host reference is out of range	SQL-02109
82110	inconsistent host cache: invalid host cache entry type	SQL-02110
82111	heap consistency error	SQL-02111
82112	unable to open message file	SQL-02113
82113	code generation internal consistency failed	SQL-02115
82114	reentrant code generator gave invalid context	SQL-02116
82115	invalid hstdef argument	SQL-02119
82116	first and second arguments to sqlrcn both null	SQL-02120
82117	invalid OPEN or PREPARE for this connection	SQL-02122
82118	application context not found	SQL-02123
82119	connect error; can't get error text	SQL-02125
82120	precompiler/SQLLIB version mismatch.	SQL-02127
82121	FETCHed number of bytes is odd	SQL-02129
82122	EXEC TOOLS interface is not available	SQL-02130

Table 9–4   SQLSTATE Codes

Code	Condition	Oracle8 Error
82123	runtime context in use	SQL-02131
82124	unable to allocate runtime context	SQL-02131
82125	unable to initialize process for use with threads	SQL-02133
82126	invalid runtime context	SQL-02134
90000	debug events	ORA-10000 .. 10999
99999	catch all	all others
HZ000	remote database access	

Using the SQL Communications Area

Oracle8 uses the SQL Communications Area (SQLCA) to store status information passed to your program at run time. The SQLCA is a record-like, COBOL data structure that is a updated after each executable SQL statement, so it always reflects the outcome of the most recent SQL operation. Its fields contain error, warning, and status information updated by Oracle8 whenever a SQL statement is executed.To determine that outcome, you can check variables in the SQLCA explicitly with your own COBOL code or implicitly with the WHENEVER statement.

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 "Concurrent Logons" on page 3-46.

When MODE={ORACLE | ANSI13}, the SQLCA is required; if the SQLCA is not declared, compile-time errors will occur. 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.

Note: If you declare SQLCODE instead of the SQLCA in a particular compilation unit, Pro*COBOL allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA.

When MODE={ANSI | ANSI14}, you must declare either SQLSTATE (see "Declaring SQLSTATE" on page 9-6) or SQLCODE (see "Declaring SQLCODE" on page 9-5) or both. The SQLSTATE status variable supports the SQLSTATE status variable

specified by the SQL92 standard. You can use the SQLSTATE status variable with or without SQLCODE.

What's in the SQLCA?

The SQLCA contains runtime information about the execution of SQL statements, such as Oracle8 error codes, warning flags, event information, rows-processed count, and diagnostics.

Figure 9–2 shows all the variables in the SQLCA. However, SQLWARN2, SQLWARN5, SQLWARN6, SQLWARN7, and SQLEXT are not currently in use.

Figure 9–2   SQLCA Variable Declarations for Pro*COBOL

```
01   SQLCA.
     05   SQLCAID              PIC X(8).
     05   SQLCABC              PIC S9(9) COMPUTATIONAL.
     05   SQLCODE              PIC S9(9) COMPUTATIONAL.
     05   SQLERRM.
          49 SQLERRML          PIC S9(4) COMPUTATIONAL.
          49 SQLERRMC          PIC X(70)
     05   SQLERRP              PIC X(8).
     05   SQLERRD OCCURS 6 TIMES
                               PIC S9(9) COMPUTATIONAL.
     05   SQLWARN.
          10 SQLWARN0          PIC X(1).
          10 SQLWARN1          PIC X(1).
          10 SQLWARN2          PIC X(1).
          10 SQLWARN3          PIC X(1).
          10 SQLWARN4          PIC X(1).
          10 SQLWARN5          PIC X(1).
          10 SQLWARN6          PIC X(1).
          10 SQLWARN7          PIC X(1).
     05   SQLEXT               PIC X(8).
```

Declaring the SQLCA

To declare the SQLCA, simply include it (using an EXEC SQL INCLUDE statement) in your Pro*COBOL source file outside the Declare Section as follows:

```
*     Include the SQL Communications Area (SQLCA).
      EXEC SQL INCLUDE SQLCA END-EXEC.
```

The SQLCA must be declared *outside* the Declare Section.

> **Warning:** *Do not* declare SQLCODE if SQLCA is declared. Likewise, do *not* declare SQLCA if SQLCODE is declared. The status variable declared by the SQLCA structure is also called SQLCODE, so errors will occur if both error-reporting mechanisms are used.

When you precompile your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow Oracle8 to communicate with the program.

> **Attention**: When using multi-byte NCHAR host variables, the SQLCA must be included.

Key Components of Error Reporting

The key components of Pro*COBOL error reporting depend on several fields in the SQLCA.

Status Codes

Every executable SQL statement returns a status code in the SQLCA variable SQLCODE, which you can check implicitly with WHENEVER SQLERROR or explicitly with your own COBOL code.

Warning Flags

Warning flags are returned in the SQLCA variables SQLWARN0 through SQLWARN7, which you can check with WHENEVER SQLWARNING or with your own COBOL code. These warning flags are useful for detecting runtime conditions that are not considered errors by Oracle8.

Rows-Processed Count

The number of rows processed by the most recently executed SQL statement is returned in the SQLCA variable SQLERRD(3). For repeated FETCHes on an OPEN cursor, SQLERRD(3) keeps a running total of the number of rows fetched.

Parse Error Offset

Before executing a SQL statement, Oracle8 must parse it; that is, examine it to make sure it follows syntax rules and refers to valid database objects. If Oracle8 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 tenth character.

If your SQL statement does not cause a parse error, Oracle8 sets SQLERRD(5) to zero. Oracle8 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 Oracle8 errors are available in the SQLCA variable SQLERRMC. For example, you might place the following statements in an error-handling routine:

```
*      Handle SQL execution errors.
       MOVE SQLERRMC TO ERROR-MESSAGE.
       DISPLAY ERROR-MESSAGE.
```

At most, the first 70 characters of message text are stored. For messages longer than 70 characters, you must call the SQLGLM subroutine, which is discussed next.

SQLCA Structure

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	Oracle8 executed the statement without detecting an error or exception.
> 0	Oracle8 executed the statement but detected an exception. This occurs when Oracle8 cannot find a row that meets your WHERE-clause search condition or when a SELECT INTO or FETCH returns no rows.

< 0	When MODE={ANSI	ANSI14	ANSI113}, +100 is returned to SQLCODE after an INSERT of no rows. This can happen when a subquery returns no rows to process.
	Oracle8 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, in most cases, be rolled back.
	Negative return codes correspond to error codes listed in *Oracle8 Server Messages*.

SQLERRM

This sub-record 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 for the error code stored in SQLCODE and can store up to 70 characters. For the full text of messages longer than 70 characters, use the SQLGLM function.
	Verify 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 table 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 a table 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.

This table of single characters has eight elements. They are used as warning flags. Oracle8 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 Oracle8 assigns a truncated column value to an output host variable.

Note: While Figure 9–2 illustrates SQLWARN as a table, it is implemented in Pro*COBOL as a group item with elementary PIC X items named SQLWARN0 through SQLWARN7. .

Descriptions of the fields in SQLWARN follow:

SQLWARN(0)	This flag is set if another warning flag is set.
SQLWARN(1)	This flag is set if a truncated column value was assigned to an output host variable. This applies only to character data. Oracle8 truncates certain numeric data without setting a warning or returning a negative SQLCODE value.
	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(2)	This flag is set if one or more nulls were ignored in the evaluation of a SQL group function such as AVG, COUNT, or MAX. This behavior is expected because, except for COUNT(*), all group functions ignore nulls. If necessary, you can use the SQL function NVL to temporarily assign values (zeros, for example) to the null column entries.
SQLWARN(3)	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(4)	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 Oracle8 sets this warning flag. That way, you can roll back the transaction if necessary
SQLWARN(5)	This flag is set when an EXEC SQL CREATE {PROCEDURE|FUNCTION|PACKAGE|PACKAGE BODY} statement fails because of a PL/SQL compilation error.
SQLWARN(6)	This flag is no longer in use.
SQLWARN(7)	This flag is no longer in use.

SQLEXT

This string field is reserved for future use.

PL/SQL Considerations

When your Pro*COBOL program executes an embedded PL/SQL block, not all fields in the SQLCA are set. For example, if the block fetches several rows, the rows-processed count, SQLERRD(3), is set to 1, *not* the actual number of rows fetched. So, you should rely only on the SQLCODE and SQLERRM fields in the SQLCA after executing a PL/SQL block.

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 subroutine.

If connected to Oracle8, you can call SQLGLM using the syntax

```
CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH
```

where:

MSG-TEXT   is the field in which to store the error message. (Oracle8 blank-pads to the end of this field.)

MAX-SIZE   is an integer that specifies the maximum size of the MSG-TEXT field in bytes.

MSG-LENGTH   is an integer variable in which Oracle8 stores the actual length of the error message.

The maximum length of an Oracle8 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 specified for MAX-SIZE.

The following example uses SQLGLM to get an error message of up to 200 characters in length:

```
 WORKING-STORAGE SECTION.
     ...
*    Declare variables for the SQL-ERROR subroutine call.
     01  MSG-TEXT    PIC X(200).
     01  MAX-SIZE    PIC S9(9) COMP VALUE 200.
     01  MSG-LENGTH  PIC S9(9) COMP.
     ...
 PROCEDURE DIVISION.
 MAIN.
     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
     ...
 SQL-ERROR.
*    Clear the previous message text.
     MOVE SPACES TO MSG-TEXT.
*    Get the full text of the error message.
     CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH.
     DISPLAY MSG-TEXT.
```

In the example, 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.

Note: If your application calls SQLGLM to get message text or your Oracle*Forms user exit calls SQLIEM to display a failure message, the message length must be passed. Do *not* use the SQLCA variable SQLERRML; SQLERRML is a PIC S9(4) COMP integer while SQLGLM and SQLIEM expect a PIC S9(9) COMP integer. Instead, use another variable declared as PIC S9(9) COMP.

DSNTIAR

DB2 provides an assembler routine called DSNTIAR to obtain a form of the SQLCA that can be displayed. For users migrating to Oracle8 from DB2, Pro*COBOL provides DSNTIAR. DSNTIAR's implementation is a wrapper around SQLGLM. The DSNTIAR interface is as follows

```
CALL 'DSNTIAR' USING SQLCA MESSAGE LRECL
```

where *MESSAGE* is the output message area, in VARCHAR form of size greater than or equal to 240, and *LRECL* is a full word containing the length of the output messages, between 72 and 240. The first half-word of the MESSAGE argument contains the length of the remaining area. The possible error codes returned by DSNTIAR are:

Table 9–5   DNSTIAR Error Codes and Their Meanings

0	successful execution
4	more data was available than could fit into the provided message
8	the logical record length (LRECL) was not between 72 and 240
12	the message area was not large enough (greater than 240)

Using the WHENEVER Statement

By default, Pro*COBOL ignores Oracle8 error and warning conditions and continues 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 Oracle8 detects an error, warning condition, or "not found" condition. These actions include continuing with the next statement, PERFORMing a paragraph, branching to a paragraph, or stopping.

You can have Oracle8 automatically check the SQLCA for any of the following conditions.

Conditions

SQLWARNING

SQLWARN(0) is set because Oracle8 returned a warning (one of the warning flags, SQLWARN(1) through SQLWARN(7), is also set) or SQLCODE has a positive value

other than +1403. For example, SQLWARN(1) is set when Oracle8 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 Oracle8 returned an error.

NOT FOUND or NOTFOUND

SQLCODE has a value of +1403 (+100 when MODE={ANSI|ANSI14| ANSI13}), because Oracle8 could not find a row that meets the search condition of a WHERE clause, 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.

Since DB2 returns a SQLCODE value of 100 when an END-OF-FETCH condition occurs after a SQL statement execution, Pro*COBOL 2 provides a new command line option for explicit control over the value returned when the END-OF-FETCH condition occurs. This option is:

```
END_OF_FETCH = 100 | 1403 (default 1403)
```

The END_OF_FETCH option must be used on the command line or in a configuration file. For more details, see "END_OF_FETCH" on page 7-19

If the user specifies MODE=ANSI in a configuration file, Pro*COBOL 2 will implement the 100 at the END_OF_FETCH, overriding the default END_OF_FETCH=1403. If the user specifies MODE=ANSI and END_OF_FETCH=1403 in the configuration file, then Pro*COBOL 2 will implement the 1403 at the END_OF_FETCH. If the user specifies MODE=ANSI in the configuration file and END_OF_FETCH=1403 on the command line, Pro*COBOL 2 will again implement the 1403 at the END_OF_FETCH.

When Oracle8 detects one of the preceding *conditions*, you can have your program take any of the following actions.

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 PERFORM
Your program transfers control to a COBOL paragraph. When the end of the paragraph is reached, control transfers to the statement that follows the failed SQL statement.

```
EXEC SQL
    WHENEVER <condition> DO PERFORM <paragraph_name>
END-EXEC.
```

GOTO or GO TO
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 Oracle8.

Using the WHENEVER Statement in COBOL

Code the WHENEVER statement using the following syntax:

```
EXEC SQL
    WHENEVER <condition> <action>
END-EXEC.
```

When using the WHENEVER ... DO statement, the usual rules for PERFORMing a paragraph apply. However, you cannot use the THRU, TIMES, UNTIL, or VARYING clauses.

For example, the following WHENEVER ... DO statement is *invalid*:

```
 PROCEDURE DIVISION.
*    Invalid statement
    EXEC SQL WHENEVER SQLERROR DO
        PERFORM DISPLAY-ERROR THRU LOG-OFF
    END-EXEC.
```

```
        ...
 DISPLAY-ERROR.
        ...
 LOG-OFF.
        ...
```

In the following example, WHENEVER SQLERROR DO statements are used to handle specific errors:

```
 PROCEDURE DIVISION.
 MAIN.
        ...
     EXEC SQL
         WHENEVER SQLERROR DO PERFORM INS-ERROR
     END-EXEC.
     EXEC SQL
         INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
         VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
     END-EXEC.
     EXEC SQL
         WHENEVER SQLERROR DO PERFORM DEL-ERROR
     END-EXEC.
     EXEC SQL
         DELETE FROM DEPT
         WHERE DEPTNO = :DEPT-NUMBER
     END-EXEC.
        ...
*    Error-handling paragraphs.
 INS-ERROR.
*    Check for "duplicate key value" Oracle8 error
     IF SQLCA.SQLCODE = -1
        ...
*    Check for "value too large" Oracle8 error
     ELSE IF SQLCA.SQLCODE = -1401
        ...
     ELSE
        ...
     END-IF.
        ...
 DEL-ERROR.
*    Check for the number of rows processed.
     IF SQLCA.SQLERRD(3) = 0
        ...
     ELSE
        ...
```

```
    END-IF.
    ...
```

Notice how the paragraphs check variables in the SQLCA to determine a course of action.

Scope

Because WHENEVER is a declarative statement, its scope is positional, not logical. It tests all executable SQL statements that 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.

> **Suggestion**: You can place WHENEVER statements at the beginning of each program unit that contains SQL statements. That way, SQL statements in one program unit will not reference WHENEVER actions in another program unit, causing errors at compile or run time.

Careless Usage: Examples

Careless use of the WHENEVER statement can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets the NOT FOUND condition, because no rows meet the search condition:

```
*      Improper use of WHENEVER.
       EXEC SQL
           WHENEVER NOT FOUND GOTO NO-MORE
       END-EXEC.
       PERFORM GET-ROWS UNTIL DONE = "YES".
       ...
   GET-ROWS.
       EXEC SQL
           FETCH EMP-CURSOR INTO :EMP-NAME, :SALARY
       END-EXEC.
       ...
   NO-MORE.
       MOVE "YES" TO DONE.
       EXEC SQL
           DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER
       END-EXEC.
       ...
```

In the next example, the NOT FOUND condition is properly handled by resetting the GOTO target:

```
*       Proper use of WHENEVER.
        EXEC SQL WHENEVER NOT FOUND GOTO NO-MORE END-EXEC.
        PERFORM GET-ROWS UNTIL DONE = "YES".
        ...
 GET-ROWS.
        EXEC SQL
            FETCH EMP-CURSOR INTO :EMP-NAME, :SALARY
        END-EXEC.
        ...
 NO-MORE.
        MOVE "YES" TO DONE.
        EXEC SQL WHENEVER NOT FOUND GOTO NONE-FOUND END-EXEC.
        EXEC SQL
            DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER
        END-EXEC.
        ...
 NONE-FOUND.
        ...
```

Getting the Text of SQL Statements

In many Pro*COBOL applications, it is convenient to know the text of the statement being processed, its length, and the SQL command (such as INSERT or SELECT) that it contains. This is especially true for applications that use dynamic SQL.

The routine SQLGLS, which is part of the SQLLIB runtime library, returns the following information:

- the text of the most recently parsed SQL statement
- the length of the statement
- a function code (see Table 9–7) for the SQL command used in the statement

You can call SQLGLS after issuing a static SQL statement. With dynamic SQL Method 1, you can call SQLGLS after the SQL statement is executed. With dynamic SQL Method 2, 3, or 4, you can call SQLGLS after the statement is prepared.

To call SQLGLS, you use the following syntax:

```
CALL "SQLGLS" USING SQLSTM STMLEN SQLFC.
```

Table 9–6   shows the host-language datatypes available for the parameters in the SQLGLS argument list.

Table 9–6   Parameter Datatypes

Parameter	Datatype
SQLSTM	PIC X(*n*)
STMLEN	PIC S9(9) COMP
SQLFC	PIC S9(9) COMP

All parameters must be passed by reference. This is usually the default parameter passing convention; you need not take special action.

The parameter SQLSTM is a blank-padded (not null-terminated) character buffer that holds the returned text of the SQL statement. Your program must statically declare the buffer or dynamically allocate memory for it.

The length parameter STMLEN is a four-byte integer. Before calling SQLGLS, set this parameter to the actual size (in bytes) of the SQLSTM buffer. When SQLGLS returns, the SQLSTM buffer contains the SQL statement text blank padded to the length of the buffer. STMLEN returns the actual number of bytes in the returned statement text, not counting the blank padding. However, STMLEN returns a zero if an error occurred.

Some possible errors follow:

- No SQL statement was parsed.

- You passed an invalid parameter (for example, a negative length value).

- An internal exception occurred in SQLLIB.

The parameter SQLFC is a four-byte integer that returns the SQL function code for the SQL command in the statement. Table 9–7   shows the function code for each SQL command.

There are no SQL function codes for these statements:

- CONNECT

- COMMIT

- FETCH

- ROLLBACK

■   RELEASE

Table 9–7   SQL Codes

Code	SQL Function	Code	SQL Function
01	CREATE TABLE	39	AUDIT
02	SET ROLE	40	NOAUDIT
03	INSERT	41	ALTER INDEX
04	SELECT	42	CREATE EXTERNAL DATA-BASE
05	UPDATE	43	DROP EXTERNAL DATABASE
06	DROP ROLE	44	CREATE DATABASE
07	DROP VIEW	45	ALTER DATABASE
08	DROP TABLE	46	CREATE ROLLBACK SEGMENT
09	DELETE	47	ALTER ROLLBACK SEGMENT
10	CREATE VIEW	48	DROP ROLLBACK SEGMENT
11	DROP USER	49	CREATE TABLESPACE
12	CREATE ROLE	50	ALTER TABLESPACE
13	CREATE SEQUENCE	51	DROP TABLESPACE
14	ALTER SEQUENCE	52	ALTER SESSION
15	(not used)	53	ALTER USER
16	DROP SEQUENCE	54	COMMIT
17	CREATE SCHEMA	55	ROLLBACK
18	CREATE CLUSTER	56	SAVEPOINT
19	CREATE USER	57	CREATE CONTROL FILE
20	CREATE INDEX	58	ALTER TRACING
21	DROP INDEX	59	CREATE TRIGGER
22	DROP CLUSTER	60	ALTER TRIGGER
23	VALIDATE INDEX	61	DROP TRIGGER
24	CREATE PROCEDURE	62	ANALYZE TABLE

Table 9–7   SQL Codes

Code	SQL Function	Code	SQL Function
25	ALTER PROCEDURE	63	ANALYZE INDEX
26	ALTER TABLE	64	ANALYZE CLUSTER
27	EXPLAIN	65	CREATE PROFILE
28	GRANT	66	DROP PROFILE
29	REVOKE	67	ALTER PROFILE
30	CREATE SYNONYM	68	DROP PROCEDURE
31	DROP SYNONYM	69	(not used)
32	ALTER SYSTEM SWITCH LOG	70	ALTER RESOURCE COST
33	SET TRANSACTION	71	CREATE SNAPSHOT LOG
34	PL/SQL EXECUTE	72	ALTER SNAPSHOT LOG
35	LOCK TABLE	73	DROP SNAPSHOT LOG
36	(not used)	74	CREATE SNAPSHOT
37	RENAME	75	ALTER SNAPSHOT
38	COMMENT	76	DROP SNAPSHOT

Using the Oracle Communications Area

The SQLCA handles standard SQL communications. The Oracle Communications Area (ORACA) is a similar structure that you can include in your program to handle Oracle8-specific communications. When you need more runtime information than the SQLCA provides, use the ORACA.

Besides helping you to diagnose problems, the ORACA lets you monitor your program's use of Oracle8 resources such as the SQL Statement Executor and the *cursor cache*, an area of memory reserved for cursor management.

What's in the ORACA?

The ORACA contains option settings, system statistics, and extended diagnostics. Figure 9–3 shows all the variables in the ORACA.

Figure 9–3   ORACA Variable Declarations for Pro*COBOL

```
ORACA

01   ORACA
     05      ORACAID PIC X(8).
     05      ORACABC PIC S9(9) COMP.
     05      ORACCHF PIC S9(9) COMP.
     05      ORADBGF PIC S9(9) COMP.
     05      ORAHCHF PIC S9(9) COMP.
     05      ORASTXTF                    PIC S9(9) COMP.
     05      ORASTXT.
             49      ORASTXTL            PIC S9(4) COMP.
             49      ORASTXTL            PIC X(70)
     05      ORASFNM.
             49      ORASFNML            PIC S9(4) COMP.
             49      ORASFNMC            PIC X(70)
     05      ORASLNR PIC X(8).
     05      ORAHOC  PIC S9(9) COMP.
     05      ORAMOC  PIC S9(9) COMP.
     05      ORACOC  PIC S9(9) COMP.
     05      ORANOR  PIC S9(9) COMP.
     05      ORANPR  PIC S9(9) COMP.
     05      ORANEX  PIC S9(9) COMP.
```

Declaring the ORACA

To declare the ORACA, simply include it (using an EXEC SQL INCLUDE
statement) in your Pro*COBOL source file outside the Declare Section as follows:

```
*      Include the Oracle Communications Area (ORACA).
       EXEC SQL INCLUDE ORACA END-EXEC.
```

Enabling the ORACA

To enable the ORACA, you must set the ORACA precompiler option to YES on the
command line or in a configuration file with

```
ORACA=YES
```

or inline with

```
       EXEC Oracle OPTION (ORACA=YES) END-EXEC.
```

Then, you must choose appropriate runtime options by setting flags in the ORACA. Enabling the ORACA is optional because it adds to runtime overhead. The default setting is ORACA=NO.

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.

ORACA Structure

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, expressed in bytes, of the ORACA data structure.

ORACCHF

If the master DEBUG flag (ORADBGF) is set, this flag lets you check the cursor cache for consistency before every cursor operation.

The Oracle8 runtime library does the consistency checking and might issue error messages, which are listed in *Oracle8 Error Messages*. They are returned to the SQLCA just like Oracle8 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 Oracle8 runtime library to check the heap for consistency every time Pro*COBOL 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 all DEBUG operations (the default).
1	Enable all DEBUG operations.

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 sub-record named ORASTXT.

Diagnostics

The ORACA provides an enhanced set of diagnostics; the following variables help you to locate errors quickly.

ORASTXT

This sub-record helps you find faulty SQL statements. It lets you save the text of the last SQL statement parsed by Oracle8. 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 Pro*COBOL, such as CONNECT, FETCH, and COMMIT, are *not* saved in the ORACA.

ORASFNM

This sub-record 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

The variables below let you gather cursor cache statistics. They are automatically set by every COMMIT or ROLLBACK statement 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 Oracle8 cursors required by your program. This number can be higher than ORAHOC if MAXOPENCURSORS was set too low, which forced Pro*COBOL to extend the cursor cache.

ORACOC

This integer field records the current number of open Oracle8 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 re-parsing. For help, see Appendix D.

ORACA 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:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ORACAEX.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
        EXEC SQL INCLUDE SQLCA END-EXEC.
        EXEC SQL INCLUDE ORACA END-EXEC.

        EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
           01 USERNAME       PIC X(20).
           01 PASSWORD       PIC X(20).
           01 EMP-NAME       PIC X(10) VARYING.
           01 DEPT-NUMBER    PIC S9(4) COMP.
           01 SALARY         PIC S9(6)V99
                             DISPLAY SIGN LEADING SEPARATE.
        EXEC SQL END DECLARE SECTION END-EXEC.

     PROCEDURE DIVISION.
        DISPLAY "Username? " WITH NO ADVANCING.
        ACCEPT USERNAME.
        DISPLAY "Password? " WITH NO ADVANCING.
        ACCEPT PASSWORD.
        EXEC SQL
            WHENEVER SQLERROR GOTO SQL-ERROR
        END-EXEC.
        EXEC SQL
            CONNECT :USERNAME IDENTIFIED BY :PASSWORD
        END-EXEC.
        DISPLAY "Connected to Oracle".

  * -- set flags in the ORACA
  * -- enable debug operations
        MOVE 1 TO ORADBGF.
  * -- enable cursor cache consistency check
        MOVE 1 TO ORACCHF.
  * -- always save the SQL statement
        MOVE 3 TO ORASTXTF.
        DISPLAY "Department number? " WITH NO ADVANCING.
        ACCEPT DEPT-NUMBER.
        EXEC SQL DECLARE EMPCURSOR CURSOR FOR
            SELECT ENAME, SAL + NVL(COMM,0)
            FROM EMP
            WHERE DEPTNO = :DEPT-NUMBER
        END-EXEC.
        EXEC SQL OPEN EMPCURSOR END-EXEC.
        EXEC SQL
            WHENEVER NOT FOUND GOTO NO-MORE
        END-EXEC.
```

```
        LOOP.
            EXEC SQL
                FETCH EMPCURSOR INTO :EMP-NAME, :SALARY
            END-EXEC.
            IF SALARY < 2500
                EXEC SQL
                    INSERT INTO PAY1 VALUES (:EMP-NAME, :SALARY)
                END-EXEC
            ELSE
                EXEC SQL
                    INSERT INTO PAY2 VALUES (:EMP-NAME, :SALARY)
                END-EXEC
            END-IF.
            GO TO LOOP.

        NO-MORE.
            EXEC SQL CLOSE EMPCURSOR END-EXEC.
            EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
            EXEC SQL COMMIT WORK RELEASE END-EXEC.
            DISPLAY "(NO-MORE.) Last SQL statement: ", ORASTXTC.
            DISPLAY "... at or near line number:  ", ORASLNR.
            DISPLAY  " ".
            DISPLAY "          Cursor Cache Statistics".
            DISPLAY "-----------------------------------------".
            DISPLAY "Maximum value of MAXOPENCURSORS     ", ORAHOC.
            DISPLAY "Maximum open cursors required:      ", ORAMOC.
            DISPLAY "Current number of open cursors:     ", ORACOC.
            DISPLAY "Number of cache reassignments:      ", ORANOR.
            DISPLAY "Number of SQL statement parses:     ", ORANPR.
            DISPLAY "Number of SQL statement executions: ", ORANEX.
            STOP RUN.

        SQL-ERROR.
            EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
            EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
            DISPLAY "(SQL-ERROR.) Last SQL statement: ", ORASTXTC.
            DISPLAY "... at or near line number:  ", ORASLNR.
            DISPLAY " ".
            DISPLAY "          Cursor Cache Statistics".
            DISPLAY "-----------------------------------------".
            DISPLAY "MAXIMUM VALUE OF MAXOPENCURSORS     ", ORAHOC.
            DISPLAY "Maximum open cursors required:      ", ORAMOC.
            DISPLAY "Current number of open cursors:     ", ORACOC.
            DISPLAY "Number of cache reassignments:      ", ORANOR.
            DISPLAY "Number of SQL statement parses:     ", ORANPR.
```

```
DISPLAY "Number of SQL statement executions: ", ORANEX.
STOP RUN.
```

10

Using Host Tables

This chapter looks at using tables to simplify coding and improve program performance. You learn how to manipulate Oracle8 data using tables, how to operate on all the elements of an table with a single SQL statement, and how to limit the number of table elements processed. Topics are:

- What Is a Host Table?
- Why Use Tables?
- Declaring Host Tables
- Using Tables in SQL Statements
- Selecting into Tables
- Inserting with Tables
- Updating with Tables
- Deleting with Tables
- Using Indicator Tables
- Using the FOR Clause
- Using the WHERE Clause
- Mimicking the CURRENT OF Clause
- Using SQLERRD(3)
- Sample Program 3: Fetching in Batches

What Is a Host Table?

An *table* is a set of related data items, called *elements*, associated with a single variable name. When declared as a host variable, the table is called a *host table*. Likewise, an indicator variable declared as an table is called an *indicator table*. An indicator table can be associated with any host table.

Why Use Tables?

Tables 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. Tables simplify the task of naming and referencing the individual items in each collection.

Using tables can boost the performance of your application. Tables let you manipulate an entire collection of data items with a single SQL statement. Thus, Oracle8 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 tables your program must do 300 individual INSERTs—one for each employee. With tables, only one INSERT need be done.

Declaring Host Tables

You declare host tables in the Data Division like simple host variables. You also *dimension* (set the size of) host tables in the Data Division. In the following example, you declare three host tables and dimension them with 50 elements:

```
    ...
 DATA DIVISION.
 WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-REC-TABLES.
    05  EMP-NUMBER   OCCURS 50 TIMES PIC S9(4) COMP.
    05  EMP-NAME     OCCURS 50 TIMES PIC X(10) VARYING.
    05  SALARY       OCCURS 50 TIMES PIC S9(6)V99
                        DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
```

Dimensioning Tables

The maximum dimension of a host table is 32,767 elements. If you use a host table that exceeds the maximum, you get a "parameter out of range" runtime error. If you use multiple host tables in a single SQL statement, their dimensions should be the same. Otherwise, an "table size mismatch" warning message is issued at pre-compile time. If you ignore this warning, the precompiler uses the *smallest* dimension for the SQL operation.

Restrictions

Host tables that might be referenced in a SQL statement are limited to one dimension. So, the two-dimensional table declared in the following example is *invalid*:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  SALARY-TABLE.
    05  ROW  OCCURS 25 TIMES.
        10  COLUMN OCCURS 25 TIMES.
            HI-LOW-SCORES  PIC 9(5).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Using Tables in SQL Statements

Pro*COBOL allows the use of host tables in data manipulation statements. You can use host tables as input variables in the INSERT, UPDATE, and DELETE statements and as output variables in the INTO clause of SELECT and FETCH statements.

The syntax used for host tables and simple host variables is nearly the same. One difference is the optional FOR clause, which lets you control table processing. Also, there are restrictions on mixing host tables and simple host variables in a SQL statement.

The following sections illustrate the use of host tables in data manipulation statements.

Selecting into Tables

You can use host tables as output variables in the SELECT statement. If you know the maximum number of rows the select will return, simply dimension the host tables with that number of elements. In the following example, you select directly into three host tables. Knowing the select will return no more than 50 rows, you dimension the tables with 50 elements:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

```
01  EMP-REC-TABLES.
      05  EMP-NUMBER   OCCURS 50 TIMES PIC S9(4) COMP.
   05 EMP-NAME     OCCURS 50 TIMES PIC X(10) VARYING.
   05  SALARY      OCCURS 50 TIMES PIC S9(6)V99
                   DISPLAY SIGN LEADING SEPARATE.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT ENAME, EMPNO, SAL
    INTO :EMP-NAME, :EMP-NUMBER, :SALARY
    FROM EMP
    WHERE SAL > 1000
END-EXEC.
```

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 re-execute the SELECT statement, it just returns the first 50 rows again, even if more are eligible. You must either dimension a larger table 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, Oracle8 issues the error message

```
SQL-02112: SELECT...INTO returns too many rows
```

unless you specify SELECT_ERROR=NO. For more information about the option SELECT_ERROR, see "SELECT_ERROR" on page 7-34.

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 END-EXEC.
01  EMP-REC-TABLES.
    05  EMP-NUMBER    OCCURS 20 TIMES PIC S9(4) COMP.
    05  EMP-NAME      OCCURS 20 TIMES PIC X(10) VARYING.
    05  SALARY        OCCURS 20 TIMES PIC S9(6)V99
                      DISPLAY SIGN LEADING SEPARATE.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
```

```
      EXEC SQL DECLARE EMPCURSOR CURSOR FOR
      SELECT EMPNO, SAL FROM EMP
      END-EXEC.
      ...
      EXEC SQL OPEN EMPCURSOR END-EXEC.
      ...
      EXEC SQL WHENEVER NOT FOUND DO PERFORM ...
 LOOP.
      EXEC SQL FETCH EMPCURSOR INTO :EMP-NUMBER, :SALARY END-EXEC.
* --  process batch of rows
      ...
      GO TO LOOP.
```

Number of Rows Fetched

Each fetch returns, at most, the number of rows in the table dimension. Fewer rows are returned in the following cases:

- The end of the active set is reached. The "no data found" Oracle8 warning code is returned to SQLCODE in the SQLCA. For example, this happens if you fetch into an table 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 table 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 Oracle8 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 END-EXEC.
      EXEC SQL OPEN CURSOR2 END-EXEC.
      EXEC SQL FETCH CURSOR1 INTO :TABLE-OF-20 END-EXEC.
* --  now running total in SQLERRD(3) is 20
      EXEC SQL FETCH CURSOR2 INTO :TABLE-OF-30 END-EXEC.
* --  now running total in SQLERRD(3) is 30, not 50
      EXEC SQL FETCH CURSOR1 INTO :TABLE-OF-20 END-EXEC.
* --  now running total in SQLERRD(3) is 40 (20 + 20)
      EXEC SQL FETCH CURSOR2 INTO :TABLE-OF-30 END-EXEC.
* --  now running total in SQLERRD(3) is 60 (30 + 30)
```

Restrictions

Using host tables in the WHERE clause of a SELECT statement is allowed only in a subquery. (For an example, see "Using the WHERE Clause" on page 10-13.) Also, you cannot mix simple host variables with host tables in the INTO clause of a SELECT or FETCH statement; if any of the host variables is a table, all must be tables.

Table 10–1 shows which uses of host tables are valid in a SELECT INTO statement:

Table 10–1   Host Tables Valid in SELECT INTO

INTO Clause	WHERE Clause	Valid?
table	table	no
scalar	scalar	yes
table	scalar	yes
scalar	table	no

Fetching Nulls

When UNSAFE_NULL=YES, if you select or fetch a null into a host table that lacks an indicator table, no error is generated. So, when doing table selects and fetches, always use indicator tables. That way, you can find nulls in the associated output host table. (To learn how to find nulls and truncated values, see "Using Indicator Variables" on page 5-3.)

When UNSAFE_NULL=NO, if you select or fetch a null into a host table that lacks an indicator table, Oracle8 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 DBMS=V7 or V8, if you select or fetch a truncated column value into a host table that lacks an indicator table, Oracle8 stops processing, sets SQLERRD(3) to the number of rows processed, and issues the following error message:

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

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 Oracle8 continues processing.

Again, when doing table selects and fetches, always use indicator tables. That way, if Oracle8 assigns one or more truncated column values to an output host table, you can find the original lengths of the column values in the associated indicator table.

Inserting with Tables

You can use host tables as input variables in an INSERT statement. Just make sure your program populates the tables with data before executing the INSERT statement. If some elements in the tables are irrelevant, you can use the FOR clause to control the number of rows inserted. See "Using the FOR Clause" on page 10-11.

An example of inserting with host tables follows:

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-REC-TABLES.
    05  EMP-NUMBER    OCCURS 50 TIMES PIC S9(4) COMP.
    05  EMP-NAME      OCCURS 50 TIMES PIC X(10) VARYING.
    05  SALARY        OCCURS 50 TIMES PIC S9(6)V99
                      DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host tables
    EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
        VALUES (:EMP-NAME, :EMP-NUMBER, :SALARY)
    END-EXEC.
```

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 Oracle8:

```
    PERFORM VARYING I FROM 1 BY 1 UNTIL I = TABLE-DIMENSION.
       EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
           VALUES (:EMP-NAME[I], :EMP-NUMBER[I], :SALARY[I])
       END_EXEC
    END-PERFORM.
```

In this imaginary example (imaginary because host variables *cannot* be subscripted in a SQL statement), you use a FOR loop to access all table elements in sequential order.

Restrictions

Mixing simple host variables with host tables in the VALUES clause of an INSERT statement is *not* allowed; if any of the host variables is a table, all must be tables.

Updating with Tables

You can also use host tables as input variables in an UPDATE statement, as the following example shows:

```
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-REC-TABLES.
    05  EMP-NUMBER    OCCURS 50 TIMES PIC S9(4) COMP.
    05  SALARY        OCCURS 50 TIMES PIC S9(6)V99
                          DISPLAY SIGN LEADING SEPARATE.

  EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host tables
  EXEC SQL
       UPDATE EMP SET SAL = :SALARY WHERE EMPNO = :EMP-NUMBER
    END-EXEC.
```

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 tables 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 table element qualified just one row for updating. In the following example, each table element qualifies multiple rows:

```
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    ...
         05  JOB-TITLE     OCCURS 10 TIMES PIC X(10) VARYING.
         05  COMMISSION    OCCURS 50 TIMES PIC S9(6)V99
                          DISPLAY SIGN LEADING SEPARATE.
  EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host tables
  EXEC SQL
       UPDATE EMP SET COMM = :COMMISSION WHERE JOB = :JOB-TITLE
```

```
        END-EXEC.
```

Restrictions

Mixing simple host variables with host tables in the SET or WHERE clause of an UPDATE statement is *not* allowed. If any of the host variables is an table, all must be tables. Furthermore, if you use a host table in the SET clause, you *must* use one in the WHERE clause. However, their dimensions and datatypes need not match.

You cannot use host tables with the CURRENT OF clause in an UPDATE statement. For an alternative, see "Mimicking the CURRENT OF Clause" on page 10-14.

Table 10–2 shows which uses of host tables are valid in an UPDATE statement:

Table 10–2   Host Tables Valid in UPDATE

SET Clause	WHERE Clause	Valid?
table	table	yes
scalar	scalar	yes
table	scalar	no
scalar	table	no

Deleting with Tables

You can also use host tables as input variables in a DELETE statement. It is like executing the DELETE statement repeatedly using successive elements of the host table in the WHERE clause. Thus, each execution might delete zero, one, or more rows from the table. An example of deleting with host tables follows:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      ...
         05  EMP-NUMBER    OCCURS 50 TIMES PIC S9(4) COMP.
   EXEC SQL END DECLARE SECTION END-EXEC.
*  --  populate the host table
   EXEC SQL
         DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER
   END-EXEC.
```

The cumulative number of rows deleted can be found in SQLERRD(3). That number does *not* include rows processed by a delete cascade.

The last example showed a typical delete using a unique key (*EMP-NUMBER*). Each table element qualified just one row for deletion. In the following example, each table element qualifies multiple rows:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   ...
          05  JOB-TITLE    OCCURS 10 TIMES PIC X(10) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host table
EXEC SQL
      DELETE FROM EMP WHERE JOB = :JOB-TITLE
   END-EXEC.
```

Restrictions

Mixing simple host variables with host tables in the WHERE clause of a DELETE statement is *not* allowed; if any of the host variables is a table, all must be tables. Also, you cannot use host tables with the CURRENT OF clause in a DELETE statement. For an alternative, see "Mimicking CURRENT OF" on "Mimicking the CURRENT OF Clause" on page 10-14.

Using Indicator Tables

You use indicator tables to assign nulls to input host tables and to detect null or truncated values in output host tables. The following example shows how to insert with indicator tables:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-REC-VARS.
    05  EMP-NUMBER  OCCURS 50 TIMES PIC S9(4) COMP.
    05  DEPT-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.
    05  COMMISSION  OCCURS 50 TIMES  PIC S9(6)V99
                          DISPLAY SIGN LEADING SEPARATE.
* -- indicator table:
     05  COMM-IND    OCCURS 50 TIMES  PIC S9(4) COMP.
 EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host tables
* --  populate the indicator table; to insert a null into
* --  the COMM column, assign -1 to the appropriate element in
* --  the indicator table
 EXEC SQL
      INSERT INTO EMP (EMPNO, DEPTNO, COMM)
      VALUES (:EMP_NUMBER, :DEPT-NUMBER, :COMMISSION:COMM-IND)
   END-EXEC.
```

The dimension of the indicator table cannot be smaller than the dimension of the host table.

Using the FOR Clause

You can use the optional FOR clause to set the number of table 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 table. 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 END-EXEC.
    01  EMP-REC-VARS.
        05  EMP-NAME  OCCURS 1000 TIMES  PIC X(20) VARYING.
        05  SALARY    OCCURS 100  TIMES  PIC S9(6)V99
                    DISPLAY SIGN LEADING SEPARATE.
    01 ROWS-TO-INSERT PIC S9(4) COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host tables
    MOVE 25 TO ROWS-TO-INSERT.
* -- set FOR-clause variable
* -- will process only 25 rows
    EXEC SQL FOR :ROWS-TO-INSERT
        INSERT INTO EMP (ENAME, SAL)
        VALUES (:EMP-NAME, :SALARY)
    END-EXEC.
```

The FOR clause must use an integer host variable to count table elements. For example, the following statement is illegal:

```
* -- illegal
    EXEC SQL FOR 25
```

```
INSERT INTO EMP (ENAME, EMPNO, SAL)
    VALUES (:EMP-NAME, :EMP-NUMBER, :SALARY)
END-EXEC.
```

The FOR-clause variable specifies the number of table elements to be processed.
Make sure the number does not exceed the smallest table dimension. Also, the num-
ber must be positive. If it is negative or zero, no rows are processed.

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 mes-
sage:

```
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 fol-
lows:

```
EXEC SQL FOR :LIMIT FETCH EMPCURSOR 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 EMPCURSOR CURSOR FOR
    SELECT ENAME, SAL FROM EMP WHERE EMPNO = :EMP-NUMBER
END-EXEC.
...
EXEC SQL OPEN EMPCURSOR END-EXEC.
...
EXEC SQL FETCH emp_cursor INTO :EM-NAME, :SALARY END-EXEC.
...
EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
    WHERE CURRENT OF EMPCURSOR
END-EXEC.
```

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 UPDA-CURSOR END-EXEC.
...
EXEC SQL FOR :LIMIT DELETE FROM EMP
    WHERE CURRENT OF EMP-CURSOR
END-EXEC.
```

Using the WHERE Clause

Oracle8 treats a SQL statement containing host tables of dimension *n* like the same SQL statement executed *n* times with *n* different scalar variables (the individual table elements). The precompiler issues the following error message only when such treatment is ambiguous:

```
PCC-S-0055: Array <name> not allowed as bind variable at ...
```
For example, assuming the declarations

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
    05  MGRP-NUMBER  OCCURS 50 TIMES  PIC S9(4) COMP.
    05  JOB-TITLE    OCCURS 50 TIMES  PIC X(20) VARYING.
01  I PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```

it would be ambiguous if the statement

```
EXEC SQL SELECT MGR INTO :MGR-NUMBER FROM EMP
    WHERE JOB = :JOB-TITLE
END-EXEC.
```

were treated like the imaginary statement

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 50
SELECT MGR INTO :MGR-NUMBER[I] FROM EMP
    WHERE JOB = :JOB_TITLE[I]
END-EXEC
END-PERFORM.
```

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)
END-EXEC.
```

were treated like the imaginary statement

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 50
    UPDATE EMP SET MGR = :MGR_NUMBER[I]
        WHERE EMPNO IN
        (SELECT EMPNO FROM EMP WHERE JOB = :JOB-TITLE[I])
    END-EXEC
END-PERFORM.
```

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*. So, no error message is issued.

Mimicking the CURRENT OF Clause

You use the CURRENT OF *cursor* clause in a DELETE or UPDATE statement to refer to the latest row fetched from the cursor. However, you cannot use CURRENT OF with host tables. 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 END-EXEC.
    05  EMP-NAME    OCCURS 25 TIMES PIC X(20) VARYING.
    05  JOB-TITLE   OCCURS 25 TIMES PIC X(15) VARYING.
    05  OLD-TITLE   OCCURS 25 TIMES PIC X(15) VARYING.
    05  ROW-ID      OCCURS 25 TIMES PIC X(18) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
    SELECT ENAME, JOB, ROWID FROM EMP
END-EXEC.
...
EXEC SQL OPEN EMPCURSOR END-EXEC.
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
...
PERFORM
```

```
            EXEC SQL
                FETCH EMPCURSOR
                INTO :EMP-NAME, :JOB-TITLE, :ROW-I
            END-EXEC
            ...
            EXEC SQL
                DELETE FROM EMP
                WHERE JOB = :OLD-TITLE AND ROWID = :ROW-ID
            END-EXEC
            EXEC SQL COMMIT WORK END-EXEC
        END-PERFORM.
```

However, the fetched rows are *not* locked because no FOR UPDATE OF clause is used. 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, and DELETE statements, SQLERRD(3) records the number of rows processed.

SQLERRD(3) is also useful when an error occurs during a table operation. Processing stops at the row that caused the error, so SQLERRD(3) gives the number of rows processed successfully.

Sample Program 3: Fetching in Batches

This program logs on to Oracle8, declares and opens a cursor, fetches in batches using host tables, and prints the results using the PRINT-IT paragraph.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HOST-TABLES.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME          PIC X(15) VARYING.
        01  PASSWD            PIC X(15) VARYING.
        01  EMP-REC-TABLES.
            05  EMP-NUMBER    OCCURS 5 TIMES PIC S9(4) COMP.
            05  EMP-NAME      OCCURS 5 TIMES PIC X(10) VARYING.
            05  SALARY        OCCURS 5 TIMES PIC S9(6)V99
                              DISPLAY SIGN LEADING SEPARATE.
```

```
                    EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
                    EXEC SQL END DECLARE SECTION END-EXEC.

                    EXEC SQL INCLUDE SQLCA END-EXEC.
                        01  NUM-RET          PIC S9(9) COMP VALUE ZERO.
                        01  PRINT-NUM        PIC S9(9) COMP VALUE ZERO.
                        01  COUNTER          PIC S9(9) COMP.
                        01  DISPLAY-VARIABLES.
                            05  D-EMP-NAME    PIC X(10).
                            05  D-EMP-NUMBER  PIC 9(4).
                            05  D-SALARY      PIC Z(4)9.99.

               PROCEDURE DIVISION.

               BEGIN-PGM.
                   EXEC SQL
                       WHENEVER SQLERROR DO PERFORM SQL-ERROR
                   END-EXEC.
                   PERFORM LOGON.
                   EXEC SQL
                       DECLARE C1 CURSOR FOR
                       SELECT EMPNO, SAL, ENAME FROM EMP
                   END-EXEC.
                   EXEC SQL
                       OPEN C1
                   END-EXEC.

               FETCH-LOOP.
                   EXEC SQL
                       WHENEVER NOT FOUND DO PERFORM SIGN-OFF
                   END-EXEC.
                   EXEC SQL
                       FETCH C1 INTO :EMP-NUMBER, :SALARY, :EMP-NAME
                   END-EXEC.
                   SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
                   PERFORM PRINT-IT.
                   MOVE SQLERRD(3) TO NUM-RET.
                   GO TO FETCH-LOOP.

               LOGON.
                   MOVE "SCOTT" TO USERNAME-ARR.
                   MOVE 5 TO USERNAME-LEN.
                   MOVE "TIGER" TO PASSWD-ARR.
                   MOVE 5 TO PASSWD-LEN.
                   EXEC SQL
```

```
            CONNECT :USERNAME IDENTIFIED BY :PASSWD
        END-EXEC.
        DISPLAY " ".
        DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

    PRINT-IT.
        DISPLAY " ".
        DISPLAY "EMPLOYEE NUMBER   SALARY   EMPLOYEE NAME".
        DISPLAY "---------------   ------   ------------".
        PERFORM PRINT-ROWS
            VARYING COUNTER FROM 1 BY 1 UNTIL COUNTER > PRINT-NUM.

    PRINT-ROWS.
        MOVE EMP-NUMBER(COUNTER) TO D-EMP-NUMBER.
        MOVE SALARY(COUNTER) TO D-SALARY.
        DISPLAY D-EMP-NUMBER, "              ", D-SALARY, "    ",
            EMP-NAME-ARR IN EMP-NAME(COUNTER).
        MOVE SPACES TO EMP-NAME-ARR IN EMP-NAME(COUNTER).

    SIGN-OFF.
        SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
        IF (PRINT-NUM > 0) PERFORM PRINT-IT.
        EXEC SQL CLOSE C1 END-EXEC.
        EXEC SQL COMMIT WORK RELEASE END-EXEC.
        DISPLAY " ".
        DISPLAY "HAVE A GOOD DAY.".
        DISPLAY " ".
        STOP RUN.

    SQL-ERROR.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        DISPLAY " ".
        DISPLAY "ORACLE ERROR DETECTED:".
        DISPLAY " ".
        DISPLAY SQLERRMC.
        EXEC SQL
            ROLLBACK WORK RELEASE
        END-EXEC.
        STOP RUN.
```

11

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.

Topics are:

- What Is Dynamic SQL?

- Advantages and Disadvantages of Dynamic SQL

- When to Use Dynamic SQL

- Requirements for Dynamic SQL Statements

- How Dynamic SQL Statements Are Processed

- Methods for Using Dynamic SQL

- Using Method 1

- Sample Program 6: Dynamic SQL Method 1

- Using Method 2

- Sample Program 7: Dynamic SQL Method 2

- Using Method 3

- Sample Program 8: Dynamic SQL Method 3

- Using Method 4

- Using the DECLARE STATEMENT Statement

- Using Host Tables
- Using PL/SQL

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 precompile 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 or more of the following items is unknown at precompile 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, user-names, 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

In most 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 place-holders, you do not declare them and can name them anything you like (hyphens are not allowed). For example, Oracle8 makes no distinction between the following two strings:

```
'DELETE FROM EMP WHERE MGR = :MGRNUMBER AND JOB = :JOBTITLE'
'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 Oracle8 *parses* the SQL statement. That is, Oracle8 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, Oracle8 *binds* the host variables to the SQL statement. That is, Oracle8 gets the addresses of the host variables so that it can read or write their values.

Then Oracle8 *executes* the SQL statement. That is, Oracle8 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 this 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 Table 11–1 shows:

Table 11–1   Appropriate Method to Use

Method	Kind of SQL Statement
1	non-query without input host variables
2	non-query 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

The term *select-list item* includes column names and expressions.

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 (unless you specify HOLD_CURSOR=YES).

Method 2

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 place-holders for input host variables and the datatypes of the input host variables must be known at precompile time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:EMPNAME, :JOBTITLE)'
'DELETE FROM EMP WHERE EMPNO = :EMPNUMBER'
```

With Method 2, the SQL statement is parsed just once (unless you specify RELEASE_CURSOR=YES), but it can be executed many times with different values for the host variables. SQL data definition statements such as CREATE 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 commands. The number of select-list items, the number of place-holders for input host variables, and the datatypes of the input host variables must be known at precompile 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 = :DEPTNUMBER'
```

Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors (discussed in "Using Method 4" on page 11-25). The number of select-list items, the number of place-holders 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 place-holders for input host variables and the datatypes of the input host variables must be known at precompile 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 re-parsing 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 11–1 will help you choose the correct 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 re-initialize) the host string before storing the SQL statement.

Do not null-terminate the host string. Oracle8 does not recognize the null terminator as an end-of-string sentinel. Instead, Oracle8 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.

EXECUTE resets the SQLWARN warning flags in the SQLCA. So, to catch mistakes such as an unconditional update (caused by omitting a WHERE clause), check the SQLWARN flags after executing the PREPARE statement but before executing the EXECUTE statement.

Figure 11–1   Choosing the Right Method

About the SQL statement...

Is it a query?

Does its select list contain an unknown number of items?

yes

yes

no

no

Does it contain input host variables?

yes

Does it contain an unknown number of input host variables?

yes

no

no

Does it contain an unknown number of input host variables?

yes

no

Will it be executed repeatedly?

yes

no

Method 1   **Method 2**   **Method 3**   **Method 4**
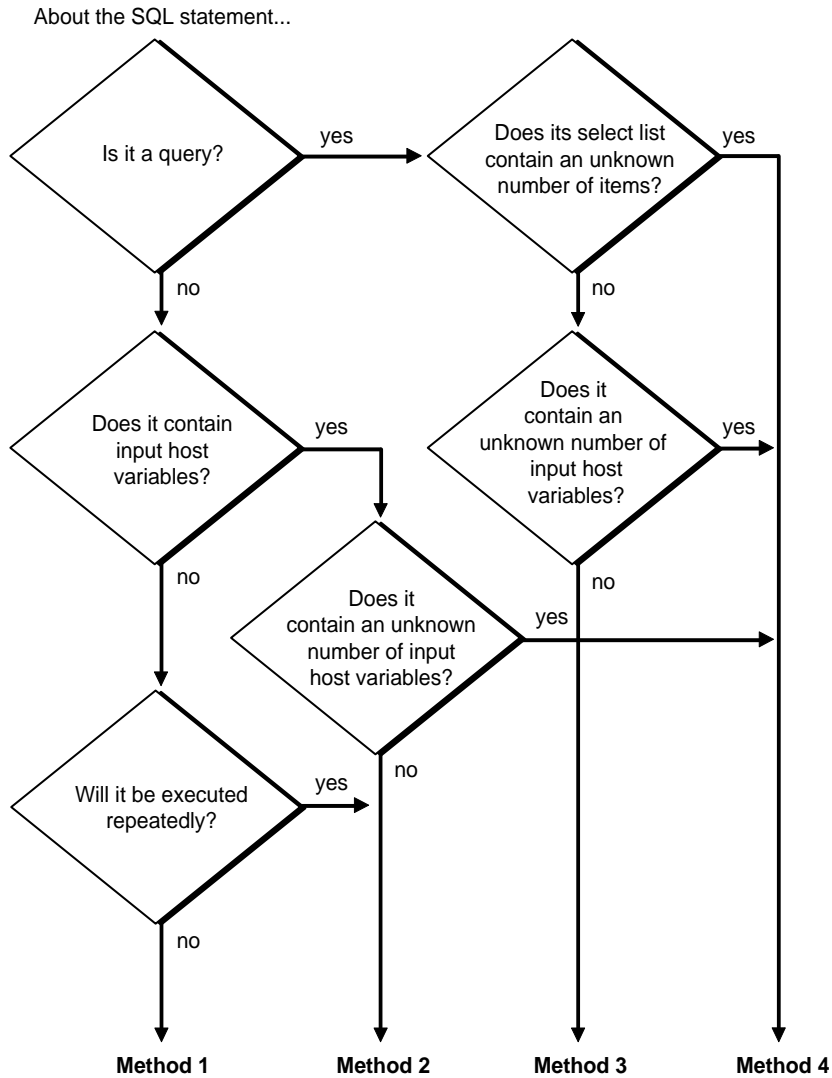
Using Method 1

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'
```

The EXECUTE IMMEDIATE Statement

Method 1 parses, then immediately executes the SQL statement using the EXE-CUTE 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 }END-EXEC.
```

In the following example, you use the host variable *SQL-STMT* to store SQL statements input by the user:

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
     01   SQL-STMT  PIC X(120);
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 LOOP.
     DISPLAY 'Enter SQL statement: ' WITH NO ADVANCING.
     ACCEPT SQL-STMT END-EXEC.
* --  sql_stmt now contains the text of a SQL statement
     EXEC SQL EXECUTE IMMEDIATE :SQL-STMT END-EXEC.
 NEXT.
     ...
```

You can also use string literals, as the following example shows:

```
     EXEC SQL
         EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER'
     END-EXEC.
```

Because EXECUTE IMMEDIATE parses the input SQL statement before every execution, Method 1 is best for statements that are executed only once. Data definition statements usually fall into this category.

An Example

The following fragment of a 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:

```
      ...
*     THE RELEASE_CURSOR=YES OPTION INSTRUCTS PRO*COBOL TO
*     RELEASE IMPLICIT CURSORS ASSOCIATED WITH EMBEDDED SQL
*     STATEMENTS.  THIS ENSURES THAT Oracle8 DOES NOT KEEP PARSE
*     LOCKS ON TABLES, SO THAT SUBSEQUENT DATA MANIPULATION
*     OPERATIONS ON THOSE TABLES DO NOT RESULT IN PARSE-LOCK
*     ERRORS.

      EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.

*
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
          01  USERNAME  PIC X(10) VALUE "SCOTT".
          01  PASSWD    PIC X(10) VALUE "TIGER".
          01  DYNSTMT   PIC X(80).
      EXEC SQL END DECLARE SECTION END-EXEC.
          01  UPDATESTMT PIC X(40).
          01  SEARCH-COND PIC X(40).
      ...
      DISPLAY "ENTER A SEARCH CONDITION FOR STATEMENT:".
      MOVE "UPDATE EMP SET COMM = 500 WHERE " TO UPDATESTMT.
      DISPLAY UPDATESTMT.
      ACCEPT SEARCH-COND.
*     Concatenate SEARCH-COND to UPDATESTMT and store result
*     in DYNSTMT.
      STRING UPDATESTMT DELIMITED BY SIZE
          SEARCH-COND DELIMITED BY SIZE INTO DYNSTMT.
      EXEC SQL EXECUTE IMMEDIATE :DYNSTMT END-EXEC.
```

Sample Program 6: Dynamic SQL Method 1

This program uses dynamic SQL Method 1 to create a table, insert a row, commit the insert, then drop the table.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID.  DYNSQL1.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        WORKING-STORAGE SECTION.

*       INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
*       THROUGH WHICH ORACLE8 MAKES ADDITIONAL RUNTIME STATUS
*       INFORMATION AVAILABLE TO THE PROGRAM.
        EXEC SQL INCLUDE SQLCA END-EXEC.

        EXEC SQL INCLUDE ORACA END-EXEC.
*       THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
*       THE ORACA.
        EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

*       THE RELEASE_CURSOR=YES OPTION INSTRUCTS PRO*COBOL TO
*       RELEASE IMPLICIT CURSORS ASSOCIATED WITH EMBEDDED SQL
*       STATEMENTS.  THIS ENSURES THAT ORACLE DOES NOT KEEP
*       PARSE LOCKS ON TABLES, SO THAT SUBSEQUENT DATA
*       MANIPULATION OPERATIONS ON THOSE TABLES DO NOT RESULT
*       IN PARSE-LOCK ERRORS.
        EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.

*       ALL HOST VARIABLES USED IN EMBEDDED SQL MUST APPEAR IN
*       THE DECLARE SECTION.
        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
            01  USERNAME  PIC X(10) VALUE "SCOTT".
            01  PASSWD    PIC X(10) VALUE "TIGER".
            01  DYNSTMT   PIC X(80) VARYING.
        EXEC SQL END DECLARE SECTION END-EXEC.

*       DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
            01  ORASLNRD  PIC 9(9).

        PROCEDURE DIVISION.

        MAIN.

*       BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
        EXEC SQL
            WHENEVER SQLERROR GOTO SQLERROR
        END-EXEC.

*       SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN
```

```
*      ERROR OCCURS.
       MOVE 1 TO ORASTXTF.

*      CONNECT TO ORACLE.
       EXEC SQL
           CONNECT :USERNAME IDENTIFIED BY :PASSWD
       END-EXEC.
       DISPLAY " ".

       DISPLAY "CONNECTED TO ORACLE.".
       DISPLAY " ".

*      EXECUTE A STRING LITERAL TO CREATE THE TABLE.  HERE,
*      YOU GENERALLY USE A STRING VARIABLE INSTEAD OF A
*      LITERAL, AS IS DONE LATER IN THIS PROGRAM.  BUT, YOU
*      CAN USE A LITERAL IF YOU WISH.
       DISPLAY "CREATE TABLE DYN1 (COL1 CHAR(4))".
       EXEC SQL
           EXECUTE IMMEDIATE "CREATE TABLE DYN1 (COL1 CHAR(4))"
       END-EXEC.

*      ASSIGN A SQL STATEMENT TO THE VARYING STRING DYNSTMT.
*      SET THE -LEN PART TO THE LENGTH OF THE -ARR PART.
       MOVE "INSERT INTO DYN1 VALUES ('TEST')"
         TO DYNSTMT-ARR.
       MOVE 36 TO DYNSTMT-LEN.
       DISPLAY DYNSTMT-ARR.

*      EXECUTE DYNSTMT TO INSERT A ROW.  THE SQL STATEMENT IS
*      A STRING VARIABLE WHOSE CONTENTS THE PROGRAM MAY
*      DETERMINE AT RUN TIME.
       EXEC SQL
           EXECUTE IMMEDIATE :DYNSTMT
       END-EXEC.

*      COMMIT THE INSERT.
       EXEC SQL
           COMMIT WORK
       END-EXEC.

*      CHANGE DYNSTMT AND EXECUTE IT TO DROP THE TABLE.
       MOVE "DROP TABLE DYN1" TO DYNSTMT-ARR.
       MOVE 19 TO DYNSTMT-LEN.
       DISPLAY DYNSTMT-ARR.
       EXEC SQL
```

```
              EXECUTE IMMEDIATE :DYNSTMT
          END-EXEC.

*     COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
          EXEC SQL
              COMMIT RELEASE
          END-EXEC.
          DISPLAY " ".
          DISPLAY "HAVE A GOOD DAY!".
          DISPLAY " ".
          STOP RUN.

 SQLERROR.
*     ORACLE ERROR HANDLER.  PRINT DIAGNOSTIC TEXT CONTAINING
*     ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF
*     ERROR.
          DISPLAY SQLERRMC.
          DISPLAY "IN ", ORASTXTC.
          MOVE ORASLNR TO ORASLNRD.
          DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

*     DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
*     SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
          EXEC SQL
              WHENEVER SQLERROR CONTINUE
          END-EXEC.

*     ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM
*     ORACLE.
          EXEC SQL
              ROLLBACK RELEASE
          END-EXEC.
          STOP RUN.
```

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 place-holders 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. Also, you need *not*

re-prepare the SQL statement after a COMMIT or ROLLBACK (unless you log off and reconnect).

Note that you can use EXECUTE for non-queries with Method 4.

The syntax of the PREPARE statement follows:

```
EXEC SQL PREPARE <STATEMENT-NAME>
    FROM { :<HOST-STRING> | <STRING-LITERAL> }
END-EXEC.
```

PREPARE parses the SQL statement and gives it a name.

STATEMENT-NAME is an identifier used by the precompiler, *not* a host or program variable, and should not be declared in a COBOL statement. 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>]
END-EXEC.
```

where *HOST-VARIABLE-LIST* stands for the following syntax:

```
:<HOST-VAR1>[:<INDICATOR1>] [, <HOST-VAR2>[:<INDICATOR2>], ...]
```

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 place-holder *n*:

```
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      ...
      01  EMP-NUMBER   PIC S9(4) COMP VALUE ZERO.
      ...
      01 DELETE-STMT   PIC X(120) VALUE SPACES.
      ...
      EXEC SQL END DECLARE SECTION END-EXEC.
      01  WHERE-STMT   PIC X(40).
      01  SEARCH-COND  PIC X(40).
      ...
      MOVE 'DELETE FROM EMP WHERE EMPNO = :N AND ' TO WHERE-STMT.
      DISPLAY 'Complete this statement's search condition:'.
      DISPLAY WHERE-STMT.
      ACCEPT SEARCH-COND.
*     Concatenate SEARCH-COND to WHERE-STMT and store in DELETE-STMT
      STRING WHERE-STMT DELIMITED BY SIZE
```

```
              SEARCH-COND DELIMITED BY SIZE INTO
              DELETE-STMT.
        EXEC SQL PREPARE SQLSTMT FROM :DELETE-STMT END-EXEC.
   LOOP.
        DISPLAY 'Enter employee number: ' WITH NO ADVANCING.
        ACCEPT EMP-NUMBER.
        IF EMP-NUMBER = 0
            GO TO NEXT.
        EXEC SQL EXECUTE SQLSTMT USING :EMP-NUMBER END-EXEC.
   NEXT.
```

With Method 2, you must know the datatypes of input host variables at precompile
time. In the last example, *EMP-NUMBER* was declared as type PIC S9(4) COMP. It
could also have been declared as type PIC X(4) or PIC S9(4) COMP-1, because
Oracle8 supports all these datatype conversions to the NUMBER internal datatype.

The USING Clause

When the SQL statement is EXECUTEd, input host variables in the USING clause
replace corresponding place-holders in the PREPAREd dynamic SQL statement.

Every place-holder in the PREPAREd dynamic SQL statement must correspond to
a host variable in the USING clause. So, if the same place-holder appears two or
more times in the PREPAREd statement, each appearance must correspond to a
host variable in the USING clause. If one of the host variables in the USING clause
is an array, all must be arrays.

The names of the place-holders need not match the names of the host variables.
However, the order of the place-holders in the PREPAREd dynamic SQL statement
must match the order of corresponding host variables in the USING clause.

To specify nulls, you can associate indicator variables with host variables in the
USING clause. For more information, see "Using Indicator Variables" on page 5-3.

Sample Program 7: Dynamic SQL Method 2

This program uses dynamic SQL Method 2 to insert two rows into the EMP table,
then delete them.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  DYNSQL2.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
      *      INCLUDE THE SQL COMMUNICATIONS AREA, A STRUCTURE THROUGH
      *      WHICH ORACLE MAKES RUNTIME STATUS INFORMATION (SUCH AS ERROR
      *      CODES, WARNING FLAGS, AND DIAGNOSTIC TEXT) AVAILABLE TO THE
      *      PROGRAM.
             EXEC SQL INCLUDE SQLCA END-EXEC.

      *      INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE THROUGH
      *      WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS INFORMATION
      *      AVAILABLE TO THE PROGRAM.
             EXEC SQL INCLUDE ORACA END-EXEC.

      *      THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
      *      THE ORACA.
             EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

      *      ALL HOST VARIABLES USED IN EMBEDDED SQL MUST APPEAR IN THE
      *      DECLARE SECTION.
             EXEC SQL BEGIN DECLARE SECTION END-EXEC.
                 01  USERNAME  PIC X(10) VALUE "SCOTT".
                 01  PASSWD    PIC X(10) VALUE "TIGER".
                 01  DYNSTMT   PIC X(80) VARYING.
                 01  EMPNO     PIC S9(4) COMPUTATIONAL VALUE 1234.
                 01  DEPTNO1   PIC S9(4) COMPUTATIONAL VALUE 97.
                 01  DEPTNO2   PIC S9(4) COMPUTATIONAL VALUE 99.
             EXEC SQL END DECLARE SECTION END-EXEC.

      *      DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
                 01  EMPNOD    PIC 9(4).
                 01  DEPTNO1D  PIC 9(2).
                 01  DEPTNO2D  PIC 9(2).
                 01  ORASLNRD  PIC 9(9).

       PROCEDURE DIVISION.

       MAIN.
      *      BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
             EXEC SQL
                 WHENEVER SQLERROR GOTO SQLERROR
             END-EXEC.

      *      SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
      *      OCCURS.
             MOVE 1 TO ORASTXTF.

      *      CONNECT TO ORACLE.
```

```
        EXEC SQL
            CONNECT :USERNAME IDENTIFIED BY :PASSWD
        END-EXEC.
        DISPLAY " ".
        DISPLAY "CONNECTED TO ORACLE.".
        DISPLAY " ".

*   ASSIGN A SQL STATEMENT TO THE VARYING STRING DYNSTMT.  BOTH
*   THE ARRAY AND THE LENGTH PARTS MUST BE SET PROPERLY.  NOTE
*   THAT THE STATEMENT CONTAINS TWO HOST VARIABLE PLACEHOLDERS,
*   V1 AND V2, FOR WHICH ACTUAL INPUT HOST VARIABLES MUST BE
*   SUPPLIED AT EXECUTE TIME.
        MOVE "INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:V1, :V2)"
            TO DYNSTMT-ARR.
        MOVE 49 TO DYNSTMT-LEN.

*   DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
*   VARIABLES.
        DISPLAY DYNSTMT-ARR.
        MOVE EMPNO TO EMPNOD.
        MOVE DEPTNO1 TO DEPTNO1D.
        DISPLAY "   V1 = ", EMPNOD, "    V2 = ", DEPTNO1D.

*   THE PREPARE STATEMENT ASSOCIATES A STATEMENT NAME WITH A
*   STRING CONTAINING A SQL STATEMENT.  THE STATEMENT NAME IS A
*   SQL IDENTIFIER, NOT A HOST VARIABLE, AND THEREFORE DOES NOT
*   APPEAR IN THE DECLARE SECTION.

*   A SINGLE STATEMENT NAME MAY BE PREPARED MORE THAN ONCE,
*   OPTIONALLY FROM A DIFFERENT STRING VARIABLE.
        EXEC SQL
            PREPARE S FROM :DYNSTMT
        END-EXEC.

*   THE EXECUTE STATEMENT EXECUTES A PREPARED SQL STATEMENT
*   USING THE SPECIFIED INPUT HOST VARIABLES, WHICH ARE
*   SUBSTITUTED POSITIONALLY FOR PLACEHOLDERS IN THE PREPARED
*   STATEMENT.  FOR EACH OCCURRENCE OF A PLACEHOLDER IN THE
*   STATEMENT THERE MUST BE A VARIABLE IN THE USING CLAUSE.
*   THAT IS, IF A PLACEHOLDER OCCURS MULTIPLE TIMES IN THE
*   STATEMENT, THE CORRESPONDING VARIABLE MUST APPEAR
*   MULTIPLE TIMES IN THE USING CLAUSE.  THE USING CLAUSE MAY
*   BE OMITTED ONLY IF THE STATEMENT CONTAINS NO PLACEHOLDERS.

*   A SINGLE PREPARED STATEMENT MAY BE EXECUTED MORE THAN ONCE,
```

```
*      OPTIONALLY USING DIFFERENT INPUT HOST VARIABLES.
       EXEC SQL
           EXECUTE S USING :EMPNO, :DEPTNO1
       END-EXEC.

*      INCREMENT EMPNO AND DISPLAY NEW INPUT HOST VARIABLES.
       ADD 1 TO EMPNO.
       MOVE EMPNO TO EMPNOD.
       MOVE DEPTNO2 TO DEPTNO2D.
       DISPLAY "    V1 = ", EMPNOD, "    V2 = ", DEPTNO2D.

*      REEXECUTE S TO INSERT THE NEW VALUE OF EMPNO AND A
*      DIFFERENT INPUT HOST VARIABLE, DEPTNO2.  A REPREPARE IS NOT
*      NECESSARY.
       EXEC SQL
           EXECUTE S USING :EMPNO, :DEPTNO2
       END-EXEC.

*      ASSIGN A NEW VALUE TO DYNSTMT.
       MOVE
           "DELETE FROM EMP WHERE DEPTNO = :V1 OR DEPTNO = :V2"
           TO DYNSTMT-ARR.
       MOVE 50 TO DYNSTMT-LEN.

*      DISPLAY THE NEW SQL STATEMENT AND ITS CURRENT INPUT HOST
*      VARIABLES.
       DISPLAY DYNSTMT-ARR.
       DISPLAY "    V1 = ", DEPTNO1D, "     V2 = ", DEPTNO2D.
       REPREPARE S FROM THE NEW DYNSTMT.
       EXEC SQL
           PREPARE S FROM :DYNSTMT
       END-EXEC.

*      EXECUTE THE NEW S TO DELETE THE TWO ROWS PREVIOUSLY INSERTED.
       EXEC SQL
           EXECUTE S USING :DEPTNO1, :DEPTNO2
       END-EXEC.

*      COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
       EXEC SQL
           COMMIT RELEASE
       END-EXEC.
       DISPLAY " ".
       DISPLAY "HAVE A GOOD DAY!".
       DISPLAY " ".
```

```
        STOP RUN.


    SQLERROR.
*       ORACLE ERROR HANDLER.  PRINT DIAGNOSTIC TEXT CONTAINING ERROR
*       MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR.
        DISPLAY SQLERRMC.
        DISPLAY "IN ", ORASTXTC.
        MOVE ORASLNR TO ORASLNRD.
        DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

*       DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
*       SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
        EXEC SQL
            WHENEVER SQLERROR CONTINUE
        END-EXEC.

*       ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
        EXEC SQL
            ROLLBACK RELEASE
        END-EXEC.
        STOP RUN.
```

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 place-holders for input host variables must be known at precompile time. However, the names of database objects such as tables and columns need not be specified until run time (they cannot duplicate the names of host variables). 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:

```
EXEC SQL
    PREPARE STATEMENTNAME FROM { :<HOST-STRING> | <STRING-LITERAL>
END-EXEC.
EXEC SQL DECLARE CURSORNAME CURSOR FOR STATEMENTNAME END-EXEC.
EXEC SQL OPEN CURSORNAME [USING <HOST-VARIABLE-LIST>] END-EXEC.
EXEC SQL FETCH CURSORNAME INTO <HOST-VARIABLE-LIST> END-EXEC.
```

```
                    EXEC SQL CLOSE CURSORNAME END-EXEC.
```

Now let us 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 *SQLSTMT*:

```
              MOVE 'SELECT MGR, JOB FROM EMP WHERE SAL < :SALARY'
                  TO SELECT-STMT.
              EXEC SQL PREPARE SQLSTMT FROM :SELECT-STMT END-EXEC.
```

Commonly, the query WHERE clause is input from a terminal at run time or is generated by the application.

The identifier *SQLSTMT* 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. The cursor declaration is local to its precompilation unit. Continuing our example, DECLARE defines a cursor named *EMP-CURSOR* and associates it with *SQL-STMT*, as follows:

```
              EXEC SQL DECLARE EMPCURSOR CURSOR FOR SQLSTMT END-EXEC.
```

The identifiers *SQLSTMT* and *EMPCURSOR* are *not* host or program variables, but must be unique. If you declare two cursors using the same statement name, Pro*COBOL considers the two cursor names synonymous. For example, if you execute the statements

```
              EXEC SQL PREPARE SQLSTMT FROM :SELECT-STMT END-EXEC.
              EXEC SQL DECLARE EMPCURSOR FOR SQLSTMT END-EXEC.
              EXEC SQL PREPARE SQLSTMT FROM :DELETE-STMT END-EXEC.
              EXEC SQL DECLARE DEPCURSOR FOR SQLSTMT END-EXEC.
```

when you OPEN *EMPCURSOR*, you will process the dynamic SQL statement stored in *DELETE-STMT*, not the one stored in *SELECT-STMT*.

OPEN

OPEN allocates an Oracle8 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 place-holders in the PREPAREd dynamic SQL statement.

In our example, OPEN allocates *EMPCURSOR* and assigns the host variable *SALARY* to the WHERE clause, as follows:

```
EXEC SQL OPEN EMPCURSOR USING :SALARY END-EXEC.
```

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. When no more rows are found, FETCH returns the "no data found" Oracle8 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 EMPCURSOR INTO :MGR-NUMBER,:JOB-TITLE END-EXEC.
```

CLOSE

CLOSE disables the cursor. Once you CLOSE a cursor, you can no longer FETCH from it. In our example, the CLOSE statement disables *EMPCURSOR*, as follows:

```
EXEC SQL CLOSE EMPCURSOR END-EXEC.
```

Sample Program 8: Dynamic SQL Method 3

This program uses dynamic SQL Method 3 to retrieve the names of all employees in a given department from the EMP table.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  DYNSQL3.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

*    INCLUDE THE SQL COMMUNICATIONS AREA, A STRUCTURE
*    THROUGH WHICH ORACLE MAKES RUNTIME STATUS INFORMATION
```

```
*      (SUCH AS ERROR CODES, WARNING FLAGS, AND DIAGNOSTIC
*      TEXT) AVAILABLE TO THE PROGRAM.
       EXEC SQL INCLUDE SQLCA END-EXEC.

*      INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
*      THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
*      INFORMATION AVAILABLE TO THE PROGRAM.
       EXEC SQL INCLUDE ORACA END-EXEC.

*      THE ORACA=YES OPTION MUST BE SPECIFIED TO ENABLE USE OF
*      THE ORACA.
       EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

*      ALL HOST VARIABLES USED IN EMBEDDED SQL MUST APPEAR IN
*      THE DECLARE SECTION.
       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
           01  USERNAME  PIC X(10) VALUE "SCOTT".
           01  PASSWD    PIC X(10) VALUE "TIGER".
           01  DYNSTMT   PIC X(80) VARYING.
           01  ENAME     PIC X(10).
           01  DEPTNO    PIC S99 COMPUTATIONAL VALUE 10.
       EXEC SQL END DECLARE SECTION END-EXEC.

*      DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
           01  DEPTNOD   PIC 9(2).
           01  ENAMED    PIC X(10).
           01  SQLERRD3  PIC 9(2).
           01  ORASLNRD  PIC 9(4).

 PROCEDURE DIVISION.

 MAIN.
*      BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
       EXEC SQL
           WHENEVER SQLERROR GO TO SQLERROR
       END-EXEC.

*      SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN
*      ERROR OCCURS.
       MOVE 1 TO ORASTXTF.

*      CONNECT TO ORACLE.
       EXEC SQL
           CONNECT :USERNAME IDENTIFIED BY :PASSWD
       END-EXEC.
```

```
        DISPLAY " ".
        DISPLAY "CONNECTED TO ORACLE.".
        DISPLAY " ".

*   ASSIGN A SQL QUERY TO THE VARYING STRING DYNSTMT.  BOTH
*   THE ARRAY AND THE LENGTH PARTS MUST BE SET PROPERLY.
*   NOTE THAT THE STATEMENT CONTAINS ONE HOST VARIABLE
*   PLACEHOLDER, V1, FOR WHICH AN ACTUAL INPUT HOST
*   VARIABLE MUST BE SUPPLIED AT OPEN TIME.
        MOVE "SELECT ENAME FROM EMP WHERE DEPTNO = :V1"
            TO DYNSTMT-ARR.
        MOVE 40 TO DYNSTMT-LEN.

*   DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
*   VARIABLE.

        DISPLAY DYNSTMT-ARR.
        MOVE DEPTNO TO DEPTNOD.
        DISPLAY "    V1 = ", DEPTNOD.
        DISPLAY " ".
        DISPLAY "EMPLOYEE".
        DISPLAY "--------".

*   THE PREPARE STATEMENT ASSOCIATES A STATEMENT NAME WITH
*   A STRING CONTAINING A SELECT STATEMENT.  THE STATEMENT
*   NAME, WHICH MUST BE UNIQUE, IS A SQL IDENTIFIER, NOT A
*   HOST VARIABLE, AND SO DOES NOT APPEAR IN THE DECLARE
*   SECTION.
        EXEC SQL
            PREPARE S FROM :DYNSTMT
        END-EXEC.

*   THE DECLARE STATEMENT ASSOCIATES A CURSOR WITHAPREPARED
*   STATEMENT.  THE CURSOR NAME, LIKE THE STATEMENT NAME,
*   DOES NOT APPEAR IN THE DECLARE SECTION.
        EXEC SQL
            DECLARE C CURSOR FOR S
        END-EXEC.

*   THE OPEN STATEMENT EVALUATES THE ACTIVE SET OF THE
*   PREPARED QUERY USING THE SPECIFIED INPUT HOST
*   VARIABLES, WHICH ARE SUBSTITUTED POSITIONALLY FOR
*   PLACEHOLDERS IN THE PREPARED QUERY.  FOR EACH
*   OCCURRENCE OF A PLACEHOLDER IN THE STATEMENT THERE MUST
*   BE A VARIABLE IN THE USING CLAUSE.
```

```
*      THAT IS, IF A PLACEHOLDER OCCURS MULTIPLE TIMES IN THE
*      STATEMENT, THE CORRESPONDING VARIABLE MUST APPEAR
*      MULTIPLE TIMES IN THE USING CLAUSE.  THE USING CLAUSE
*      MAY BE OMITTED ONLY IF THE STATEMENT CONTAINS NO
*      PLACEHOLDERS. OPEN PLACES THE CURSOR AT THE FIRST ROW
*      OF THE ACTIVE SET IN PREPARATION FOR A FETCH.

*      A SINGLE DECLARED CURSOR MAY BE OPENED MORE THAN ONCE,
*      OPTIONALLY USING DIFFERENT INPUT HOST VARIABLES.
       EXEC SQL
           OPEN C USING :DEPTNO
       END-EXEC.

*      BRANCH TO PARAGRAPH NOTFOUND WHEN ALL ROWS HAVE BEEN
*      RETRIEVED.
       EXEC SQL
           WHENEVER NOT FOUND GO TO NOTFOUND
       END-EXEC.

 GETROWS.
*      THE FETCH STATEMENT PLACES THE SELECT LIST OF THE
*      CURRENT ROW INTO THE VARIABLES SPECIFIED BY THE INTO
*      CLAUSE, THEN ADVANCES THE CURSOR TO THE NEXT ROW.  IF
*      THERE ARE MORE SELECT-LIST FIELDS THAN OUTPUT HOST
*      VARIABLES, THE EXTRA FIELDS ARE NOT RETURNED.
*      SPECIFYING MORE OUTPUT HOST VARIABLES THAN SELECT-LIST
*      FIELDS RESULTS IN AN ORACLE ERROR.
       EXEC SQL
           FETCH C INTO :ENAME
       END-EXEC.
       MOVE ENAME TO ENAMED.
       DISPLAY ENAMED.

*      LOOP UNTIL NOT FOUND CONDITION IS DETECTED.
       GO TO GETROWS.

 NOTFOUND.
       MOVE SQLERRD(3) TO SQLERRD3.
       DISPLAY " ".
       DISPLAY "QUERY RETURNED ", SQLERRD3, " ROW(S).".

*      THE CLOSE STATEMENT RELEASES RESOURCES ASSOCIATED WITH
*      THE CURSOR.
       EXEC SQL
           CLOSE C
```

```
        END-EXEC.

*       COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
        EXEC SQL
            COMMIT RELEASE
        END-EXEC.
        DISPLAY " ".
        DISPLAY "HAVE A GOOD DAY!".
        DISPLAY " ".
        STOP RUN.

 SQLERROR.
*       ORACLE ERROR HANDLER.  PRINT DIAGNOSTIC TEXT CONTAINING
*       ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF
*       ERROR.
        DISPLAY SQLERRMC.
        DISPLAY "IN ", ORASTXTC.
        MOVE ORASLNR TO ORASLNRD.
        DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

*       DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
*       SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
        EXEC SQL
            WHENEVER SQLERROR CONTINUE
        END-EXEC.

*       RELEASE RESOURCES ASSOCIATED WITH THE CURSOR.
        EXEC SQL
            CLOSE C
        END-EXEC.

*       ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM
*       ORACLE.
        EXEC SQL
            ROLLBACK RELEASE
        END-EXEC.
        STOP RUN.
*       exit program with an error;
```

Using Method 4

This section only gives an overview. For details, see Chapter 12, "Using Dynamic SQL: Advanced Concepts".

There is a kind of dynamic SQL statement that your program cannot process using Method 3. When the number of select-list items or place-holders 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 Oracle8 to hold a complete description of the variables in a dynamic SQL statement.

Recall that for a multi-row 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 precompile time by the INTO clause. For example, you know the following query returns two column values:

```
EXEC SQL
    SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

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 place-holders for input host variables, the host-variable list cannot be established at precompile time by the USING clause.

To process the dynamic SQL statement, your program must issue the DESCRIBE BIND VARIABLES command and declare another kind of SQLDA called a *bind descriptor* to hold descriptions of the place-holders for the 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 place-holder 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 place-holder 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.

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 place-holders that can be DESCRIBEd
- actual number of place-holders 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 place-holder names

- sizes of buffers to store place-holder names

- current lengths of place-holder names

- addresses of buffers to store indicator-variable names

- sizes of buffers to store indicator-variable names

- current lengths of indicator-variable names

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> }
END-EXE
EXEC SQL
    DECLARE <CURSOR-NAME> CURSOR FOR <STATEMENT-NAME>
END-EXEC.
EXEC SQL
    DESCRIBE BIND VARIABLES FOR <STATEMENT-NAME>
    INTO <BIND-DESCRIPTOR-NAME>
END-EXEC.
EXEC SQL
    OPEN <CURSOR-NAME>
    [USING DESCRIPTOR <BIND-DESCRIPTOR-NAME>]
END-EXEC.
EXEC SQL
    DESCRIBE [SELECT LIST FOR] <STATEMENT-NAME>
    INTO <SELECT-DESCRIPTOR-NAME>
END-EXEC.
EXEC SQL
    FETCH <CURSOR-NAME>
    USING DESCRIPTOR <SELECT-DESCRIPTOR-NAME>
END-EXEC.
EXEC SQL CLOSE <CURSOR-NAME> END-EXEC.
```

Select and bind descriptors need not work in tandem. If the number of columns in a query select list is known, but the number of place-holders for input host vari-

ables is unknown, you can use the Method 4 OPEN statement with the following Method 3 FETCH statement:

```
EXEC SQL FETCH <EMPCURSOR> INTO :<HOST-VARIABLE-LIST> END-EXEC.
```

Conversely, if the number of place-holders for input host variables is known, but the number of columns in the select list is unknown, you can use the following Method 3 OPEN statement with the Method 4 FETCH statement:

```
EXEC SQL OPEN <CURSORNAME> [USING <HOST-VARIABLE-LIST>] END-EXEC.
```

Note that EXECUTE can be used for non-queries with Method 4.

Using the DECLARE STATEMENT Statement

With Methods 2, 3, and 4, you might need to use the statement

```
EXEC SQL [AT <dbname>] DECLARE <statementname> STATEMENT END-EXEC.
```

where *dbname* and *statementname* are identifiers used by Pro*COBOL, *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 <remotedb> DECLARE <sqlstmt> STATEMENT END-EXEC.
EXEC SQL PREPARE <sqltmt> FROM :<SQL-STRING> END-EXEC.
EXEC SQL EXECUTE <sqlstmt> END-EXEC.
```

In the example, *remotedb* tells Oracle8 where to 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 <sqlstmt> STATEMENT END-EXEC.
EXEC SQL DECLARE <empcursor> CURSOR FOR <sqlstmt> END-EXEC.
EXEC SQL PREPARE <sqlstmt> FROM :<SQL-STRING> END-EXEC.
```

The usual sequence of statements is

```
EXEC SQL PREPARE <sqlstmt> FROM :<SQL-STRING> END-EXEC.
EXEC SQL DECLARE <empcursor> CURSOR FOR <sqlstmt> END-EXEC.
```

Using Host Tables

Usage of host tables in static and dynamic SQL is similar. For example, to use input host tables with dynamic SQL Method 2, use the syntax

```
EXEC SQL EXECUTE <statementname> USING :HOST-TABLE-LIST END-EXEC.
```

where *HOST-TABLE-LIST* contains one or more host tables. With Method 3, use the following syntax:

```
OPEN <cursorname> USING :<HOST-TABLE-LIST> END-EXEC.
```

To use output host tables with Method 3, use the following syntax:

```
FETCH <cursorname> INTO :<HOST-TABLE-LIST> END-EXEC.
```

With Method 4, you must use the optional FOR clause to tell Oracle8 the size of your input or output host table. To learn how this is done, see your host-language supplement.

Using PL/SQL

Pro*COBOL treats 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 Pro*COBOL handles SQL and PL/SQL:

■ Pro*COBOL 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 place-holders in the PREPAREd string. Though Pro*COBOL 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 place-holder in the PREPAREd PL/SQL string must correspond to a host variable in the USING clause. So, if the same place-holder 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, 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 set up 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 Pro*COBOL 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 host-language supplement.

Attention:

In dynamic SQL Method 4, a host array cannot be bound to a PL/SQL procedure with a parameter of type "table."

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 (/* ... */).

12

Using Dynamic SQL: Advanced Concepts

This chapter shows you how to implement dynamic SQL Method 4, which lets your program accept or build dynamic SQL statements that contain a varying number of host variables. Subjects discussed include the following:

- Meeting the Special Requirements of Method 4

- Understanding the SQL Descriptor Area (SQLDA)

- The SQLDA Variables

- Some Preliminaries

- The Basic Steps

- A Closer Look at Each Step

- Using Host Tables with Method 4

- Sample Program 10: Dynamic SQL Method 4

  Note: For a discussion of dynamic SQL Methods 1, 2, and 3, and an overview of Method 4, see Chapter 11, "Using Dynamic SQL"

Meeting the Special Requirements of Method 4

Before looking into the requirements of Method 4, you should feel comfortable with the terms *select-list item* and *place-holder*. Select-list items are the columns or expressions following the keyword SELECT in a query. For example, the following dynamic query contains three select-list items:

```
SELECT ENAME, JOB, SAL + COMM FROM EMP WHERE DEPTNO = 20
```

Place-holders are dummy bind (input) variables that hold places in a SQL statement for actual bind variables. You do not declare place-holders and can name them anything you like. place-holders for bind variables are most often used in the SET, VALUES, and WHERE clauses. For example, the following dynamic SQL statements each contain two place-holders.

```
INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:E, :D)
DELETE FROM DEPT WHERE DEPTNO = :DNUM AND LOC = :DLOC
```

place-holders cannot reference table or column names.

What Makes Method 4 Special?

Unlike Methods 1, 2, and 3, dynamic SQL Method 4 lets your program

- accept or build dynamic SQL statements that contain an unknown number of select-list items or place-holders

- take explicit control over datatype conversion between Oracle8 and COBOL types

To add this flexibility to your program, you must give the Oracle8 runtime library additional information.

What Information Does Oracle8 Need?

The Pro*COBOL Precompiler generates calls to Oracle8 for all executable dynamic SQL statements. If a dynamic SQL statement contains no select-list items or place-holders, Oracle8 needs no additional information to execute the statement. The following DELETE statement falls into this category:

```
*    Dynamic SQL statement...
     MOVE 'DELETE FROM EMP WHERE DEPTNO = 30' TO STMT.
```

However, most dynamic SQL statements contain select-list items or place-holders for bind variables, as shown in the following UPDATE statement:

```
*    Dynamic SQL statement with place-holders...
     MOVE 'UPDATE EMP SET COMM = :C WHERE EMPNO = :E' TO STMT.
```

To execute a dynamic SQL statement that contains select-list items and/or place-holders for bind variables, Oracle8 needs information about the program variables that will hold output or input values. Specifically, Oracle8 needs the following information:

- the number of select-list items and the number of bind variables

- the length of each select-list item and bind variable

- the datatype of each select-list item and bind variable

- the memory address of each output variable that will store the value of a select-list item, and the address of each bind variable

For example, to write the value of a select-list item, Oracle8 needs the address of the corresponding output variable.

Where Is the Information Stored?

All the information Oracle8 needs about select-list items or place-holders for bind variables, except their values, is stored in a program data structure called the SQL Descriptor Area (SQLDA).

Descriptions of select-list items are stored in a *select SQLDA*, and descriptions of place-holders for bind variables are stored in a *bind SQLDA*.

The values of select-list items are stored in output buffers; the values of bind variables are stored in input buffers. You use the library routine SQLADR to store the addresses of these data buffers in a select or bind SQLDA, so that Oracle8 knows where to write output values and read input values.

How do values get stored in these data variables? Output values are FETCHed using a cursor, and input values are filled in by your program, typically from information entered interactively by the user.

How Is the Information Obtained?

You use the DESCRIBE statement to help get the information Oracle8 needs. The DESCRIBE SELECT LIST statement examines each select-list item to determine its name, datatype, constraints, length, scale, and precision, then stores this information in the select SQLDA for your use. For example, you might use select-list names as column headings in a printout. DESCRIBE also stores the total number of select-list items in the SQLDA.

The DESCRIBE BIND VARIABLES statement examines each place-holder to determine its name and length, then stores this information in an input buffer and bind SQLDA for your use. For example, you might use place-holder names to prompt the user for the values of bind variables.

Understanding the SQL Descriptor Area (SQLDA)

This section describes the SQLDA data structure in detail. You learn how to declare it, what variables it contains, how to initialize them, and how to use them in your program.

Purpose of the SQLDA

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or place-holders for bind variables. To process this kind of dynamic SQL statement, your program must explicitly declare SQLDAs, also called *descriptors.* Each descriptor corresponds to a group item in your program.

A *select descriptor* stores descriptions of select-list items and the addresses of output buffers that hold the names and values of select-list items.

> **Note:** The name of a select-list item can be a column name, a column alias, or the text of an expression such as SAL + COMM.

A *bind descriptor* stores descriptions of bind variables and indicator variables, and the addresses of input buffers where the names and values of bind variables and indicator variables are stored.

Remember, some descriptor variables contain addresses, not values. So, you must declare data buffers to hold the values. You decide the sizes of the required input and output buffers. Because COBOL does not support pointers, you must use the library subroutine SQLADR to get the addresses of input and output buffers. You learn how to call SQLADR in the section "Using SQLADR" on page 12-14.

Multiple SQLDAs

If your program has more than one active dynamic SQL statement, each statement must have its own SQLDA(s). You can declare any number of SQLDAs with different names. For example, you might declare three select SQLDAs named SELDSC1, SELDSC2, and SELDSC3, so that you can FETCH from three concurrently open cursors. However, non-concurrent cursors can reuse SQLDAs.

Declaring a SQLDA

To declare select and bind SQLDAs, you can hardcode them into your program using the sample select and bind SQLDAs shown in Figure 12–1. You can modify the table dimensions to suit your needs.

Note: For byte-swapped platforms, use COMP5 instead of COMP when declaring a SQLDA.

Figure 12–1   Sample Pro*COBOL SQLDA Descriptors and Data Buffers

```
01 SELDSC.
   05  SQLDNUM                    PIC S9(9) COMP.
   05  SQLDFND                    PIC S9(9) COMP.
   05  SELDVAR                    OCCURS 20 TIMES.
       10  SELDV                  PIC S9(9) COMP.
       10  SELDFMT                PIC S9(9) COMP.
       10  SELDVLN                PIC S9(9) COMP.
       10  SELDFMTL               PIC S9(4) COMP.
       10  SELDVTYP               PIC S9(4) COMP.
       10  SELDI                  PIC S9(9) COMP.
       10  SELDH-VNAME            PIC S9(9) COMP.
       10  SELDH-MAX-VNAMEL       PIC S9(4) COMP.
       10  SELDH-CUR-VNAMEL       PIC S9(4) COMP.
       10  SELDI-VNAME            PIC S9(9) COMP.
       10  SELDI-MAX-VNAMEL       PIC S9(4) COMP.
       10  SELDI-CUR-VNAMEL       PIC S9(4) COMP.
       10  SELDFCLP               PIC S9(9) COMP.
       10  SELDFCRCP              PIC S9(9) COMP.
01  XSELDI.
    05 SEL-DI        OCCURS 20 TIMES PIC S9(4) COMP.
01  XSELDIVNAME.
    05 SEL-DI-VNAME   OCCURS 20 TIMES PIC X(80).
01  XSELDV.
    05 SEL-DV         OCCURS 20 TIMES PIC X(80).
01  XSELDHVNAME
    05 SEL-DH-VNAME   OCCURS 20 TIMES PIC X(80).
01  XSEL-DFMT        PIC X(6).

01 BNDDSC.
   05  SQLDNUM                    PIC S9(9) COMP.
   05  SQLDFND                    PIC S9(9) COMP.
   05  BNDDVAR                    OCCURS 20 TIMES.
       10  BNDDV                  PIC S9(9) COMP.
       10  BNDDFMT                PIC S9(9) COMP.
       10  BNDDVLN                PIC S9(9) COMP.
       10  BNDDFMTL               PIC S9(4) COMP.
       10  BNDDVTYP               PIC S9(4) COMP.
       10  BNDDI                  PIC S9(9) COMP.
       10  BNDDH-VNAME            PIC S9(9) COMP.
       10  BNDDH-MAX-VNAMEL       PIC S9(4) COMP.
       10  BNDDH-CUR-VNAMEL       PIC S9(4) COMP.
       10  BNDDI-VNAME            PIC S9(9) COMP.
       10  BNDDI-MAX-VNAMEL       PIC S9(4) COMP.
       10  BNDDI-CUR-VNAMEL       PIC S9(4) COMP.
       10  BNDDFCLP               PIC S9(9) COMP.
       10  BNDDFCRCP              PIC S9(9) COMP.
01  XBNDDI.
    05 BND-DI        OCCURS 20 TIMES PIC S9(4) COMP.
01  XBNDDINAME.
    05 BND-DI-VNAME   OCCURS 20 TIMES PIC X(80).
01  XBNDDV.
    05 BND-DV         OCCURS 20 TIMES PIC X(80).
01  XBNDDHVNAME
    05 BND-DH-VNAME   OCCURS 20 TIMES PIC X(80).
01  XBND-DFMT        PIC X(6).
```

You might want to store the SQLDAs in files (named SELDSC and BNDDSC, for example), then copy the files into your program with the INCLUDE statement as follows:

```
EXEC SQL INCLUDE SELDSC END-EXEC.
EXEC SQL INCLUDE BNDDSC END-EXEC.
```

Figure 12–2 shows whether variables are set by SQLADR calls, DESCRIBE commands, FETCH commands, or program assignments.

Figure 12–2   How Variables Are Set

Dynamic SQL Statement

'SELECT ENAME FROM EMP WHERE EMPNO=:NUM'

	select–list item (SLI)	placeholder (P) for bind variable (BV)
	Select SQLDA	**Bind SQLDA**
Set by:		
SQLADR	Address of SLI name buffer	Address of P name buffer
SQLADR	Address of SLI value buffer	Address of BV value buffer
DESCRIBE	Length of SLI name	Length of P name
DESCRIBE	Datatype of select-list item	
Program	Length of SLI name buffer	Length of P name buffer
Program	Length of BV value buffer	Length of BV value buffer
Program	Datatype of SLI value buffer	Datatype of BV value buffer
	Output Buffers	**Input Buffers**
DESCRIBE	Name of select-list item	Name of placeholder
FETCH	Value of select-list item	Value of bind variable

The SQLDA Variables

This section explains the purpose and use of each variable in the SQLDA.

SQLDNUM

This variable specifies the maximum number of select-list items or place-holders that can be DESCRIBEd. Thus, SQLDNUM determines the number of elements in the descriptor tables.

Before issuing a DESCRIBE command, you must set this variable to the dimension of the descriptor tables. After the DESCRIBE, you must reset it to the actual number of variables DESCRIBEd, which is stored in SQLDFND.

SQLDFND

This is the actual number of select-list items or place-holders found by the DESCRIBE command.

SQLDFND is set by DESCRIBE. If SQLDFND is negative, the DESCRIBE command found too many select-list items or place-holders for the size of the descriptor. For example, if you set SQLDNUM to 10 but DESCRIBE finds 11 select-list items or place-holders, SQLDFND is set to -11. If this happens, you cannot process the SQL statement without reallocating the descriptor.

After the DESCRIBE, you must set SQLDNUM equal to SQLDFND.

SELDV|BNDDV

This is a table containing the addresses of data buffers that store select-list or bind-variable values.

You must set the elements of SELDV or BNDDV using SQLADR.

Select Descriptors

The following statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

directs Oracle8 to store FETCHed select-list values in the data buffers addressed by SELDV(1) through SELDV(SQLDNUM). Thus, Oracle8 stores the Jth select-list value in SEL-DV(J).

Bind Descriptors

You must set this table before issuing the OPEN command. The following statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

directs Oracle8 to execute the dynamic SQL statement using the bind-variable values addressed by BNDDV(1) through BNDDV(SQLDNUM). (Typically, the values are entered by the user.) Oracle8 finds the Jth bind-variable value in BND-DV(J).

SELDFMT|BNDDFMT

This is a table containing the addresses of data buffers that store select-list or bind-variable conversion format strings. Currently, you can use it only for COBOL packed decimals. The format for the conversion string is PP.+SS or PP.-SS where PP is the precision and SS is the scale. For definitions of precision and scale, see the section "Extracting Precision and Scale" on page 12-20.

The use of format strings is optional. If you want a conversion format for the Jth select-list item or bind variable, set SELDFMT(J) or BNDDFMT(J) using SQLADR, then store the packed-decimal format ("07.+02" for example) in SEL-DFMT or BND-DFMT. Otherwise, set SELDFMT(J) or BNDDFMT(J) to zero.

SELDVLN|BNDDVLN

This is a table containing the lengths of select-list or bind-variable values stored in the data buffers.

Select Descriptors

DESCRIBE SELECT LIST sets the table of lengths to the maximum expected for each select-list item. However, you might want to reset some lengths before issuing a FETCH command. FETCH returns at most *n* characters, where *n* is the value of SELD-VLN(J) before the FETCH command.

The format of the length differs among Oracle8 datatypes. For CHAR select-list items, DESCRIBE SELECT LIST sets SELDVLN(J) to the maximum length in bytes of the select-list item. For NUMBER select-list items, scale and precision are returned respectively in the low and next-higher bytes of the variable. You can use the library routine SQLPRC to extract precision and scale values from SELDVLN. See the section "Extracting Precision and Scale" on page 12-20.

You must reset SELDVLN(J) to the required length of the data buffer before the FETCH. For example, when coercing a NUMBER to a COBOL character string, set SELDVLN(J) to the precision of the number plus two for the sign and decimal point. When coercing a NUMBER to a COBOL floating point number, set SELD-VLN(J) to the length of the appropriate floating point type on your system. For

more information about the lengths of coerced datatypes, see the section "Converting Data" on page 12-15.

Bind Descriptors

You must set the table of lengths before issuing the OPEN command. For example, you can use the following statements to set the lengths of bind-variable character strings entered by the user:

```
PROCEDURE DIVISION.
    ...
    PERFORM GET-INPUT-VAR
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
    ...
GET-INPUT-VAR.
    DISPLAY "Enter value of ", BND-DH-VNAME(J).
    ACCEPT INPUT-STRING.
    UNSTRING INPUT-STRING DELIMITED BY "   "
        INTO BND-DV(J) COUNT IN BNDDVLN(J).
```

Because Oracle8 accesses a data buffer indirectly, using the address in SELDV(J) or BNDDV(J), it does not know the length of the value in that buffer. If you want to change the length Oracle8 uses for the Jth select-list or bind-variable value, reset SELDVLN(J) or BNDDVLN(J) to the length you need. Each input or output buffer can have a different length.

SELDFMTL | BNDDFMTL

This is a table containing the lengths of select-list or bind-variable conversion format strings. Currently, you can use it only for COBOL packed decimal.

The use of format strings is optional. If you want a conversion format for the Jth select-list item or bind variable, set SELDFMTL(J) before the FETCH or BNDDFMTL(J) before the OPEN to the length of the packed-decimal format stored in SEL-DFMT or BND-DFMT. Otherwise, set SELDFMTL(J) or BNDDFMTL(J) to zero.

If the value of SELDFMTL(J) or BNDDFMTL(J) is zero, SELDFMT(J) or BNDDFMT(J) is not used.

SELDVTYP|BNDDVTYP

This is a table containing the datatype codes of select-list or bind-variable values. These codes determine how Oracle8 data is converted when stored in the data buffers addressed by elements of SELDV. This topic is covered in "Converting Data" on page 12-15.

Select Descriptors

DESCRIBE SELECT LIST sets the table of datatype codes to the *internal* datatype (for example, VARCHAR2, CHAR, NUMBER, or DATE) of the items in the select list.

Before FETCHing, you might want to reset some datatypes because the internal format of Oracle8 datatypes can be difficult to handle. For display purposes, it is usually a good idea to coerce the datatype of select-list values to VARCHAR2. For calculations, you might want to coerce numbers from Oracle8 to COBOL format. See "Coercing Datatypes" on page 12-18.

The high bit of SELDVTYP(J) is set to indicate the null/not null status of the Jth select-list column. You must always clear this bit before issuing an OPEN or FETCH command. You use the library routine SQLNUL to retrieve the datatype code and clear the null/not null bit. For more information, see "Handling Null/Not Null Datatypes" on page 12-21.

You should change the Oracle8 NUMBER internal datatype to an external datatype compatible with that of the COBOL data buffer addressed by SELDV(J).

Bind Descriptors

DESCRIBE BIND VARIABLES sets the table of datatype codes to zeros. You must reset the table of datatypes before issuing the OPEN command. The code represents the external (COBOL) datatype of the buffer addressed by BNDDV(J). Often, bind-variable values are stored in character strings, so the datatype table elements are set to 1 (the VARCHAR2 datatype code).

To change the datatype of the Jth select-list or bind-variable value, reset SELDVTYP(J) or BNDDVTYP(J) to the datatype you want.

SELDI|BNDDI

This is a table containing the addresses of data buffers that store indicator-variable values. You must set the elements of SELDI or BNDDI using SQLADR.

Select Descriptors

You must set this table before issuing the FETCH command. When Oracle8 executes the statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

if the Jth returned select-list value is null, the buffer addressed by SELDI(J) is set to -1. Otherwise, it is set to zero (the value is not null) or a positive integer (the value was truncated).

Bind Descriptors

You must initialize this table and set the associated indicator variables before issuing the OPEN command. When Oracle8 executes the statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

the buffer addressed by BNDDI(J) determines whether the Jth bind variable is null. If the value of an indicator variable is -1, its associated bind variable is null.

SELDH-VNAME|BNDDH-VNAME

This is a table containing the addresses of data buffers that store select-list or place-holder names as they appear in dynamic SQL statements. You must set the elements of SELDH-VNAME or BNDDH-VNAME using SQLADR before issuing the DESCRIBE command.

DESCRIBE directs Oracle8 to store the name of the Jth select-list item or place-holder in the data buffer addressed by SELDH-VNAME(J) or BNDDH-VNAME(J). Thus, Oracle8 stores the Jth select-list or place-holder name in SEL-DH-VNAME(J) or BND-DH-VNAME(J).

SELDH-MAX-VNAMEL|BNDDH-MAX-VNAMEL

This is a table containing the maximum lengths of the data buffers that store select-list or place-holder names. The buffers are addressed by the elements of SELDH-VNAME or BNDDH-VNAME.

You must set the elements of SELDH-MAX-VNAMEL or BNDDH-MAX-VNAMEL before issuing the DESCRIBE command. Each select-list or place-holder name buffer can have a different length.

SELDH-CUR-VNAMEL|BNDDH-CUR-VNAMEL

This is a table containing the actual lengths of the names of the select-list or place-holder. DESCRIBE sets the table of actual lengths to the number of characters in each select-list or place-holder name.

SELDI-VNAME|BNDDI-VNAME

This is a table containing the addresses of data buffers that store indicator-variable names.

You can associate indicator-variable values with select-list items and bind variables. However, you can associate indicator-variable names only with bind variables. So,

you can use this table only with bind descriptors. You must set the elements of BNDDI-VNAME using SQLADR before issuing the DESCRIBE command.

DESCRIBE BIND VARIABLES directs Oracle8 to store any indicator-variable names in the data buffers addressed by BNDDI-VNAME(1) through BNDDI-VNAME(SQLDNUM). Thus, Oracle8 stores the Jth indicator-variable name in BNDDI-VNAME(J).

SELDI-MAX-VNAMEL|BNDDI-MAX-VNAMEL

This is a table containing the maximum lengths of the data buffers that store indicator-variable names. The buffers are addressed by the elements of SELDI-VNAME or BNDDI-VNAME.

You can associate indicator-variable names only with bind variables. So, you can use this table only with bind descriptors.

You must set the elements BNDDI-MAX-VNAMEL(1) through BNDDI-MAX-VNAMEL(SQLDNUM) before issuing the DESCRIBE command. Each indicator-variable name buffer can have a different length.

SELDI-CUR-VNAMEL|BNDDI-CUR-VNAMEL

This is a table containing the actual lengths of the names of the indicator variables. You can associate indicator-variable names only with bind variables. So, you can use this table only with bind descriptors.

DESCRIBE BIND VARIABLES sets the table of actual lengths to the number of characters in each indicator-variable name.

SELDFCLP|BNDDFCLP

This is a table reserved for future use. It must be present because Oracle8 expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCLP or BNDDFCLP to zero.

SELDFCRCP|BNDDFCRCP

This is a table reserved for future use. It must be present because Oracle8 expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCRCP or BNDDFCRCP to zero.

Some Preliminaries

You need a working knowledge of the following subjects to implement dynamic SQL Method 4:

- using the library routine SQLADR
- converting data
- coercing datatypes
- handling null/not null datatypes

Using SQLADR

You must call the library subroutine SQLADR to get the addresses of data buffers that store input and output values. You store the addresses in a bind or select SQLDA so that Oracle8 knows where to read bind-variable values or write select-list values.

Call SQLADR using the syntax

```
CALL "SQLADR" USING BUFFER, ADDRESS.
```

where:

BUFFER
Is a data buffer that stores the value or name of a select-list item, bind variable, or indicator variable.

ADDRESS
Is an integer variable that returns the address of the data buffer.

A call to SQLADR stores the address of BUFFER in ADDRESS. In the next example, you use SQLADR to initialize the select descriptor tables SELDV, SELDH-VNAME, and SELDI. Their elements address data buffers for select-list values, select-list names, and indicator values.

```
PROCEDURE DIVISION.
    ...
    PERFORM INIT-SELDSC
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
    ...
INIT-SELDSC.
    CALL "SQLADR" USING SEL-DV(J), SELDV(J).
    CALL "SQLADR" USING SEL-DH-VNAME(J), SELDH-VNAME(J).
    CALL "SQLADR" USING SEL-DI(J), SELDI(J).
```

Converting Data

This section provides more detail about the datatype descriptor table. In host programs that use neither datatype equivalencing nor dynamic SQL Method 4, the conversion between Oracle8 internal and external datatypes is determined at precompile time. By default, Pro*COBOL assigns a specific external datatype to each host variable. For example, Pro*COBOL assigns the INTEGER external datatype to host variables of type PIC S9(n) COMP.

However, Method 4 lets you control data conversion and formatting. You specify conversions by setting datatype codes in the datatype descriptor table.

Internal Datatypes

Internal datatypes specify the formats used by Oracle8 to store column values in database tables and the formats to represent pseudocolumn values.

When you issue a DESCRIBE SELECT LIST command, Oracle8 returns the internal datatype code for each select-list item to the SELDVTYP (datatype) descriptor table. For example, the datatype code for the Jth select-list item is returned to SELDVTYP(J).

Table 12–1 shows the Oracle8 internal datatypes and their codes.

Table 12–1   Internal Datatypes and Related Codes

Oracle8 Internal Datatype	Code
VARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHAR	96
MLSLABEL	105

External Datatypes

External datatypes specify the formats used to store values in input and output host variables.

The DESCRIBE BIND VARIABLES command sets the BNDDVTYP table of datatype codes to zeros. So, you must reset the codes *before* issuing the OPEN command. The codes tell Oracle8 which external datatypes to expect for the various bind variables. For the Jth bind variable, reset BNDDVTYP(J) to the external datatype you want.

The following table shows the Oracle8 external datatypes and their codes, as well as the corresponding COBOL datatypes:

Table 12–2   Oracle8 External and Related COBOL Datatypes

Name	Code	COBOL Datatype
VARCHAR2	1	PIC X(n) when MODE != ANSI
NUMBER	2	PIC X(n)
INTEGER	3	PIC S9(n) COMP PIC S9(n) COMP5 (COMP5 for byte-swapped platforms)
FLOAT	4	COMP-1 COMP-2
STRING (1)	5	PIC X(n)
VARNUM	6	PIC X(n)
DECIMAL	7	PIC S9(n)V9(n) COMP-3
LONG	8	PIC X(n)
VARCHAR (2)	9	PIC X(n) VARYING PIC N(n) VARYING
ROWID	11	PIC X(n)
DATE	12	PIC X(n)
VARRAW (2)	15	PIC X(n)
RAW	23	PIC X(n)
LONG RAW	24	PIC X(n)

Table 12–2   Oracle8 External and Related COBOL Datatypes

Name	Code	COBOL Datatype
UNSIGNED	68	(not supported)
DISPLAY	91	PIC S9...9V9...9 DISPLAY SIGN LEADING SEPARATE PIC S9(*n*)V9(*n*) DISPLAY SIGN LEADING SEPARATE
LONG VARCHAR (2)	94	PIC X(*n*)
LONG VARRAW (2)	95	PIC X(*n*)
CHARF	96	PIC X(*n*) when MODE = ANSI PIC N(*n*) when MODE = ANSI
CHARZ (1)	97	PIC X(*n*)
CURSOR	102	SQL-CURSOR
MLSLABEL	106	PIC X(*n*)

Notes:

1.   For use in an EXEC SQL VAR statement only.

2.   Include the *n*-byte length field.

For more information about the Oracle8 datatypes and their formats, see "The Oracle8 Datatypes" on page 4-2.

PL/SQL Datatypes

PL/SQL provides a variety of predefined scalar and composite datatypes. A *scalar* type has no internal components. A *composite* type has internal components that can be manipulated individually. Table 12–3 shows the predefined PL/SQL scalar datatypes and their Oracle8 internal datatype equivalences.

Table 12–3   PL/SQL Datatype Equivalences with Oracle8 Internal Datatypes

PL/SQL Datatype	Oracle8 Internal Datatype
VARCHAR VARCHAR2	VARCHAR2
BINARY_INTEGER DEC DECIMAL DOUBLE PRECISION FLOAT INT INTEGER NATURAL NUMBER NUMERIC POSITIVE REAL SMALLINT	NUMBER
LONG	LONG
ROWID	ROWID
DATE	DATE
RAW	RAW
LONG RAW	LONG RAW
CHAR CHARACTER STRING	CHAR
MLSLABEL	MLSLABEL

Coercing Datatypes

For a select descriptor, DESCRIBE SELECT LIST can return any of the Oracle8 internal datatypes. Often, as in the case of character data, the internal datatype corre-

sponds exactly to the external datatype you want to use. However, a few internal datatypes map to external datatypes that can be difficult to handle. So, you might want to reset some elements in the SELDVTYP descriptor table.

For example, you might want to reset NUMBER values to FLOAT values, which correspond to PIC S9(*n*)V9(*n*) COMP-1 values in COBOL. Oracle8 does any necessary conversion between internal and external datatypes at FETCH time. So, be sure to reset the datatypes *after* the DESCRIBE SELECT LIST but *before* the FETCH.

For a bind descriptor, DESCRIBE BIND VARIABLES does *not* return the datatypes of bind variables, only their number and names. Therefore, you must explicitly set the BND-DVTYP table of datatype codes to tell Oracle8 the external datatype of each bind variable. Oracle8 does any necessary conversion between external and internal datatypes at OPEN time.

When you reset datatype codes in the SELDVTYP or BNDDVTYP descriptor table, you are "coercing datatypes." For example, to coerce the Jth select-list value to VARCHAR2, use the following statement:

```
*    Coerce select-list value to VARCHAR2.
     MOVE 1 TO SELDVTYP(J).
```

When coercing a NUMBER select-list value to VARCHAR2 for display purposes, you must also extract the precision and scale bytes of the value and use them to compute a maximum display length. Then, before the FETCH, you must reset the appropriate element of the SELDVLN (length) descriptor table to tell Oracle8 the buffer length to use. To specify the length of the Jth select-list value, set SELD-VLN(J) to the length you need.

For example, if DESCRIBE SELECT LIST finds that the Jth select-list item is of type NUMBER, and you want to store the returned value in a COBOL variable declared as PIC S9(*n*)V9(*n*) COMP-1, simply set SELDVTYP(J) to 4 and SELDVLN(J) to the length of COMP-1 numbers on your system.

Exceptions

In some cases, the internal datatypes that DESCRIBE SELECT LIST returns might not suit your purposes. Two examples of this are DATE and NUMBER. When you DESCRIBE a DATE select-list item, Oracle8 returns the datatype code 12 to the SELDVTYP table. Unless you reset the code before the FETCH, the date value is returned in its 7-byte internal format. To get the date in its default character format, you must change the datatype code from 12 to 1 (VARCHAR2), and increase the SELDVLN value from 7 to 9.

Similarly, when you DESCRIBE a NUMBER select-list item, Oracle8 returns the datatype code 2 to the SELDVTYP table. Unless you reset the code before the FETCH, the numeric value is returned in its internal format, which is probably not what you want. So, change the code from 2 to 1 (VARCHAR2), 3 (INTEGER), 4 (FLOAT), or some other appropriate datatype.

Extracting Precision and Scale

The library subroutine SQLPRC extracts precision and scale. Normally, it is used after the DESCRIBE SELECT LIST, and its first parameter is SELDVLN(J). To call SQLPRC, use the following syntax

```
CALL "SQLPRC" USING LENGTH, PRECISION, SCALE.
```

where:

LENGTH	Is an integer variable that stores the length of an Oracle8 NUMBER value. The scale and precision of the value are stored in the low and next-higher bytes, respectively.
PRECISION	Is an integer variable that returns the *precision* of the NUMBER value. Precision is the number of significant digits. It is set to zero if the select-list item refers to a NUMBER of unspecified size. In this case, because the size is unspecified, assume the maximum precision, 38.
SCALE	Is an integer variable that returns the *scale* of the NUMBER value. Scale specifies where rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of -3 means that the number is rounded to the nearest thousand (3.456 becomes 3000).

The following example shows how SQLPRC is used to compute maximum display lengths for NUMBER values that will be coerced to VARCHAR2:

```
WORKING-STORAGE SECTION.
    01  PRECISION      PIC S9(9) COMP.
    01  SCALE          PIC S9(9) COMP.
    01  DISPLAY-LENGTH PIC S9(9) COMP.
    01  MAX-LENGTH     PIC S9(9) COMP VALUE 80.
    ...
PROCEDURE DIVISION.
    ...
    PERFORM ADJUST-LENGTH
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
ADJUST-LENGTH.
*   If datatype is NUMBER, extract precision and scale.
    IF SELDVTYP(J) = 2
```

```
        CALL "SQLPRC" USING SELDVLN(J), PRECISION, SCALE.
    MOVE 0 TO DISPLAY-LENGTH.
*   Precision is set to zero if the select-list item
*   refers to a NUMBER of unspecified size.  We allow for
*   a maximum precision of 10.
    IF SELDVTYP(J) = 2 AND PRECISION = 0
        MOVE 10 TO DISPLAY-LENGTH.
*   Allow for possible decimal point and sign.
    IF SELDVTYP(J) = 2 AND PRECISION > 0
        ADD 2 TO PRECISION
        MOVE PRECISION TO DISPLAY-LENGTH.
    ...
```

Notice that the first parameter in the subroutine call is the Jth element in the table of select-list lengths.

The SQLPRC procedure, defined in the SQLLIB runtime library, returns zero as the precision and scale values for certain SQL datatypes. The SQLPR2 procedure is similar to SQLPRC in that it has the same syntax and returns the same binary values, except for the datatypes shown in Table 12–4

.

Table 12–4   Datatype Exceptions to the SQLPR2 Proceure

SQL Datatype	Binary Precision	Binary Scale
FLOAT	126	-127
FLOAT(*n*)	*n* (range is 1 .. 126)	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

Handling Null/Not Null Datatypes

For every select-list column (not expression), DESCRIBE SELECT LIST returns a null/not null indication in the datatype table of the select descriptor. If the Jth select-list column is constrained to be not null, the high-order bit of SELDVTYP(J) datatype variable is clear; otherwise, it is set.

Before using the datatype in an OPEN or FETCH statement, if the null status bit is set, you must clear it. Never set the bit.

You can use the library routine SQLNUL to find out if a column allows nulls, and to clear the datatype's null status bit. You call SQLNUL using the syntax

```
                       CALL "SQLNUL" USING VALUE-TYPE, TYPE-CODE, NULL-STATUS.
```

where:

VALUE-TYPE	Is a 2-byte integer variable that stores the datatype code of a select-list column.
TYPE-CODE	Is a 2-byte intger variable that returns the datatype code of the select-list column with the high-order bit cleared.
NULL-STATUS	Is an integer variable that returns the null status of the select-list column. 1 means that the column allows nulls; 0 means that it does not.

The following example shows how to use SQLNUL:

```
 WORKING-STORAGE SECTION.
     ...
*    Declare variable for subroutine call.
     01  NULL-STATUS  PIC S9(9) COMP.
     ...
 PROCEDURE DIVISION.
 MAIN.
     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
     ...
     PERFORM HANDLE-NULLS
         VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
     ...
 HANDLE-NULLS.
*    Find out if column is NOT NULL, and clear high-order bit.
     CALL "SQLNUL" USING SELDVTYP(J), SELDVTYP(J), NULL-STATUS.
*    If NULL-STATUS = 1, nulls are allowed.
```

Notice that the first and second parameters in the subroutine call are the same. Respectively, they are the datatype variable before and after its null status bit is cleared.

The Basic Steps

Method 4 can be used to process *any* dynamic SQL statement. In the example in "Using Host Tables with Method 4" on page 12-40, a query is processed so that you can see how both input and output host variables are handled.

To process the dynamic query, our example program takes the following steps:

1. Declare a host string to hold the query text.

2. Declare select and bind descriptors.

3. Set the maximum number of select-list items and place-holders that can be DESCRIBEd.

4. Initialize the select and bind descriptors.

5. Store the query text in the host string.

6. PREPARE the query from the host string.

7. DECLARE a cursor FOR the query.

8. DESCRIBE the bind variables INTO the bind descriptor.

9. Reset the number of place-holders to the number actually found by DESCRIBE.

10. Get values for the bind variables found by DESCRIBE.

11. OPEN the cursor USING the bind descriptor.

12. DESCRIBE the select list INTO the select descriptor.

13. Reset the number of select-list items to the number actually found by DESCRIBE.

14. Reset the length and datatype of each select-list item for display purposes.

15. FETCH a row from the database INTO data buffers using the select descriptor.

16. Process the select-list values returned by FETCH.

17. CLOSE the cursor when there are no more rows to FETCH.

Note: If the dynamic SQL statement is *not* a query or contains a known number of select-list items or place-holders, then some of the above steps are unnecessary.

A Closer Look at Each Step

This section discusses each step in more detail. Also, at the end of this chapter is a full-length program illustrating Method 4. With Method 4, you use the following sequence of embedded SQL statements:

```
EXEC SQL
    PREPARE <statement_name>
    FROM {:<host_string>|<string_literal>}
END-EXEC.
EXEC SQL
    DECLARE <cursor_name> CURSOR FOR <statement_name>
END-EXEC.
```

```
EXEC SQL
    DESCRIBE BIND VARIABLES FOR <statement_name>
    INTO <bind_descriptor_name>
END-EXEC.
EXEC SQL
    OPEN <cursor_name>
    [USING DESCRIPTOR <bind_descriptor_name>]
END-EXEC.
EXEC SQL
    DESCRIBE [SELECT LIST FOR] <statement_name>
    INTO <select_descriptor_name>
END-EXEC.
EXEC SQL
    FETCH <cursor_name> USING DESCRIPTOR <select_descriptor_name>
END-EXEC.
EXEC SQL
    CLOSE <cursor_name>
END-EXEC.
```

If the number of select-list items in a dynamic query is known, you can omit
DESCRIBE SELECT LIST and use the following Method 3 FETCH statement:

```
EXEC SQL FETCH <cursor_name> INTO <host_variable_list> END-EXEC.
```

Or, if the number of place-holders for bind variables in a dynamic SQL statement is
known, you can omit DESCRIBE BIND VARIABLES and use the following Method
3 OPEN statement:

```
EXEC SQL OPEN <cursor_name> [USING <host_variable_list>] END-EXEC.
```

Next, you see how these statements allow your host program to accept and process
a dynamic SQL statement using descriptors.

> **Note:** Several figures accompany the following discussion. To avoid cluttering
> the figures, it was necessary to confine descriptor tables to 3 elements and to
> limit the maximum length of names and values to 5 and 10 characters, respec-
> tively.

Declare a Host String

Your program needs a host variable to store the text of the dynamic SQL statement.
The host variable (SELECT-STMT in our example) must be declared as a character
string:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    ...
```

```
      01  SELECT-STMT  PIC X(120).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Declare the SQLDAs

Because the query in our example might contain an unknown number of select-list items or place-holders, you must declare select and bind descriptors. Instead of hardcoding the SQLDAs, you use INCLUDE to copy them into your program, as follows:

```
EXEC SQL INCLUDE SELDSC END-EXEC.
EXEC SQL INCLUDE BNDDSC END-EXEC.
```

For reference, the INCLUDEd declaration of SELDSC follows:

```
WORKING-STORAGE SECTION.
    ...
    01  SELDSC.
        05  SQLDNUM            PIC S9(9) COMP.
        05  SQLDFND            PIC S9(9) COMP.
        05  SELDVAR            OCCURS 3 TIMES.
            10 SELDV           PIC S9(9) COMP.
            10 SELDFMT          PIC S9(9) COMP.
            10 SELDVLN          PIC S9(9) COMP.
            10 SELDFMTL         PIC S9(4) COMP.
            10 SELDVTYP         PIC S9(4) COMP.
            10 SELDI            PIC S9(9) COMP.
            10 SELDH-VNAME      PIC S9(9) COMP.
            10 SELDH-MAX-VNAMEL PIC S9(4) COMP.
            10 SELDH-CUR-VNAMEL PIC S9(4) COMP.
            10 SELDI-VNAME      PIC S9(9) COMP.
            10 SELDI-MAX-VNAMEL PIC S9(4) COMP.
            10 SELDI-CUR-VNAMEL PIC S9(4) COMP.
            10 SELDFCLP         PIC S9(9) COMP.
            10 SELDFCRCP        PIC S9(9) COMP.

    01  XSELDI.
        05  SEL-DI      OCCURS 3 TIMES PIC S9(9) COMP.
    01  XSELDIVNAME.
        05  SEL-DI-VNAME  OCCURS 3 TIMES PIC X(5).
    01  XSELDV.
        05  SEL-DV      OCCURS 3 TIMES PIC X(10).
    01  XSELDHVNAME.
        05  SEL-DH-VNAME  OCCURS 3 TIMES PIC X(5).
```

Set the Maximum Number to DESCRIBE

Next, you set the maximum number of select-list items or place-holders that can be DESCRIBEd, as follows:

```
MOVE 3 TO SQLDNUM IN SELDSC.
MOVE 3 TO SQLDNUM IN BNDDSC.
```

Initialize the Descriptors

You must initialize several descriptor variables. Some require the library subroutine SQLADR.

In our example, you store the maximum lengths of name buffers in the SELDH-MAX-VNAMEL, BNDDH-MAX-VNAMEL, and BNDDI-MAX-VNAMEL tables, and use SQLADR to store the addresses of value and name buffers in the SELDV, SELDI, BNDDV, BNDDI, SELDH-VNAME, BNDDH-VNAME, and BNDDI-VNAME tables.

```
PROCEDURE DIVISION.
    ...
    PERFORM INIT-SELDSC
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
    PERFORM INIT-BNDDSC
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
    ...
INIT-SELDSC.
    MOVE SPACES TO SEL-DV(J).
    MOVE SPACES TO SEL-DH-VNAME(J).
    MOVE 5 TO SELDH-MAX-VNAMEL(J).
    CALL "SQLADR" USING SEL-DV(J), SELDV(J).
    CALL "SQLADR" USING SEL-DH-VNAME(J), SELDH-VNAME(J).
    CALL "SQLADR" USING SEL-DI(J), SELDI(J).
    ...
INIT-BNDDSC.
    MOVE SPACES TO BND-DV(J).
    MOVE SPACES TO BND-DH-VNAME(J).
    MOVE SPACES TO BND-DI-VNAME(J).
    MOVE 5 TO BNDDH-MAX-VNAMEL(J).
    MOVE 5 TO BNDDI-MAX-VNAMEL(J).
    CALL "SQLADR" USING BND-DV(J), BNDDV(J).
    CALL "SQLADR" USING BND-DH-VNAME(J), BNDDH-VNAME(J).
    CALL "SQLADR" USING BND-DI(J), BNDDI(J).
```

```
        CALL "SQLADR" USING BND-DI-VNAME(J), BNDDI-VNAME(J).
        ...
```

Figure 12–3 and Figure 12–4 represent the resulting descriptors.

Figure 12–3   Initialized Select Descriptor

Figure 12–4   Initialized Bind Descriptor

Store the Query Text in the Host String

Continuing our example, you prompt the user for a SQL statement, then store the input string in SELECT-STMT as follows:

```
DISPLAY "Enter a SELECT statement: " WITH NO ADVANCING.
ACCEPT SELECT-STMT.
```

We assume the user entered the following string:

```
SELECT ENAME, EMPNO, COMM FROM EMP WHERE COMM < :BONUS
```

PREPARE the Query from the Host String

PREPARE parses the SQL statement and gives it a name. In our example, PREPARE parses the host string SELECT-STMT and gives it the name SQL-STMT, as follows:

```
EXEC SQL PREPARE SQL-STMT FROM :SELECT-STMT END-EXEC.
```

DECLARE a Cursor

DECLARE CURSOR defines a cursor by giving it a name and associating it with a specific SELECT statement.

To declare a cursor for static queries, you use the following syntax:

```
EXEC SQL DECLARE cursor_name CURSOR FOR SELECT ...
```

To declare a cursor for dynamic queries, the statement name given to the dynamic query by PREPARE is substituted for the static query. In our example, DECLARE CURSOR defines a cursor named EMP-CURSOR and associates it with SQL-STMT, as follows:

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR SQL-STMT END-EXEC.
```

Note: You must declare a cursor for all dynamic SQL statements, not just queries. With non-queries, OPENing the cursor executes the dynamic SQL statement.

DESCRIBE the Bind Variables

DESCRIBE BIND VARIABLES puts descriptions of bind variables into a bind descriptor. In our example, DESCRIBE readies BNDDSC as follows:

```
EXEC SQL
    DESCRIBE BIND VARIABLES FOR SQL-STMT
    INTO BNDDSC
```

```
END-EXEC.
```

Note that BNDDSC must *not* be prefixed with a colon.

The DESCRIBE BIND VARIABLES statement must follow the PREPARE statement but precede the OPEN statement.

Figure 12–5 shows the bind descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of place-holders found in the processed SQL statement.

Figure 12–5   Bind Descriptor after the DESCRIBE
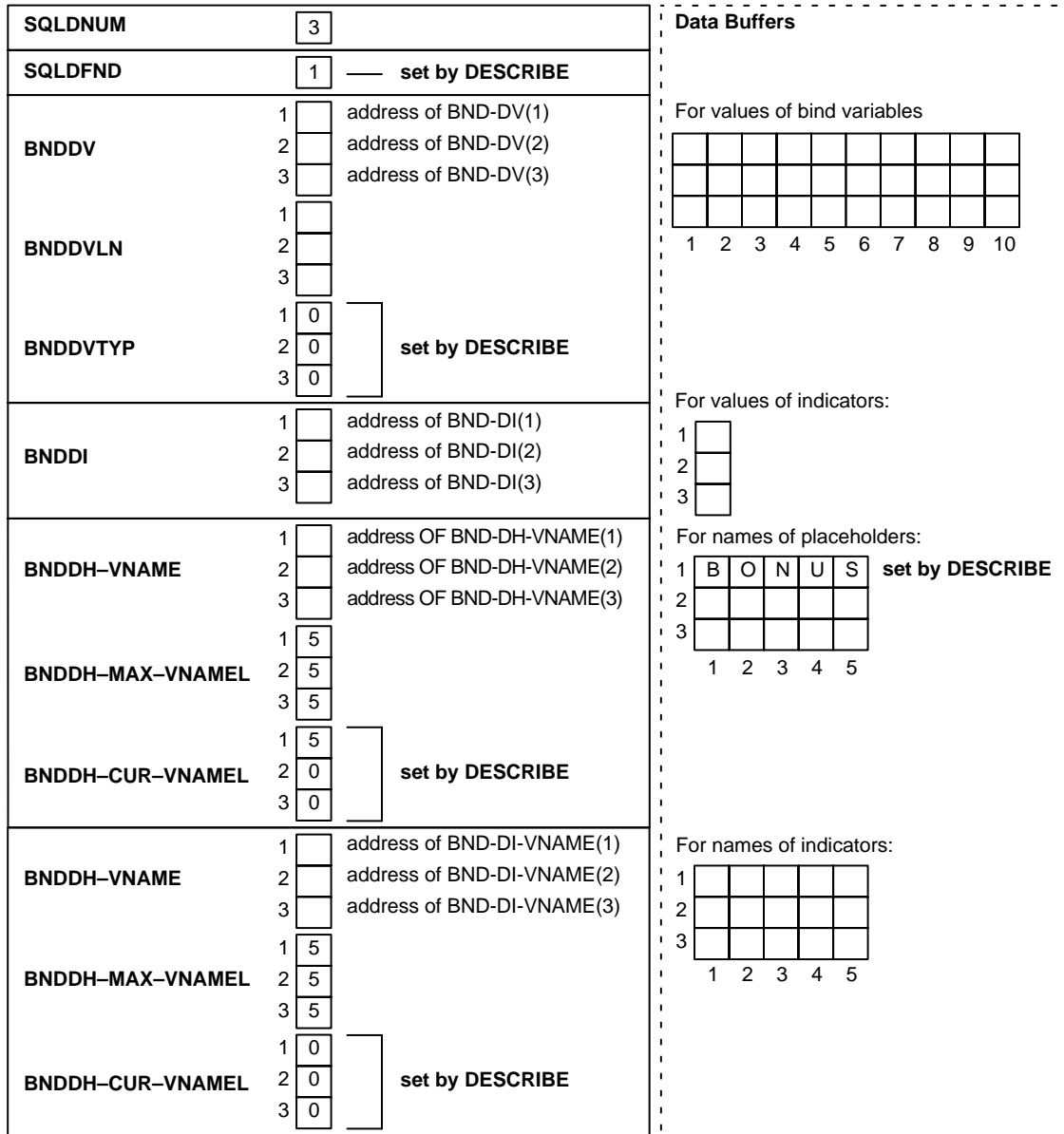
Reset Number of place-holders

Next, you must reset the maximum number of place-holders to the number actually found by DESCRIBE, as follows:

```
IF SQLDFND IN BNDDSC < 0
    DISPLAY "Too many bind variables"
    GOTO ROLL-BACK
ELSE
    MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC
END-IF.
```

Get Values for Bind Variables

Your program must get values for the bind variables in the SQL statement. How the program gets the values is up to you. For example, they can be hardcoded, read from a file, or entered interactively.

In our example, a value must be assigned to the bind variable that replaces the place-holder BONUS in the query WHERE clause. Prompt the user for the value, then process it, as follows:

```
PROCEDURE DIVISION.
    ...
    PERFORM GET-INPUT-VAR
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
    ...
GET-INPUT-VAR.
    ...
*   Replace the 0 DESCRIBEd into the datatype table
*   with a 1 to avoid an "invalid datatype" Oracle error.
    MOVE 1 TO BNDDVTYP(J).
*   Get value of bind variable.
    DISPLAY "Enter value of ", BND-DH-VNAME(J).
    ACCEPT INPUT-STRING.
    UNSTRING INPUT-STRING DELIMITED BY " "
        INTO BND-DV(J) COUNT IN BNDDVLN(J).
```

Assuming that the user supplied a value of 625 for BONUS, the next table shows the resulting bind descriptor.

Figure 12–6   Bind Descriptor after Assigning Values

SQLDNUM	1	—— **reset by program**	
SQLDFND	1		

BNDDV
- 1 [ ] address of BND-DV(1)
- 2 [ ] address of BND-DV(2)
- 3 [ ] address of BND-DV(3)

BNDDVLN
- 1 [3] —— **set by program**
- 2 [ ]
- 3 [ ]

BNDDVTYP
- 1 [1] —— **reset by program**
- 2 [0]
- 3 [0]

BNDDI
- 1 [ ] address of BND-DI(1)
- 2 [ ] address of BND-DI(2)
- 3 [ ] address of BND-DI(3)

BNDDH–VNAME
- 1 [ ] address of BND-DH-VNAME(1)
- 2 [ ] address of BND-DH-VNAME(2)
- 3 [ ] address of BND-DH-VNAME(3)

BNDDH–MAX–VNAMEL
- 1 [5]
- 2 [5]
- 3 [5]

BNDDH–CUR–VNAMEL
- 1 [5]
- 2 [0]
- 3 [0]

BNDDH–VNAME
- 1 [ ] address of BND-DI-VNAME(1)
- 2 [ ] address of BND-DI-VNAME(2)
- 3 [ ] address of BND-DI-VNAME(3)

BNDDH–MAX–VNAMEL
- 1 [5]
- 2 [5]
- 3 [5]

BNDDH–CUR–VNAMEL
- 1 [0]
- 2 [0]
- 3 [0]

Data Buffers

For values of bind variables:

6	2	5							

1 2 3 4 5 6 7 8 9 10

For values of indicators:

1 [0] —— **set by program**
2 [ ]
3 [ ]

For names of placeholders:

1	B	O	N	U	S
2					
3					

1 2 3 4 5

For names of indicators:

1					
2					
3					

1 2 3 4 5

OPEN the Cursor

The OPEN statement for dynamic queries is similar to the one for static queries, except the cursor is associated with a bind descriptor. Values determined at run time and stored in buffers addressed by elements of the bind descriptor tables are used to evaluate the SQL statement. With queries, the values are also used to identify the active set.

In our example, OPEN associates EMP-CURSOR with BNDDSC as follows:

```
EXEC SQL
    OPEN EMP-CUR USING DESCRIPTOR BNDDSC
END-EXEC.
```

Remember, BNDDSC must *not* be prefixed with a colon.

Then, OPEN executes the SQL statement. With queries, OPEN also identifies the active set and positions the cursor at the first row.

DESCRIBE the Select List

If the dynamic SQL statement is a query, the DESCRIBE SELECT LIST statement must follow the OPEN statement but precede the FETCH statement.

DESCRIBE SELECT LIST puts descriptions of select-list items into a select descriptor. In our example, DESCRIBE readies SELDSC as follows:

```
EXEC SQL
    DESCRIBE SELECT LIST FOR SQL-STMT INTO SELDSC
END-EXEC.
```

Accessing the Oracle8 data dictionary, DESCRIBE sets the length and datatype of each select-list value.

Figure 12–7 shows the select descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of items found in the query select list. If the SQL statement is not a query, SQLDFND is set to zero.

Also notice that the NUMBER lengths are not usable yet. For columns defined as NUMBER, you must use the library subroutine SQLPRC to extract precision and scale. See the section "Coercing Datatypes" on page 12-18.

Figure 12–7   Select Descriptor after the DESCRIBE
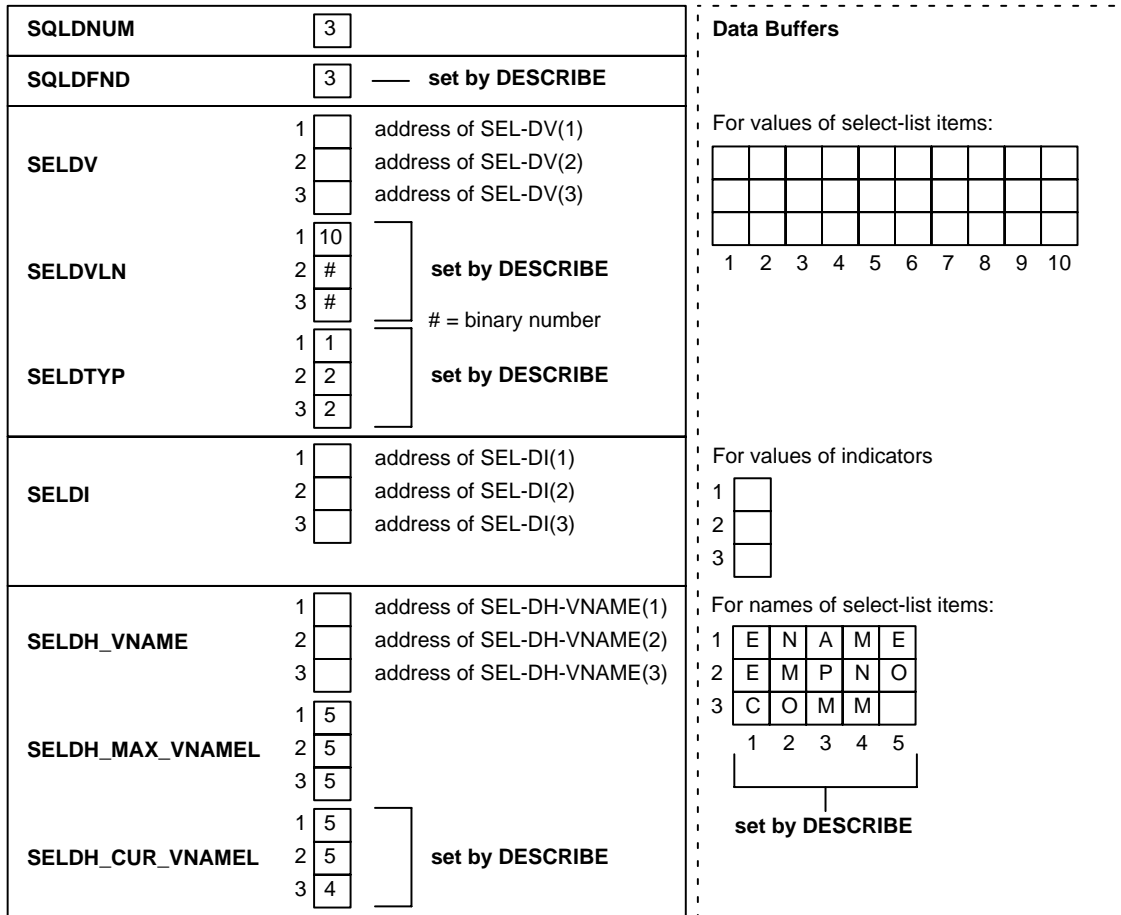
Reset Number of Select-List Items

Next, you must reset the maximum number of select-list items to the number actually found by DESCRIBE, as follows:

```
MOVE SQLDFND IN SELDSC TO SQLDNUM IN SELDSC.
```

Reset Length/Datatype of Each Select-List Item

In our example, before fetching the select-list values, you reset some elements in the length and datatype tables for display purposes.

```
PROCEDURE DIVISION.
    ...
    PERFORM COERCE-COLUMN-TYPE
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
    ...
 COERCE-COLUMN-TYPE.
*    Clear NULL bit.
     CALL "SQLNUL" USING SELDVTYP(J), SELDVTYP(J), NULL-STATUS.

*    If datatype is DATE, lengthen to 9 characters.
     IF SELDVTYP(J) = 12
         MOVE 9 TO SELDVLN(J).

*    If datatype is NUMBER, extract precision and scale.
     MOVE 0 TO DISPLAY-LENGTH.
     IF SELDVTYP(J) = 2 AND PRECISION = 0
        MOVE 10 TO DISPLAY-LENGTH.
     IF SELDVTYP(J) = 2 AND PRECISION > 0
        ADD 2 TO PRECISION
        MOVE PRECISION TO DISPLAY-LENGTH.
     IF SELDVTYP(J) = 2
        IF DISPLAY-LENGTH > MAX-LENGTH
            DISPLAY "Column value too large for data buffer."
            GO TO END-PROGRAM
        ELSE
            MOVE DISPLAY-LENGTH TO SELDVLN(J).

*    Coerce datatypes to VARCHAR2.
     MOVE 1 TO SELDVTYP(J).
```

Figure 12–8 shows the resulting select descriptor. Notice that the NUMBER lengths are now usable and that all the datatypes are VARCHAR2. The lengths in SELD-VLN(2) and SELDVLN(3) are 6 and 9 because we increased the DESCRIBEd lengths of 4 and 7 by 2 to allow for a possible sign and decimal point.

Figure 12–8   Select Descriptor before the FETCH

SQLDNUM	3	—— reset by program	Data Buffers

| SQLDFND | 3 | | |

SELDV
1 address of SEL-DV(1)
2 address of SEL-DV(2)
3 address of SEL-DV(3)

For values of select-list items:

1 2 3 4 5 6 7 8 9 10

SELDVLN
1 10
2 6   **reset by program**
3 6

= binary number

SELDTYP
1 1
2 1   **reset by program**
3 1

SELDI
1 address 0f SEL-DI(1)
2 address 0f SEL-DI(2)
3 address 0f SEL-DI(3)

For values of indicators:
1
2
3

SELDH_VNAME
1 address of SEL-DH-VNAME(1)
2 address of SEL-DH-VNAME(2)
3 address of SEL-DH-VNAME(3)

For names of select-list items:
1 | E | N | A | M | E
2 | E | M | P | N | O
3 | C | O | M | M |
  1 2 3 4 5

SELDH_MAX_VNAMEL
1 5
2 5
3 5

SELDH_CUR_VNAMEL
1 5
2 5
3 4

FETCH Rows from the Active Set

FETCH returns a row from the active set, stores select-list values in the data buffers, and advances the cursor to the next row in the active set. If there are no more rows, FETCH sets SQLCODE in the SQLCA, the SQLCODE variable, or the SQL-STATE variable to the "no data found" Oracle8 error code. In the following example, FETCH returns the values of columns ENAME, EMPNO, and COMM to SELDSC:

```
EXEC SQL
    FETCH EMP-CURSOR USING DESCRIPTOR SELDSC
END-EXEC.
```

Figure 12–9 shows the select descriptor in our example after the FETCH. Notice that Oracle8 has stored the select-list and indicator values in the data buffers addressed by the elements of SELDV and SELDI.

For output buffers of datatype 1, Oracle8, using the lengths stored in SELDVLN, left-justifies CHAR or VARCHAR2 data, and right-justifies NUMBER data.

The value "MARTIN" was retrieved from a VARCHAR2(10) column in the EMP table. Using the length in SELDVLN(1), Oracle8 left-justifies the value in a 10-byte field, filling the buffer.

The value 7654 was retrieved from a NUMBER(4) column and coerced to "7654." However, the length in SELDVLN(2) was increased by two to allow for a possible sign and decimal point, so Oracle8 right-justifies the value in a 6-byte field.

The value 482.50 was retrieved from a NUMBER(7,2) column and coerced to "482.50." Again, the length in SELDVLN(3) was increased by two, so Oracle8 right-justifies the value in a 9-byte field.

Get and Process Select-List Values

After the FETCH, your program can process the select-list values returned by FETCH. In our example, values for columns ENAME, EMPNO, and COMM are processed.

CLOSE the Cursor

CLOSE disables the cursor. In our example, CLOSE disables EMP-CURSOR as follows:

```
EXEC SQL CLOSE EMP-CURSOR END-EXEC.
```

Figure 12–9   Select Descriptor after the FETCH
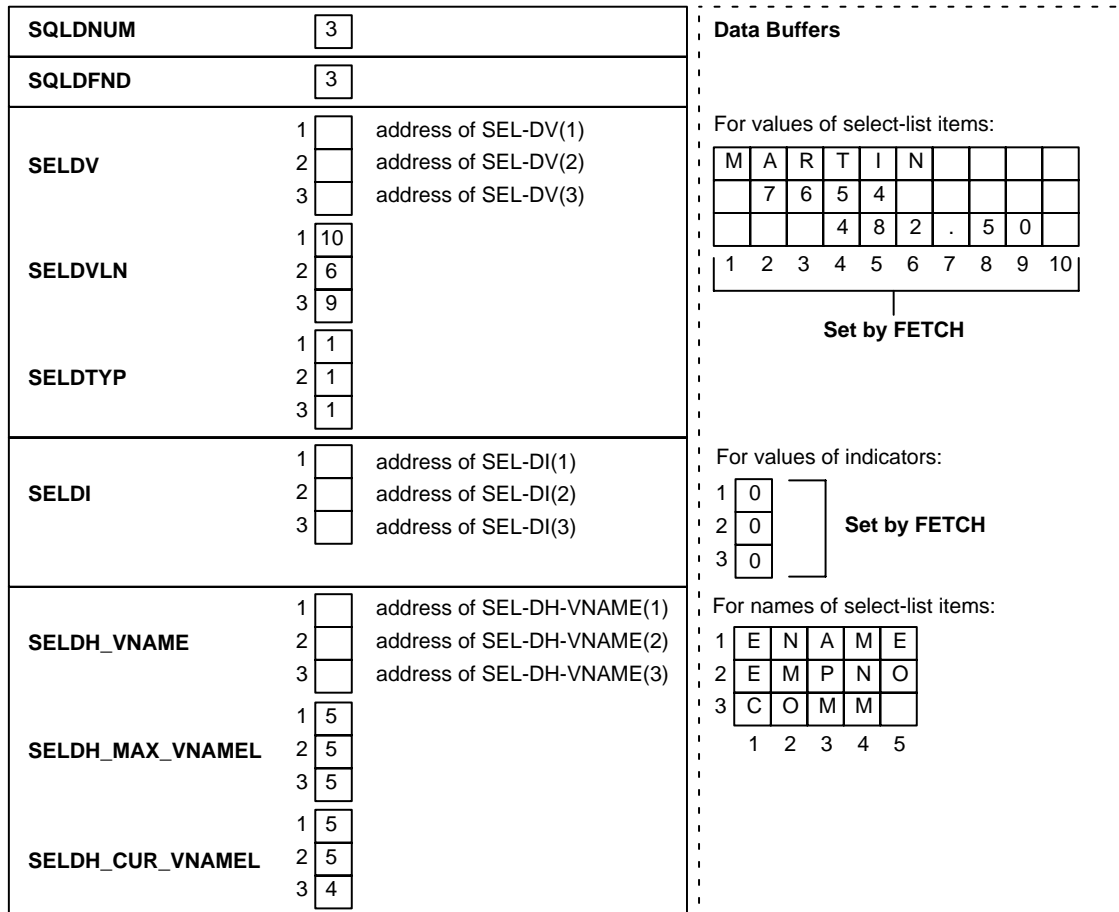
Using Host Tables with Method 4

To use input or output host tables with Method 4, you must use the optional FOR clause to tell Oracle8 the size of your host table. For more information about the FOR clause, see Chapter 10, "Using Host Tables".

Set descriptor entries for the Jth select-list item or bind variable, but instead of addressing a single data buffer, SELDVLN(J) or BNDDVLN(J) addresses a table of

data buffers. Then use a FOR clause in the EXECUTE or FETCH statement, as appropriate, to tell Oracle8 the number of table elements you want to process.

This procedure is necessary, because Oracle8 has no other way of knowing the size of your host table.

In the example below, two input host tables are used to insert 8 pairs of values of EMPNO and DEPTNO into the table EMP. Note that EXECUTE can be used for non-queries with Method 4.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DYN4INS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   BNDDSC.
     02   SQLDNUM            PIC S9(9) COMP VALUE 2.
     02   SQLDFND            PIC S9(9) COMP.
     02   BNDDVAR            OCCURS 2 TIMES.
          03 BNDDV           PIC S9(9) COMP.
          03 BNDDFMT         PIC S9(9) COMP.
          03 BNDDVLN         PIC S9(9) COMP.
          03 BNDDFMTL        PIC S9(4) COMP.
          03 BNDDVTYP        PIC S9(4) COMP.
          03 BNDDI           PIC S9(9) COMP.
          03 BNDDH-VNAME     PIC S9(9) COMP.
          03 BNDDH-MAX-VNAMEL  PIC S9(4) COMP.
          03 BNDDH-CUR-VNAMEL  PIC S9(4) COMP.
          03 BNDDI-VNAME     PIC S9(9) COMP.
          03 BNDDI-MAX-VNAMEL  PIC S9(4) COMP.
          03 BNDDI-CUR-VNAMEL  PIC S9(4) COMP.
          03 BNDDFCLP        PIC S9(9) COMP.
          03 BNDDFCRCP       PIC S9(9) COMP.
 01   XBNDDI.
      03  BND-DI             OCCURS 2 TIMES PIC S9(4) COMP.
 01   XBNDDIVNAME.
      03  BND-DI-VNAME       OCCURS 2 TIMES PIC X(80).
 01   XBNDDV.
*     Since you know what the SQL statement will be, you can set
*     up a two-dimensional table with a maximum of 2 columns and
*     8 rows.  Each element can be up to 10 characters long.  (You
*     can alter these values according to your needs.)
      03  BND-COLUMN         OCCURS 2 TIMES.
          05  BND-ELEMENT    OCCURS 8 TIMES PIC X(10).
 01   XBNDDHVNAME.
      03  BND-DH-VNAME       OCCURS 2 TIMES PIC X(80).
```

```
              01  COLUMN-INDEX            PIC 999.
              01  ROW-INDEX               PIC 999.
              01  DUMMY-INTEGER           PIC 9999.
                  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
                      01  USERNAME        PIC X(20).
                      01  PASSWD          PIC X(20).
                      01  DYN-STATEMENT   PIC X(80).
                      01  NUMBER-OF-ROWS  PIC S9(4) COMP.
                  EXEC SQL END DECLARE SECTION END-EXEC.

                  EXEC SQL INCLUDE SQLCA END-EXEC.

           PROCEDURE DIVISION.
           START-MAIN.

                  EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

                  MOVE "SCOTT" TO USERNAME.
                  MOVE "TIGER" TO PASSWD.
                  EXEC SQL
                      CONNECT :USERNAME IDENTIFIED BY :PASSWD
                  END-EXEC.
                  DISPLAY "Connected to Oracle".

       *      Initialize bind and select descriptors.
                  PERFORM INIT-BNDDSC THRU INIT-BNDDSC-EXIT
                      VARYING COLUMN-INDEX FROM 1 BY 1
                      UNTIL COLUMN-INDEX > 2.

       *      Set up the SQL statement.
                  MOVE SPACES TO DYN-STATEMENT.
                  MOVE "INSERT INTO EMP(EMPNO, DEPTNO) VALUES(:EMPNO,:DEPTNO)"
                      TO DYN-STATEMENT.
                  DISPLAY DYN-STATEMENT.

       *      Prepare the SQL statement.
                  EXEC SQL
                      PREPARE S1 FROM :DYN-STATEMENT
                  END-EXEC.

       *      Describe the bind variables.
                  EXEC SQL
                      DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC
                  END-EXEC.
```

```
        PERFORM Z-BIND-TYPE THRU Z-BIND-TYPE-EXIT
            VARYING COLUMN-INDEX FROM 1 BY 1
            UNTIL COLUMN-INDEX > 2.

        IF SQLDFND IN BNDDSC < 0
            DISPLAY "TOO MANY BIND VARIABLES."
            GO TO SQL-ERROR
        ELSE
            DISPLAY "BIND VARS = " WITH NO ADVANCING
            MOVE SQLDFND IN BNDDSC TO DUMMY-INTEGER
            DISPLAY DUMMY-INTEGER
            MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC.

            MOVE 8 TO NUMBER-OF-ROWS.
            PERFORM GET-ALL-VALUES THRU GET-ALL-VALUES-EXIT
                VARYING ROW-INDEX FROM 1 BY 1
                UNTIL ROW-INDEX > NUMBER-OF-ROWS.

*    Execute the SQL statement.
        EXEC SQL FOR :NUMBER-OF-ROWS
            EXECUTE S1 USING DESCRIPTOR BNDDSC
        END-EXEC.

        DISPLAY "INSERTED " WITH NO ADVANCING.
        MOVE SQLERRD(3) TO DUMMY-INTEGER.
        DISPLAY DUMMY-INTEGER WITH NO ADVANCING.
        DISPLAY " ROWS.".
        GO TO END-SQL.

  SQL-ERROR.
*    Display any SQL error message and code.
        DISPLAY SQLERRMC.
        EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
        STOP RUN.

  END-SQL.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        EXEC SQL COMMIT WORK RELEASE END-EXEC.
        STOP RUN.

  INIT-BNDDSC.
*    Start of COBOL PERFORM procedures, initialize the bind
*    descriptor.
        MOVE 80 TO BNDDH-MAX-VNAMEL(COLUMN-INDEX).
        CALL "SQLADR" USING
```

```
                    BND-DH-VNAME(COLUMN-INDEX)
                    BNDDH-VNAME(COLUMN-INDEX).
            MOVE 80 TO BNDDI-MAX-VNAMEL(COLUMN-INDEX).
            CALL "SQLADR" USING
                    BND-DI-VNAME(COLUMN-INDEX)
                    BNDDI-VNAME (COLUMN-INDEX).
            MOVE 10 TO BNDDVLN(COLUMN-INDEX).
            CALL "SQLADR" USING
                    BND-ELEMENT(COLUMN-INDEX,1)
                    BNDDV(COLUMN-INDEX).
            MOVE ZERO TO BNDDI(COLUMN-INDEX).
            CALL "SQLADR" USING
                    BND-DI(COLUMN-INDEX)
                    BNDDI(COLUMN-INDEX).
            MOVE ZERO TO BNDDFMT(COLUMN-INDEX).
            MOVE ZERO TO BNDDFMTL(COLUMN-INDEX).
            MOVE ZERO TO BNDDFCLP(COLUMN-INDEX).
            MOVE ZERO TO BNDDFCRCP(COLUMN-INDEX).
     INIT-BNDDSC-EXIT.
            EXIT.


     Z-BIND-TYPE.
*        Replace the 0s DESCRIBEd into the datatype table with 1s to
*        avoid an "invalid datatype" Oracle error.
            MOVE 1 TO BNDDVTYP(COLUMN-INDEX).

     Z-BIND-TYPE-EXIT.
            EXIT.

     GET-ALL-VALUES.
*        Get the bind variables for each row.
            DISPLAY "ENTER VALUES FOR ROW NUMBER ",ROW-INDEX.
            PERFORM GET-BIND-VARS
                    VARYING COLUMN-INDEX FROM 1 BY 1
                    UNTIL COLUMN-INDEX > SQLDFND IN BNDDSC.
     GET-ALL-VALUES-EXIT.
            EXIT.

     GET-BIND-VARS.
*        Get the value of each bind variable.
            DISPLAY "    ENTER VALUE FOR ",BND-DH-VNAME(COLUMN-INDEX)
                    WITH NO ADVANCING.
            ACCEPT BND-ELEMENT(COLUMN-INDEX,ROW-INDEX).
     GET-BIND-VARS-EXIT.
            EXIT.
```

Sample Program 10: Dynamic SQL Method 4

This program shows the basic steps required to use dynamic SQL Method 4. After logging on to Oracle8, the program prompts the user for a SQL statement, PRE-PAREs the statement, DECLAREs a cursor, checks for any bind variables using DESCRIBE BIND, OPENs the cursor, and DESCRIBEs any select-list variables. If the input SQL statement is a query, the program FETCHes each row of data, then CLOSEs the cursor.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  DYNSQL4.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   BNDDSC.
     02   SQLDNUM             PIC S9(9) COMP VALUE 20.
     02   SQLDFND             PIC S9(9) COMP.
     02   BNDDVAR             OCCURS 20 TIMES.
         03   BNDDV           PIC S9(9) COMP.
         03   BNDDFMT         PIC S9(9) COMP.
         03   BNDDVLN         PIC S9(9) COMP.
         03   BNDDFMTL        PIC S9(4) COMP.
         03   BNDDVTYP        PIC S9(4) COMP.
         03   BNDDI           PIC S9(9) COMP.
         03   BNDDH-VNAME     PIC S9(9) COMP.
         03   BNDDH-MAX-VNAMEL  PIC S9(4) COMP.
         03   BNDDH-CUR-VNAMEL  PIC S9(4) COMP.
         03   BNDDI-VNAME     PIC S9(9) COMP.
         03   BNDDI-MAX-VNAMEL  PIC S9(4) COMP.
         03   BNDDI-CUR-VNAMEL  PIC S9(4) COMP.
         03   BNDDFCLP        PIC S9(9) COMP.
         03   BNDDFCRCP       PIC S9(9) COMP.
01   XBNDDI.
         03   BND-DI          OCCURS 20 TIMES PIC S9(4) COMP.
01   XBNDDIVNAME.
         03   BND-DI-VNAME    OCCURS 20 TIMES PIC X(80).
01   XBNDDV.
         03   BND-DV          OCCURS 20 TIMES PIC X(80).
01   XBNDDHVNAME.
         03   BND-DH-VNAME    OCCURS 20 TIMES PIC X(80).
01   SELDSC.
     02   SQLDNUM             PIC S9(9) COMP VALUE 20.
     02   SQLDFND             PIC S9(9) COMP.
```

```
        02  SELDVAR              OCCURS 20 TIMES.
            03  SELDV            PIC S9(9) COMP.
            03  SELDFMT          PIC S9(9) COMP.
            03  SELDVLN          PIC S9(9) COMP.
            03  SELDFMTL         PIC S9(4) COMP.
            03  SELDVTYP         PIC S9(4) COMP.
            03  SELDI            PIC S9(9) COMP.
            03  SELDH-VNAME      PIC S9(9) COMP.
            03  SELDH-MAX-VNAMEL PIC S9(4) COMP.
            03  SELDH-CUR-VNAMEL PIC S9(4) COMP.
            03  SELDI-VNAME      PIC S9(9) COMP.
            03  SELDI-MAX-VNAMEL PIC S9(4) COMP.
            03  SELDI-CUR-VNAMEL PIC S9(4) COMP.
            03  SELDFCLP         PIC S9(9) COMP.
            03  SELDFCRCP        PIC S9(9) COMP.
    01  XSELDI.
            03  SEL-DI           OCCURS 20 TIMES PIC S9(4) COMP.
    01  XSELDIVNAME.
            03  SEL-DI-VNAME     OCCURS 20 TIMES PIC X(80).
    01  XSELDV.
            03  SEL-DV           OCCURS 20 TIMES PIC X(80).
    01  XSELDHVNAME.
            03  SEL-DH-VNAME     OCCURS 20 TIMES PIC X(80).
    01  TABLE-INDEX             PIC 9(3).
    01  VAR-COUNT              PIC 9(2).
    01  ROW-COUNT             PIC 9(4).
    01  NO-MORE-DATA         PIC X(1) VALUE "N".
    01  NULLS-ALLOWED       PIC S9(9) COMP.
    01  PRECISION           PIC S9(9) COMP.
    01  SCALE               PIC S9(9) COMP.
    01  DISPLAY-LENGTH      PIC S9(9) COMP.
    01  MAX-LENGTH          PIC S9(9) COMP VALUE 80.
    01  COLUMN-NAME         PIC X(30).

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  USERNAME        PIC X(20).
        01  PASSWD          PIC X(20).
        01  DYN-STATEMENT   PIC X(80).
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
PROCEDURE DIVISION.
START-MAIN.
    EXEC SQL
        WHENEVER SQLERROR GOTO SQL-ERROR
    END-EXEC.
```

```
        DISPLAY "USERNAME: " WITH NO ADVANCING.
        ACCEPT USERNAME.
        DISPLAY "PASSWORD: " WITH NO ADVANCING.
        ACCEPT PASSWD.
        EXEC SQL
            CONNECT :USERNAME IDENTIFIED BY :PASSWD
        END-EXEC.
        DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME.

*       INITIALIZE THE BIND AND SELECT DESCRIPTORS.
        PERFORM INIT-BNDDSC
            VARYING TABLE-INDEX FROM 1 BY 1
            UNTIL TABLE-INDEX > 20.
        PERFORM INIT-SELDSC
            VARYING TABLE-INDEX FROM 1 BY 1
            UNTIL TABLE-INDEX > 20.

*       GET A SQL STATEMENT FROM THE OPERATOR.
        DISPLAY "ENTER SQL STATEMENT WITHOUT TERMINATOR:".
        DISPLAY ">" WITH NO ADVANCING.
        ACCEPT DYN-STATEMENT.
        DISPLAY " ".

*       PREPARE THE SQL STATEMENT AND DECLARE A CURSOR.
        EXEC SQL
            PREPARE S1 FROM :DYN-STATEMENT
        END-EXEC.
        EXEC SQL
            DECLARE C1 CURSOR FOR S1
        END-EXEC.

*       DESCRIBE ANY BIND VARIABLES.
        EXEC SQL
            DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC
        END-EXEC.
        IF SQLDFND IN BNDDSC < 0
            DISPLAY "TOO MANY BIND VARIABLES."
            GO TO END-SQL
        ELSE
            DISPLAY "NUMBER OF BIND VARIABLES: " WITH NO ADVANCING
            MOVE SQLDFND IN BNDDSC TO VAR-COUNT
            DISPLAY VAR-COUNT
            MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC
        END-IF.
*       REPLACE THE 0S DESCRIBED INTO THE DATATYPE FIELDS OF THE BIND
```

```
*     DESCRIPTOR WITH 1S TO AVOID AN ORACLE "INVALID DATATYPE"
*     ERROR.
      MOVE 1 TO TABLE-INDEX.
 FIX-BIND-TYPE.
      MOVE 1 TO BNDDVTYP(TABLE-INDEX).
      ADD 1 TO TABLE-INDEX.
      IF TABLE-INDEX <= 20
         GO TO FIX-BIND-TYPE.

*     LET THE USER FILL IN THE BIND VARIABLES.
      IF SQLDFND IN BNDDSC = 0
         GO TO DESCRIBE-ITEMS.
      MOVE 1 TO TABLE-INDEX.
 GET-BIND-VAR.
      DISPLAY "ENTER VALUE FOR ", BND-DH-VNAME(TABLE-INDEX).
      ACCEPT BND-DV(TABLE-INDEX).
      ADD 1 TO TABLE-INDEX.
      IF TABLE-INDEX <= SQLDFND IN BNDDSC
         GO TO GET-BIND-VAR.

 DESCRIBE-ITEMS.
*     OPEN THE CURSOR AND DESCRIBE THE SELECT-LIST ITEMS.
      EXEC SQL
          OPEN C1 USING DESCRIPTOR BNDDSC
      END-EXEC.
      EXEC SQL
          DESCRIBE SELECT LIST FOR S1 INTO SELDSC
      END-EXEC.

      IF SQLDFND IN SELDSC < 0
          DISPLAY "TOO MANY SELECT-LIST ITEMS."
          GO TO END-SQL
      ELSE
          DISPLAY "NUMBER OF SELECT-LIST ITEMS: "
              WITH NO ADVANCING
          MOVE SQLDFND IN SELDSC TO VAR-COUNT
          DISPLAY VAR-COUNT
          DISPLAY " "
          MOVE SQLDFND IN SELDSC TO SQLDNUM IN SELDSC
      END-IF.

*     COERCE THE DATATYPE OF ALL SELECT-LIST ITEMS TO VARCHAR2.
      IF SQLDNUM IN SELDSC > 0
          PERFORM COERCE-COLUMN-TYPE
          VARYING TABLE-INDEX FROM 1 BY 1
```

```
                    UNTIL TABLE-INDEX > SQLDNUM IN SELDSC
              DISPLAY " ".

*     FETCH EACH ROW AND PRINT EACH SELECT-LIST VALUE.
      IF SQLDNUM IN SELDSC > 0
          PERFORM FETCH-ROWS UNTIL NO-MORE-DATA = "Y".
      DISPLAY " "
      DISPLAY "NUMBER OF ROWS PROCESSED: " WITH NO ADVANCING.
      MOVE SQLERRD(3) TO ROW-COUNT.
      DISPLAY ROW-COUNT.

*     CLEAN UP AND TERMINATE.
      EXEC SQL
          CLOSE C1
      END-EXEC.
      EXEC SQL
          COMMIT WORK RELEASE
      END-EXEC.
      DISPLAY " ".
      DISPLAY "HAVE A GOOD DAY!".
      DISPLAY " ".
      STOP RUN.

 SQL-ERROR.
*     DISPLAY ORACLE ERROR MESSAGE AND CODE.
      DISPLAY " ".
      DISPLAY SQLERRMC.

 END-SQL.
      EXEC SQL
          WHENEVER SQLERROR CONTINUE
      END-EXEC.
      EXEC SQL
          ROLLBACK WORK RELEASE
      END-EXEC.
      STOP RUN.

*     PERFORMED SUBROUTINES BEGIN HERE:

 INIT-BNDDSC.
*     INITIALIZE THE BIND DESCRIPTOR.
      MOVE SPACES TO BND-DH-VNAME(TABLE-INDEX).
      MOVE 80 TO BNDDH-MAX-VNAMEL(TABLE-INDEX).
      CALL "SQLADR" USING
          BND-DH-VNAME(TABLE-INDEX)
```

```
                          BNDDH-VNAME(TABLE-INDEX).
              MOVE SPACES TO BND-DI-VNAME(TABLE-INDEX).
              MOVE 80 TO BNDDI-MAX-VNAMEL(TABLE-INDEX).
              CALL "SQLADR" USING
                  BND-DI-VNAME(TABLE-INDEX)
                  BNDDI-VNAME (TABLE-INDEX).
              MOVE SPACES TO BND-DV(TABLE-INDEX).
              MOVE 80 TO BNDDVLN(TABLE-INDEX).
              CALL "SQLADR" USING
                  BND-DV(TABLE-INDEX)
                  BNDDV(TABLE-INDEX).
              MOVE ZERO TO BND-DI(TABLE-INDEX).
              CALL "SQLADR" USING
                  BND-DI(TABLE-INDEX)
                  BNDDI(TABLE-INDEX).
              MOVE ZERO TO BNDDFMT(TABLE-INDEX).
              MOVE ZERO TO BNDDFMTL(TABLE-INDEX).
              MOVE ZERO TO BNDDFCLP(TABLE-INDEX).
              MOVE ZERO TO BNDDFCRCP(TABLE-INDEX).
              EXIT.

          INIT-SELDSC.
      *     INITIALIZE THE SELECT DESCRIPTOR.
              MOVE SPACES TO SEL-DH-VNAME(TABLE-INDEX).
              MOVE 80 TO SELDH-MAX-VNAMEL(TABLE-INDEX).
              CALL "SQLADR" USING
                  SEL-DH-VNAME(TABLE-INDEX)
                  SELDH-VNAME(TABLE-INDEX).
              MOVE SPACES TO SEL-DI-VNAME(TABLE-INDEX).
              MOVE 80 TO SELDI-MAX-VNAMEL(TABLE-INDEX).
              CALL "SQLADR" USING
                  SEL-DI-VNAME(TABLE-INDEX)
                  SELDI-VNAME (TABLE-INDEX).
              MOVE SPACES TO SEL-DV(TABLE-INDEX).
              MOVE 80 TO SELDVLN(TABLE-INDEX).
              CALL "SQLADR" USING
                  SEL-DV(TABLE-INDEX)
                  SELDV(TABLE-INDEX).
              MOVE ZERO TO SEL-DI(TABLE-INDEX).
              CALL "SQLADR" USING
                  SEL-DI(TABLE-INDEX)
                  SELDI(TABLE-INDEX).
              MOVE ZERO TO SELDFMT(TABLE-INDEX).
              MOVE ZERO TO SELDFMTL(TABLE-INDEX).
              MOVE ZERO TO SELDFCLP(TABLE-INDEX).
```

```
       MOVE ZERO TO SELDFCRCP(TABLE-INDEX).
       EXIT.

 COERCE-COLUMN-TYPE.
*      COERCE SELECT-LIST DATATYPES TO VARCHAR2.
       CALL "SQLNUL" USING
            SELDVTYP(TABLE-INDEX)
            SELDVTYP(TABLE-INDEX)
            NULLS-ALLOWED.

*      IF DATATYPE IS DATE, LENGTHEN TO 9 CHARACTERS.
       IF SELDVTYP(TABLE-INDEX) = 12
          MOVE 9 TO SELDVLN(TABLE-INDEX).

*      IF DATATYPE IS NUMBER, SET LENGTH TO PRECISION.
       IF SELDVTYP(TABLE-INDEX) = 2
            CALL "SQLPRC" USING SELDVLN(TABLE-INDEX) PRECISION SCALE.
       MOVE 0 TO DISPLAY-LENGTH.
       IF SELDVTYP(TABLE-INDEX) = 2 AND PRECISION = 0
          MOVE 40 TO DISPLAY-LENGTH.
       IF SELDVTYP(TABLE-INDEX) = 2 AND PRECISION > 0
          ADD 2 TO PRECISION
          MOVE PRECISION TO DISPLAY-LENGTH.
       IF SELDVTYP(TABLE-INDEX) = 2
          IF DISPLAY-LENGTH > MAX-LENGTH
             DISPLAY "COLUMN VALUE TOO LARGE FOR DATA BUFFER."
             GO TO END-SQL
          ELSE
             MOVE DISPLAY-LENGTH TO SELDVLN(TABLE-INDEX).

*      COERCE DATATYPES TO VARCHAR2.
       MOVE 1 TO SELDVTYP(TABLE-INDEX).

*      DISPLAY COLUMN HEADING.
       MOVE SEL-DH-VNAME(TABLE-INDEX) TO COLUMN-NAME.
       DISPLAY COLUMN-NAME(1:SELDVLN(TABLE-INDEX)), " "
          WITH NO ADVANCING.
       EXIT.

 FETCH-ROWS.
*      FETCH A ROW AND PRINT THE SELECT-LIST VALUE.
       EXEC SQL
            FETCH C1 USING DESCRIPTOR SELDSC
       END-EXEC.
       IF SQLCODE NOT = 0
```

```
                    MOVE "Y" TO NO-MORE-DATA.
             IF SQLCODE = 0
                 PERFORM PRINT-COLUMN-VALUES
                     VARYING TABLE-INDEX FROM 1 BY 1
                     UNTIL TABLE-INDEX > SQLDNUM IN SELDSC
                 DISPLAY " ".

  PRINT-COLUMN-VALUES.
*     PRINT A SELECT-LIST VALUE.
      DISPLAY SEL-DV(TABLE-INDEX)(1:SELDVLN(TABLE-INDEX)), " "
          WITH NO ADVANCING.
```

13

Writing User Exits

This chapter focuses on writing user exits for your SQL*Forms and Oracle Forms applications. First, you learn the EXEC IAF statements that allow a SQL*Forms application to interface with user exits. Then, you learn how to write and link a SQL*Forms user exit. You also learn how to use EXEC TOOLS statements with Oracle Forms. (SQL*Forms does not support EXEC TOOLS.) That way, you can use EXEC IAF statements to enhance your existing applications and EXEC TOOLS statements to build new applications.

Use EXEC TOOLS rather than the obsolescent EXEC IAF in any new applications.

The following topics are covered:

- What Is a User Exit?
- Why Write a User Exit?
- Developing a User Exit
- Writing a User Exit
- Calling a User Exit
- Passing Parameters to a User Exit
- Returning Values to a Form
- Sample Program 5: Oracle Forms User Exit
- Precompiling and Compiling a User Exit
- Using the GENXTB Utility
- Linking a User Exit into SQL*Forms
- Guidelines for SQL*Forms User Exits

- EXEC TOOLS Statements

This chapter is supplemental. For more information about user exits, see the *Oracle Forms Designer's Reference*, the *Oracle Forms Reference Manual, Vol. 2*, and your system-specific Oracle manuals.

What Is a 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 precompile 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 13–1). Your user exit can display messages on the SQL*Forms status line, get and put field values, manipulate Oracle8 data, do high-speed computations and table lookups—even log on to different databases.

*Figure 13–1   SQL*Forms Communicating with a User Exit*
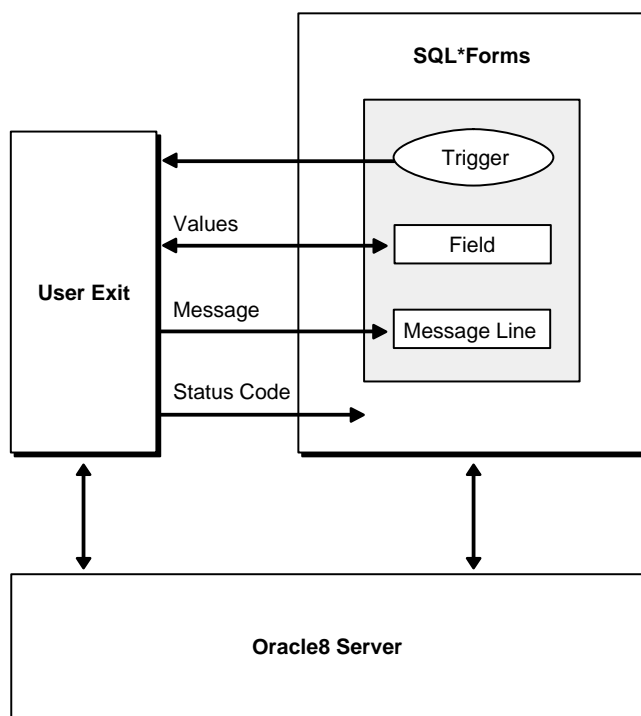
Why Write a User Exit?

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 performed in third generation languages like C and FORTRAN (for example, numeric integration)

- controlling real time devices or processes (for example, issuing a sequence of instructions to a printer or graphics device)

- data manipulations that need extended procedural capabilities (for example, recursive sorting)

- 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. For information about EXEC TOOLS statements, which are available with Oracle Forms, see "EXEC TOOLS Statements" on page 13-14.

To incorporate a user exit into a form, you take the following steps:

1. Write the user exit in a supported host language.

2. Precompile the source code.

3. Compile the modified source code.

4. Use the GENXTB utility to create a database table, IAPXTB.

5. Use the GENXTB form in SQL*Forms to insert your user exit information into the database table.

6. Use the GENXTB utility to read the information from the table and create an IAPXIT source module. Then, compile the source module.

7. Create a new IAP (the SQL*Forms component that runs a form) by linking the standard IAP object modules, your user exit object module, and the IAPXIT object module created in step 6.

8. In the form, define a trigger to call the user exit.

9. Instruct operators to use the new IAP when running the form. This is unnecessary if the new IAP replaces the standard one. For details, see your system-specific Oracle manuals.

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, you get an error. 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, ... END-EXEC.
```

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, it must be unique.

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 END-EXEC.
```

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:

```
MOVE "employee.job" TO BLKFLD.
EXEC IAF GET :BLKFLD INTO :NEW-JOB END-EXEC.
```

Unless the field is unique, the host string must contain the full *block.field* reference with intervening period. For example, the following usage is *invalid*:

```
MOVE "employee" TO BLK.
MOVE "job" TO FLD.
EXEC IAF GET :BLK.:FLD INTO :NEW-JOB END-EXEC.
```

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*:

```
MOVE "job" TO FLD.
EXEC IAF GET employee.:FLD INTO :NEW-JOB END-EXEC.
```

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, ...) END-EXEC.
```

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

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) END-EXEC.
```

Like GET, PUT lets you use a host string to specify block and field names, as follows:

```
MOVE "employee.job" TO BLKFLD.
EXEC IAF PUT :BLKFLD VALUES (:NEW-JOB) END-EXEC.
```

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:

However, the user exit string allows you to pass additional parameters to the user exit. For example, the following trigger command 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 generated by precompiler (see the next section). The three results have the following meanings:

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

The precompiler generates three symbolic constants for use as return codes. They are prefixed with IAP. For example, the three constants 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:

```
CALL "SQLIEM" USING ERROR-MESSAGE  ERROR-MESSAGE-LEN.
```

where *ERROR-MESSAGE* and *ERROR-MESSAGE-LEN* 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.

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).

Sample Program 5: Oracle Forms User Exit

This user exit concatenates form fields. To call the user exit from a Oracle Forms trigger, use the syntax

```
<user_exit>('CONCAT <field1>, <field2>, ..., <result_field>');
```

where *user_exit* is a packaged procedure supplied with Oracle Forms and CONCAT is the name of the user exit. A sample CONCAT form invokes the user exit.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  CONCAT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01  FIELD-NAME            PIC X(80)  VARYING.
        01  FIELD-VALUE           PIC X(80)  VARYING.
        01  RESULT                PIC X(800) VARYING.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
        01  EXIT-MESSAGE          PIC X(80).
        01  EXIT-MESSAGE-LEN      PIC S9(9) COMP.
        01  RTN-CODE              PIC S9(9) COMP.
            77  INDX              PIC S9(4) COMP.
        01  DONE-FLAG             PIC X.
            88  DONE              VALUE 'Y'.
        01  PTR                   PIC S9(4) COMP.
        01  WS-CMD-LINE.
            05  WS-CMD-LINE-Y     PIC X(80).
            05  WS-CMD-LINE-X     REDEFINES WS-CMD-LINE-Y
        01  WS-FIELD-NAME-AREA.
```

```
                05  WS-FIELD-NAME       PIC X(80).
                05  WS-FIELD-NAME-X     REDEFINES WS-FIELD-NAME
                                        PIC X OCCURS 80.
                05  WS-FIELD-NAME-LEN PIC S9(4) COMP.


        LINKAGE SECTION.
                01  CMD-LINE            PIC X(80).
                01  CMD-LINE-LEN        PIC S9(9) COMP.
                01  ERR-MSG             PIC X(80).
                01  ERR-MSG-LEN         PIC S9(9) COMP.
                01  IN-QUERY            PIC S9(9) COMP.
                01  RETURN-VALUE        PIC S9(9) COMP.



        PROCEDURE DIVISION USING CMD-LINE, CMD-LINE-LEN,
                            ERR-MSG, ERR-MSG-LEN,
                            IN-QUERY, RETURN-VALUE.


        MAIN.
            MOVE 1 TO PTR.
            MOVE SPACE TO RESULT-ARR.
            MOVE ZERO TO RESULT-LEN.
            MOVE SPACE TO DONE-FLAG.
            MOVE 7 TO INDX.
            MOVE CMD-LINE TO WS-CMD-LINE-Y.
            PERFORM CMD-LINE-PARSE UNTIL DONE.
            EXEC SQL
                WHENEVER SQLERROR GOTO SQL-ERROR
            END-EXEC.
            MOVE WS-FIELD-NAME TO FIELD-NAME-ARR.
            MOVE WS-FIELD-NAME-LEN TO FIELD-NAME-LEN.
            EXEC IAF
                PUT :FIELD-NAME VALUES(:RESULT)
            END-EXEC.
            MOVE SQL-IAPXIT-SUCCESS TO RTN-CODE.
            EXIT PROGRAM GIVING RTN-CODE.


        CMD-LINE-PARSE.
            MOVE ZERO TO WS-FIELD-NAME-LEN.
            MOVE SPACES TO WS-FIELD-NAME.
            MOVE SPACES TO FIELD-NAME-ARR.
            MOVE ZERO TO FIELD-NAME-LEN.
            PERFORM GET-FIELD-NAME
```

```
                    UNTIL WS-CMD-LINE-X(INDX) = ',' OR DONE.
            IF WS-CMD-LINE-X(INDX) = ','
            MOVE SPACES TO FIELD-NAME-ARR
            MOVE WS-FIELD-NAME TO FIELD-NAME-ARR
            MOVE WS-FIELD-NAME-LEN TO FIELD-NAME-LEN
            MOVE SPACES TO FIELD-VALUE-ARR
            EXEC IAF
                GET :FIELD-NAME INTO :FIELD-VALUE
            END-EXEC
                STRING FIELD-VALUE-ARR
                    DELIMITED BY SPACE
                    INTO RESULT-ARR
                    WITH POINTER PTR
                ADD FIELD-VALUE-LEN TO RESULT-LEN
                ADD 1 TO INDX.

    GET-FIELD-NAME.
        IF WS-CMD-LINE-X(INDX) NOT EQUAL SPACE
            ADD 1 TO WS-FIELD-NAME-LEN
            MOVE WS-CMD-LINE-X(INDX) TO
                WS-FIELD-NAME-X(WS-FIELD-NAME-LEN).
        ADD 1 TO INDX.
        IF INDX > CMD-LINE-LEN MOVE 'Y' TO DONE-FLAG.

    SQL-ERROR.
        EXEC SQL
            WHENEVER SQLERROR CONTINUE
        END-EXEC.
        MOVE SQLERRMC TO EXIT-MESSAGE.
        MOVE SQLERRML TO EXIT-MESSAGE-LEN.
        CALL "SQLIEM" USING EXIT-MESSAGE EXIT-MESSAGE-LEN.
        MOVE SQL-IAPXIT-FAILURE TO RTN-CODE.
        EXIT PROGRAM.
```

Precompiling and Compiling a User Exit

User exits are precompiled like stand-alone host programs. Refer to Chapter 7, "Running the Pro*COBOL Precompiler".

For instructions on compiling a user exit, see your system-specific Oracle manuals.

Using the GENXTB Utility

The IAP 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 a new 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

- host-language code (COBOL or C)

- date created

- date last modified

- Comments

After modifying the IAPXTB database table, use the GENXTB utility to read the table and create an Assembler 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 outfile
```

where *outfile* is the name you give the Assembler or 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 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, so check your system-specific Oracle manuals.

Guidelines for SQL*Forms User Exits

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 file name for your host language and operating system.

Connecting to Oracle

User exits communicate with Oracle8 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 "Concurrent Logons" on page 3-46.

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. 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 work space is applied to the table, overwriting the user exit UPDATE.

Issuing Commands

Avoid issuing a COMMIT or ROLLBACK command from your user exit because Oracle8 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 and CREATE) because they issue an implicit COMMIT before and after executing.

EXEC TOOLS Statements

EXEC TOOLS statements support the basic Oracle Toolset (Oracle Forms, Oracle Reports, and Oracle Graphics) by providing a generic way to handle get, set, and exception call-backs from user exits. The following discussion focuses on Oracle Forms but the same concepts apply to Oracle Reports and Oracle Graphics.

Besides EXEC SQL, EXEC ORACLE, and host language statements, you can use the following EXEC TOOLS statements to write an Oracle Forms user exit:

- SET

- GET

- MESSAGE

The EXEC TOOLS GET and SET statements replace the EXEC IAF GET and PUT statements used with SQL*Forms. Unlike IAF GET and PUT, TOOLS GET and SET accept indicator variables. The EXEC TOOLS MESSAGE statement replaces the message-handling function SQLIEM. The EXEC TOOLS SET CONTEXT and GET CONTEXT statements are new and not available with SQL*Forms, Version 3.

> **Note:** COBOL does not have a pointer datatype, so you cannot use the SET CONTEXT and GET CONTEXT statements in a Pro*COBOL program.

EXEC TOOLS SET

The EXEC TOOLS SET statement passes values from your user exit to Oracle Forms. Specifically, it assigns the values of host variables and constants to Oracle Forms variables and items. The values are displayed after the user exit returns control to the form.

To code the EXEC TOOLS SET statement, you use the syntax

```
EXEC TOOLS SET form_variable[, ...]
    VALUES ({:host_variable[:indicator] | constant}[, ...])
END-EXEC.
```

where *form_variable* is an Oracle Forms field, parameter, system variable, or global variable, or a host variable (prefixed with a colon) containing the name of one of the foregoing items.

In the following Pro*COBOL example, your user exit passes an employee name (with optional indicator) to Oracle Forms:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
    01  ENAME       PIC X(20) VARYING.
    01  ENAME-IND   PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE "MILLER" TO ENAME-ARR.
MOVE 6 TO ENAME-LEN.
MOVE ZERO TO ENAME-IND.
EXEC TOOLS SET emp.ename VALUES (:ENAME:ENAME-IND) END-EXEC.
```

In this example, *emp.ename* is an Oracle Forms block.field.

EXEC TOOLS GET

The EXEC TOOLS GET statement passes values from Oracle Forms to your user exit. Specifically, it assigns the values of Oracle Forms variables and items to host variables. As soon as the values are passed, the user exit can use them for any purpose.

To code the EXEC TOOLS GET statement, you use the syntax

```
EXEC TOOLS GET form_variable[, ...]
    INTO :host_variable[:indicator][, ...] END-EXEC.
```

where *form_variable* is an Oracle Forms field, parameter, system variable, or global variable, or a host variable containing the name of one of the foregoing items.

In the following example, Oracle Forms passes an employee name from the block.field *emp.ename* to your user exit:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
    01  ENAME       PIC X(20) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC TOOLS GET emp.ename INTO :ENAME END-EXEC.
...
```

EXEC TOOLS MESSAGE

The EXEC TOOLS MESSAGE statement passes a message from your user exit to Oracle Forms. The message is displayed on the Oracle Forms message line after the user exit returns control to the form.

To code the EXEC TOOLS MESSAGE statement, you use the syntax

```
EXEC TOOLS MESSAGE message_text [severity_code] END-EXEC.
```

where *message_text* is a quoted string or a character host variable, and the optional *severity_code* is an integer constant or host variable. The MESSAGE statement does *not* accept indicator variables.

In the following Pro*COBOL example, your user exit passes an error message and severity code to Oracle Forms:

```
EXEC TOOLS MESSAGE "Bad field name! Please reenter." 15
END-EXEC.
```

A

New Features

This appendix looks at the improvements and new features offered by the Oracle Pro*COBOL Precompiler, Release 8.0. Each is briefly described, and a reference to the more complete description in the chapters is provided.

These topics are presented:

- DB2 Compatibility Features
- Other New Features

DB2 Compatibility Features

These new features in Pro*COBOL 8.0 help you when migrating applications from DB2 to Oracle8, but all users of Pro*COBOL should review them.

Optional DECLARE SECTION

Use of the BEGIN DECLARE SECTION and END DECLARE SECTION statements is now optional when DECLARE_SECTION=NO (the default). If used, the DECLARE statements must be properly paired within the same WORKING-STORAGE SECTION or other COBOL declaration unit.

The form of the precompiler option is:

```
DECLARE_SECTION={YES | NO (default)}
```

which must be used on the command-line or in a configuration file. This option is a micro option with respect to the MODE option (a micro option controls only one behavior; a macro option controls several behaviors) and is subject to the general rule: macro options have precedence over micro options when, and only when, the macro level is at a higher level than the micro option. The levels are, in descending precedence:

- In-line
- Command line
- User configuration file
- System configuration file
- Default

This option allows the user to specify MODE=ORACLE together with DECLARE_SECTION=YES to get the same effect that previous releases provided when using MODE=ORACLE alone (only variables declared inside the DECLARE statements are allowed as host variables.) For further details, see "DECLARE_SECTION" on page 7-17. For further discussion of precedence of this option, and a table showing how macro option values set micro option values, see "Macro and Micro Options" on page 7-4.

Support of Additional Datatypes

The computational usage datatype COMP-4 (COMPUTATIONAL-4) is treated as a binary datatype. The IBM-implemented computational data type, COMP-4 (also represented as COMPUTATIONAL-4, will be treated as a binary datatype.

Display usage datatypes now supported are:

- Over-Punch (ZONED-DECIMAL). This is the default signed numeric for the COBOL language. Digits are held in ASCII or EBCDIC format in radix 10, with one digit per byte of computer storage. The sign is held in the high order nibble of one of the bytes. It is called overpunch because the sign is "punched-over" the digit in either the first or last byte. The default sign position will be over the trailing byte. PIC S9(n)V9(m) TRAILING or PIC S9(n)V9(m) LEADING is used to specify the overpunch.

- Display-1 Multibyte type (PIC G). This datatype is equivalent to PIC N and is used for multibyte characters.

See "Host Variables" on page 3-12.

Support of Group Items as Host Variables

Pro*COBOL now allows the use of group items in embedded SQL statements. The host group items can be referenced in the INTO clause of a SELECT or a FETCH statement, and in the VALUES list of an INSERT statement. When a group item is used as a host variable, only the group name is used in the SQL statement. For more details see "Group Items as Host Variables" on page 3-20.

Implicit Form of VARCHAR Group Items

The declaration of COBOL groups that are recognized as VARCHAR are of the following format:

```
nn    <identifier-1>
      49    <identifier-2> PIC S9(4) <integer declaration>.
      49    <identifier-3> PIC X(nc).
```

where the level, nn, is in the range 01 to 48, the length, nc, is in the range 1 to 65533, and PIC G or PIC N can be used instead of PIC X.

The VARCHAR=YES command line option must be specified for Pro*COBOL to recognize the extended form of the VARCHAR group items. Otherwise, any declarations in the above format will be interpreted as regular group items. For more details, see "Referencing VARCHAR Variables" on page 3-38.

Explicit Control Over the END_OF_FETCH SQLCODE Number

DB2 returns a SQLCODE value of 100 when an end-of-fetch condition occurs. To provide explicit control over the value returned by Oracle, the following option is available:

```
END_OF_FETCH={100 | 1403 (default)}
```

This option must be used on the command line or in a configuration file. For more details see "END_OF_FETCH" on page 7-19.

Support of the WITH HOLD Clause in the DECLARE CURSOR Statement

DB2 closes all cursors on commit, by default. This can be overridden on a cursor (which has been declared as for update) by using the WITH HOLD clause in the declaration of the cursor. Any cursor with the WITH HOLD clause will remain open after a commit or a rollback. The DB2 default occurs when MODE=ANSI, but then all host variables must be declared in a declare section. See "Declaring a Cursor" on page 5-11.

New Precompiler Option CLOSE_ON_COMMIT

A new precompiler option is provided:

```
CLOSE_ON_COMMIT={YES | NO (default)}
```

This option must be used on the command line or in a configuration file. It will only have an effect when a cursor is not coded using the WITH HOLD clause, since that will override both the new option and the existing behavior which is associated with MODE option. For more details, see "CLOSE_ON_COMMIT" on page 7-14

Support for DSNTIAR

DB2 provides a routine DSNTIAR to obtain a form of the SQLCA that can be displayed. Pro*COBOL now provides DSNTIAR. The interface is:

```
CALL "DSNTIAR" USING SQLCA MESSAGE LRECL.
```

where SQLCA is a SQL communication area, MESSAGE is the output message area, in VARCHAR form of size greater than or equal to 240, and LRECL is a full-word containing the length of the output messages, between 72 and 240. For more details, see "DSNTIAR" on page 9-27.

Date String Format Precompiler Option

For compatibility with DB2, Pro*COBOL now provides the following precompiler option to specify date strings:

```
DATE_FORMAT={ISO | USA | EUR | JIS | LOCAL | 'fmt' (default LOCAL)}
```

The DATE_FORMAT option must be used on the command line or in a configuration file. The date strings are shown in the following table:

Table A–1   Formats for Date Strings

Format Name	Abbreviation	Date Format
International Standards Organization	ISO	yyyy-mm-dd
USA standard	USA	mm/dd/yyyy
European standard	EUR	dd.mm.yyyy
Japanese Industrial Standard	JIS	yyyy-mm-dd
installation-defined	LOCAL	Any installation-defined form.

'*fmt*' is a date format model, such as 'Month dd, yyyy'. See the *Oracle8 SQL Reference* for the list of date format model elements. For more details, see "DATE_FORMAT" on page 7-15.

Any Terminator Allowed after END-EXEC

A SQL statement now can be terminated by a comma, a period or another COBOL statement. For more details, see "Sentence Terminator" on page 3-8.

Other New Features

New Name for Configuration File

The configuration file is now called *pcbcfg.cfg*, instead of *pccob.cfg*. See "Configuration Files" on page 7-6.

Support of Other Additional Datatypes

The computational usage datatype PACKED-DECIMAL is treated as COMP-3 datatype for ANSI compatibility.

The datatype SCALED DISPLAY (PIC 9(n) and PIC S9(n)) is supported. Digits are held in ASCII or EBCDIC format in radix 10, with one digit per byte of computer storage. If present, the sign is held in a separate byte (designated by the phrase SIGN SEPARATE). The position is trailing, the default, or may be specified using the SIGN TRAILING clause.

Support of Nested Programs

Pro*COBOL now allows nested programs with embedded SQL within a single source file. Nested programs cannot be recursive. All level 01 items which are marked as global in a containing program and are valid host variables at the containing program level are usable as valid host variables in any programs directly or indirectly contained by the containing program. For more details, see "Nested Programs" on page 3-22.

Support for REDEFINES and FILLER

The REDEFINES clause can be used to redefine group items. For more details, see "REDEFINES Clause" on page 3-7

The word FILLER is now allowed in host variable declarations. For more details, see "FILLER is Allowed" on page 3-9

New Precompiler Option PICX

The default datatype for PIC X variables is changed from VARCHAR2 to CHARF. A new precompiler option provides backwards compatibility:

PICX={VARCHAR2 | CHARF (default)}

This option is allowed only on the command line or in a configuration file. The new default behavior is consistent with the normal COBOL move behavior.

For more details, see "PICX" on page 7-32.

Optional CONVBUFSZ Clause in VAR Statement

This clause specifies an optional buffer used for conversion between character sets.

For more details, see "CONVBUFSZ Clause in VAR Statement" on page 4-23.

Improved Error Reporting

Errors are now associated with the proper line in any list file or in any terminal output. "Invalid host variable" errors state why the given COBOL variable is invalid for use in embedded SQL.

Changing Password When Connecting

The executable embedded SQL statement, CONNECT, has a new optional, final clause which allows you to change the password:

```
EXEC SQL CONNECT ... [ALTER AUTHORIZATION :new_password] END-EXEC.
```

See "Changing Passwords at Runtime" on page 3-55 and the connect statement entry on page F-12.

B

Operating System Dependencies

Some details of Pro*COBOL programming vary from one system to another. This appendix is a collection of all system-specific issues regarding Pro*COBOL. References are provided, where applicable, to other sources in your document set.

System-Specific References in this Manual

The references in this section appear in Chapter 3, "Writing a Pro*COBOL Program" using similar order and headings.

COBOL Versions

The Pro*COBOL Precompiler supports the standard implementation of COBOL for your operating system (usually COBOL-85 or COBOL-74). Some platforms may support both COBOL implementations. Check your Oracle system-specific documentation.

Host Variables

How you declare and name host variables depends on which COBOL compiler you use. Check your COBOL user's guide for details about declaring and naming host variables.

Declaring

Declare host variables according to COBOL rules, specifying a COBOL datatype supported by Oracle. Table 3–2, "Host Variable Declarations" shows the COBOL datatypes and pseudotypes you can specify. However, your COBOL implementation might not include all of them.

Naming

Host variable names must consist only of letters, digits, and hyphens, and must begin with a letter. They can be any length, but only the first 30 characters are significant. Your compiler might allow a different maximum length.

INCLUDE Statements

You can INCLUDE any file. When you precompile your Pro*COBOL program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

If your system uses file extensions but you do not specify one, the Pro*COBOL Precompiler assumes the default extension for source files (usually COB). The default extension is system-dependent. Check your Oracle system-specific documentation.

If your system uses directories, you can set a directory path for INCLUDEd files by specifying the precompiler option INCLUDE=*path*. You must use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current

directory. The syntax for specifying a directory path is system-specific. Check your Oracle system-specific documentation.

MAXLITERAL Default

With the MAXLITERAL precompiler option you can specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. The MAXLITERAL default value is 256, but you might have to specify a lower value.

For example, if your COBOL compiler cannot handle string literals longer than 132 characters, specify "MAXLITERAL=132." Check your COBOL compiler user's guide. For more information about the MAXLITERAL option, see Chapter 7, "Running the Pro*COBOL Precompiler"

PIC N Clause for Multi-byte NLS Characters

Some COBOL compilers may not support the use of the PIC N or PIC G clause for declaring multi-byte NLS character variables. Check your COBOL user's guide before writing source code that uses these clauses to declare multi-byte NLS character variables.

C

Oracle8 Reserved Words, Keywords, and Namespaces

This appendix lists words that have a special meaning to Oracle8. Each word plays a specific role in the context in which it appears. For example, in an INSERT statement, the reserved word INTO introduces the tables to which rows will be added. But, in a FETCH or SELECT statement, the reserved word INTO introduces the output host variables to which column values will be assigned.

Topics are:

- Oracle8 Reserved Words and Keywords
- Oracle8 Reserved Namespaces

Oracle8 Reserved Words and Keywords

|	&	:
,	-	=
>	[	<
(	.	+
]	)	!
/	*	^
@		ACCESS
ACCOUNT	ACTIVATE	ADD
ADMIN	ADVISE	AFTER
ALL	ALL_ROWS	ALLOCATE
ALTER	ANALYZE	AND
ANY	ARCHIVE	ARCHIVELOG
ARRAY	AS	ASC
AT	AUDIT	AUTHENTICATED
AUTHORIZATION	AUTOEXTEND	AUTOMATIC
BACKUP	BECOME	BEFORE
BEGIN	BETWEEN	BFILE
BITMAP	BLOB	BLOCK
BODY	BY	CACHE
CACHE_INSTANCES	CANCEL	CASCADE
CAST	CFILE	CHAINED
CHANGE	CHAR	CHAR_CS
CHARACTER	CHECK	CHECKPOINT

CHOOSE	CHUNK	CLEAR
CLOB	CLONE	CLOSE
CLOSE_CACHED_OPEN_CURSORS	CLUSTER	COALESCE
COLUMN	COLUMNS	COMMENT
COMMIT	COMMITTED	COMPATIBILITY
COMPILE	COMPLETE	COMPOSITE_LIMIT
COMPRESS	COMPUTE	CONNECT
CONNECT_TIME	CONSTRAINT	CONSTRAINTS
CONTENTS	CONTINUE	CONTROLFILE
CONVERT	COST	CPU_PER_CALL
CPU_PER_SESSION	CREATE	CURRENT
CURRENT_SCHEMA	CURRENT_USER	CURSOR
CYCLE	DANGLING	DATABASE
DATAFILE	DATAFILES	DATAOBJNO
DATE	DBA	DBHIGH
DBLOW	DBMAC	DEALLOCATE
DEBUG	DEC	DECIMAL
DECLARE	DEFAULT	DEFERRABLE
DEFERRED	DEGREE	DELETE
DEREF	DESC	DIRECTORY
DISABLE	DISCONNECT	DISMOUNT
DISTINCT	DISTRIBUTED	DML
DOUBLE	DROP	DUMP
EACH	ELSE	ENABLE
END	ENFORCE	ENTRY

ESCAPE	ESTIMATE	EVENTS
EXCEPT	EXCEPTIONS	EXCHANGE
EXCLUDING	EXCLUSIVE	EXECUTE
EXISTS	EXPIRE	EXPLAIN
EXTENT	EXTENTS	EXTERNALLY
FAILED_LOGIN_ATTEMPTS	FALSE	FAST
FILE	FIRST_ROWS	FLAGGER
FLOAT	FLOB	FLUSH
FOR	FORCE	FOREIGN
FREELIST	FREELISTS	FROM
FULL	FUNCTION	
GLOBAL	GLOBALLY	GLOBAL_NAME
GRANT	GROUP	GROUPS
HASH	HASHKEYS	HAVING
HEADER	HEAP	IDENTIFIED
IDGENERATORS	IDLE_TIME	IF
IMMEDIATE	IN	INCLUDING
INCREMENT	INDEX	INDEXED
INDEXES	INDICATOR	IND_PARTITION
INITIAL	INITIALLY	INITRANS
INSERT	INSTANCE	INSTANCES
INSTEAD	INT	INTEGER
INTERMEDIATE	INTERSECT	INTO
IS	ISOLATION	ISOLATION_LEVEL
KEEP	KEY	KILL

LABEL	LAYER	LESS
LEVEL	LIBRARY	LIKE
LIMIT	LINK	LIST
LOB	LOCAL	LOCK
LOCKED	LOG	LOGFILE
LOGGING	LOGICAL_READS_PER_CALL	LOGICAL_READS_PER_SESSION
LONG	MANAGE	MASTER
MAX	MAXARCHLOGS	MAXDATAFILES
MAXEXTENTS	MAXINSTANCES	MAXLOGFILES
MAXLOGHISTORY	MAXLOGMEMBERS	MAXSIZE
MAXTRANS	MAXVALUE	MIN
MEMBER	MINIMUM	MINEXTENTS
MINUS	MINVALUE	MLSLABEL
MLS_LABEL_FORMAT	MODE	MODIFY
MOUNT	MOVE	MTS_DISPATCHERS
MULTISET	NATIONAL	NCHAR
NCHAR_CS	NCLOB	NEEDED
NESTED	NETWORK	NEW
NEXT	NOARCHIVELOG	NOAUDIT
NOCACHE	NOCOMPRESS	NOCYCLE
NOFORCE	NOLOGGING	NOMAXVALUE
NOMINVALUE	NONE	NOORDER
NOOVERRIDE	NOPARALLEL	NORESETLOGS
NOREVERSE	NORMAL	NOSORT
NOT	NOTHING	NOWAIT

NULL	NUMBER	NUMERIC
NVARCHAR2	OBJECT	OBJNO
OBJNO_REUSE	OF	OFF
OFFLINE	OID	OIDINDEX
OLD	ON	ONLINE
ONLY	OPCODE	OPEN
OPTIMAL	OPTIMIZER_GOAL	OPTION
OR	ORDER	ORGANIZATION
OSLABEL	OVERFLOW	OWN
PACKAGE	PARALLEL	PARTITION
PASSWORD	PASSWORD_GRACE_TIME	PASSWORD_LIFE_TIME
PASSWORD_LOCK_TIME	PASSWORD_REUSE_MAX	PASSWORD_REUSE_TIME
PASSWORD_VERIFY_FUNCTION	PCTFREE	PCTINCREASE
PCTTHRESHOLD	PCTUSED	PCTVERSION
PERCENT	PERMANENT	PLAN
PLSQL_DEBUG	POST_TRANSACTION	PRECISION
PRESERVE	PRIMARY	PRIOR
PRIVATE	PRIVATE_SGA	PRIVILEGE
PRIVILEGES	PROCEDURE	PROFILE
PUBLIC	PURGE	QUEUE
QUOTA	RANGE	RAW
RBA	READ	READUP
REAL	REBUILD	RECOVER
RECOVERABLE	RECOVERY	REF
REFERENCES	REFERENCING	REFRESH

RENAME	REPLACE	RESET
RESETLOGS	RESIZE	RESOURCE
RESTRICTED	RETURN	RETURNING
REUSE	REVERSE	REVOKE
ROLE	ROLES	ROLLBACK
ROW	ROWID	ROWNUM
ROWS	RULE	SAMPLE
SAVEPOINT	SB4	SCAN_INSTANCES
SCHEMA	SCN	SCOPE
SD_ALL	SD_INHIBIT	SD_SHOW
SEGMENT	SEG_BLOCK	SEG_FILE
SELECT	SEQUENCE	SERIALIZABLE
SESSION	SESSION_CACHED_CURSORS	SESSIONS_PER_USER
SET	SHARE	SHARED
SHARED_POOL	SHRINK	SIZE
SKIP	SKIP_UNUSABLE_INDEXES	SMALLINT
SNAPSHOT	SOME	SORT
SPECIFICATION	SPLIT	SQL_TRACE
STANDBY	START	STATEMENT_ID
STATISTICS	STOP	STORAGE
STORE	STRUCTURE	SUCCESSFUL
SWITCH	SYS_OP_ENFORCE_NOT_NULL$	SYS_OP_NTCIMG$
SYNONYM	SYSDATE	SYSDBA
SYSOPER	SYSTEM	TABLE

TABLES	TABLESPACE	TABLESPACE_NO
TABNO	TEMPORARY	THAN
THE	THEN	THREAD
TIMESTAMP	TIME	TO
TOPLEVEL	TRACE	TRACING
TRANSACTION	TRANSITIONAL	TRIGGER
TRIGGERS	TRUE	TRUNCATE
TX	TYPE	UB2
UBA	UID	UNARCHIVED
UNDO	UNION	UNIQUE
UNLIMITED	UNLOCK	UNRECOVERABLE
UNTIL	UNUSABLE	UNUSED
UPDATABLE	UPDATE	USAGE
USE	USER	USING
VALIDATE	VALIDATION	VALUE
VALUES	VARCHAR	VARCHAR2
VARYING	VIEW	WHEN
WHENEVER	WHERE	WITH
WITHOUT	WORK	WRITE
WRITEDOWN	WRITEUP	XID

Oracle8 Reserved Namespaces

Table C–1 contains a list of namespaces that are reserved by Oracle8. The initial characters of function names in Oracle8 libraries are restricted to the character strings in this list. Because of potential name conflicts, use function names that do not begin with these characters.

For example, the SQL*Net Transparent Network Service functions all begin with the characters "NS," so you need to avoid writing functions whose names begin with "NS."

Table C–1   Oracle8 Reserved Namespaces

Namespace	Library
O	OCI functions
S	function names from SQLLIB and system-dependent libraries
XA	external functions for XA applications only
GEN KP L NA NC ND NL NM NR NS NT NZ TTC UPI	Internal functions

D

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. Topics are:

- What Causes Poor Performance?
- How Can Performance be Improved?
- Using Host Tables
- Using Embedded PL/SQL
- Optimizing SQL Statements
- Using Indexes
- Taking Advantage of Row-Level Locking
- Eliminating Unnecessary Parsing

What Causes Poor Performance?

One cause of poor performance is high Oracle communication overhead. Oracle8 must process SQL statements one at a time. Thus, each statement results in another call to Oracle8 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)
END-EXEC.

EXEC SQL SELECT DNAME, DEPTNO
    FROM DEPT
    WHERE EXISTS
    (SELECT DEPTNO FROM EMP WHERE DEPT.DEPTNO = EMP.DEPTNO)
END-EXEC.
```

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, Oracle8 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 Oracle8 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 Tables

Host tables 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 tables 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) END-EXEC.
```

If *SALARY* is a simple host variable, Oracle8 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 table of size 300, Oracle8 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* table.

For more information, see Chapter 10, "Using Host Tables"

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 Oracle8. This can drastically reduce communication between your application and Oracle8.

Also, you can use PL/SQL subprograms to reduce calls from your application to Oracle8. 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.

Unlike anonymous blocks, PL/SQL subprograms can be compiled separately and stored in an Oracle8 database. When called, they are passed to the PL/SQL engine immediately. Moreover, only one copy of a subprogram need be loaded into memory for execution by multiple users.

Figure D–1   PL/SQL Boosts Performance
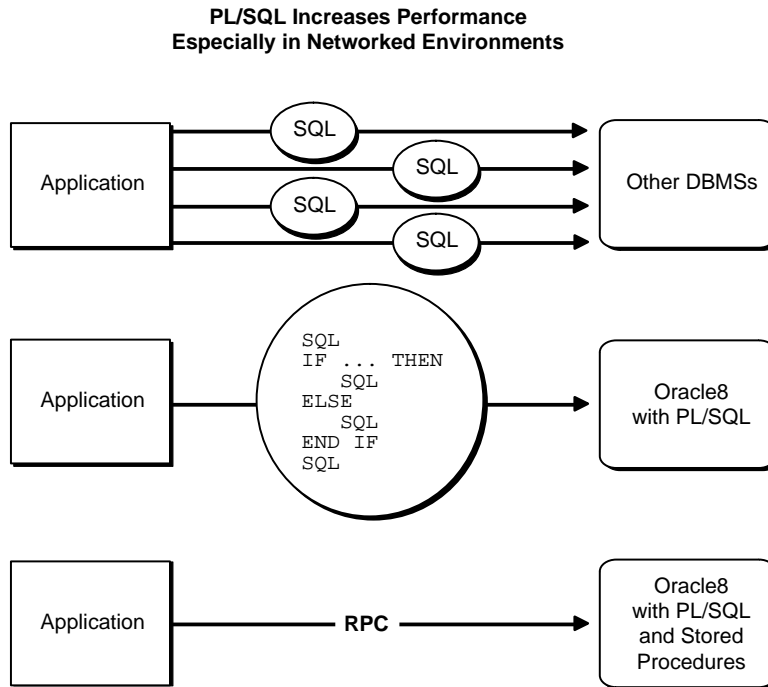
PL/SQL can also cooperate with Oracle8 application development tools such as Oracle Forms and Oracle Reports. By adding procedural processing power to these tools, PL/SQL boosts performance. Using PL/SQL, a tool can do any computation quickly and efficiently without calling on Oracle8. This saves time and reduces network traffic. For more information, see Chapter 6, "Using Embedded PL/SQL" and the *PL/SQL User's Guide and Reference.*

Optimizing SQL Statements

For every SQL statement, the Oracle8 optimizer generates an *execution plan*, which is a series of steps that Oracle8 takes to execute the statement. These steps are determined by rules given in the *Oracle8 Application Developer's Guide*. Following these rules will help you write optimal SQL statements.

Optimizer Hints

For every SQL statement, the Oracle8 optimizer generates an *execution plan*, which is a series of steps that Oracle8 takes to execute the statement. In some cases, you can suggest to Oracle8 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

Giving Hints

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. You can choose rule-based or cost-based optimization. With cost-based optimization, hints help maximize throughput or response time. In the following example, the ALL_ROWS hint helps maximize query throughput:

```
EXEC SQL SELECT /*+ ALL_ROWS (cost-based) */ EMPNO, ENAME, SAL
    INTO :EMP-NUMBER, :EMP-NAME, :SALARY
    FROM EMP
    WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

The plus sign (+), which must immediately follow the Comment opener, indicates that the Comment contains one or more hints. Notice that the Comment can contain remarks as well as hints.

For more information about optimizer hints, see the *Oracle8 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 trace facility generates statistics for every SQL statement executed by Oracle8. 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. You can use the execution plan to identify inefficient SQL statements.

For instructions on using these tools and analyzing their output, see the *Oracle8 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 *Oracle8 SQL Reference*.

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 a *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 *Oracle8 Application Developer's Guide*.

Taking Advantage of Row-Level Locking

By default, Oracle8 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 "Using the LOCK TABLE Statement" on "Using the LOCK TABLE Statement" on page 8-12.

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.

Note: You can use the ORACA to get cursor cache statistics. See "Using the Oracle Communications Area" on page 9-35.

Handling Explicit Cursors

Recall that there are two types of cursors: implicit and explicit (see "Private SQL Areas, Cursors, and Active Sets" on page 2-8). Oracle8 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 tables). You use the DECLARE CURSOR statement to declare an explicit cursor. How you handle the opening and closing of explicit cursors affects performance.

If you need to reevaluate the active set, simply reopen the cursor. The OPEN statement 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, the precompiler lets you reopen an already open cursor. However, this is an Oracle8 extension to the ANSI/ISO embedded SQL standard. So, when MODE=ANSI, 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 three ways to control an explicitly declared cursor:

- use the DECLARE, OPEN, and CLOSE statements

- use the PREPARE, DECLARE, OPEN, and CLOSE statements

- COMMIT closes the cursor when MODE=ANSI

With the first way, beware of unnecessary parsing. The OPEN statement 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, re-open 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 (dynamic SQL Methods 3 and 4), the PREPARE statement does the parsing, and the parsed statement is available until a CLOSE statement is executed. Your program should prepare the SQL statement and declare the cursor, re-open the cursor every time the value of a host variable changes, re-prepare the SQL statement and re-open the cursor if the SQL statement changes, and close the cursor only when the SQL statement is no longer needed.

When possible, avoid placing OPEN and CLOSE statements in a loop; this is a potential cause of unnecessary re-parsing of the SQL statement. In the next example, both the OPEN and CLOSE statements are inside the outer loop. When MODE=ANSI, the CLOSE statement must be positioned as shown, because ANSI requires a cursor to be closed before being re-opened.

```
     EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
         SELECT ENAME, SAL FROM EMP
         WHERE SAL >  :SALARY
         AND SAL <= :SALARY + 1000
     END-EXEC.
     MOVE 0 TO SALARY.
 TOP.
     EXEC SQL OPEN EMP-CURSOR END-EXEC.
 LOOP.
     EXEC SQL FETCH EMP-CURSOR INTO ....

     ...
         IF SQLCODE = 0
             GO TO LOOP
          ELSE
             ADD 1000 TO SALARY
          END-IF.
     EXEC SQL CLOSE EMP-CURSOR END-EXEC.
     IF SALARY < 5000
         GO TO TOP.
```

With MODE=ORACLE, however, a CLOSE statement can execute without the cursor being OPENed. By placing the CLOSE statement outside the outer loop, you can avoid possible re-parsing at each iteration of the OPEN statement.

```
 TOP.
     EXEC SQL OPEN EMP-CURSOR END-EXEC.
```

```
LOOP.
    EXEC SQL FETCH EMP-CURSOR INTO ....
    ...
        IF SQLCODE = 0
             GO TO LOOP
        ELSE
             ADD 1000 TO SALARY
        END-IF.
    IF SALARY < 5000
        GO TO TOP.
    EXEC SQL CLOSE EMP-CURSOR END-EXEC.
```

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 MAXOPENCUR-SORS options give you some control over how Oracle8 manages the parsing and re-parsing 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 Oracle8, 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 D–2 represents the cursor cache after your program has done an insert and a delete.

Figure D–2   Cursors Linked via the Cursor Cache
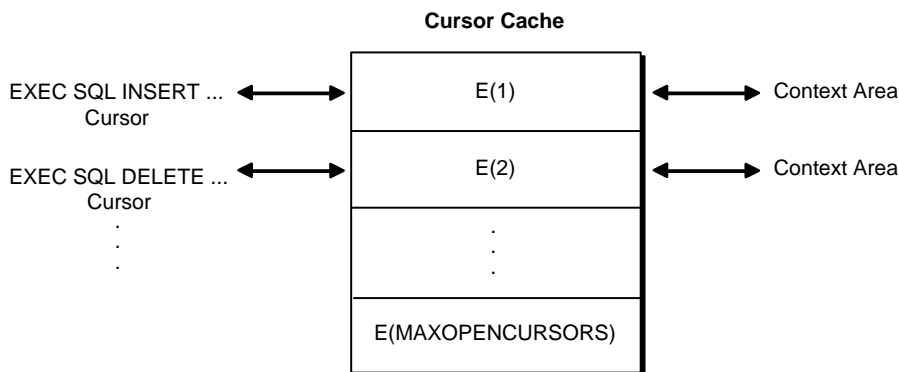
Resource Use

The maximum number of open cursors per user session is set by the Oracle8 initial-ization parameter OPEN_CURSORS.

MAXOPENCURSORS specifies the *initial* size of the cursor cache. If a new cursor is needed and there are no free cache entries, Oracle8 tries to reuse an entry. Its success depends on the values of HOLD_CURSOR and RELEASE_CURSOR and, for explicit cur-sors, on the status of the cursor itself.

If the value of MAXOPENCURSORS is less than the number of cache entries actu-ally needed, Oracle8 uses the first cache entry marked as reusable. For example, suppose the cache entry $E(1)$ for an INSERT statement is marked as reusable, and the number of cache entries already equals MAXOPENCURSORS. If the program executes a new statement, cache entry $E(1)$ and its private SQL area might be reassigned to the new statement. To re-execute the INSERT statement, Oracle8 would have to re-parse it and reas-sign another cache entry.

Oracle8 allocates an additional cache entry if it cannot find one to reuse. For exam-ple, if MAXOPENCURSORS=8 and all eight entries are active, a ninth is created. If necessary, Oracle8 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 de-allocations 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 Oracle8 executes the SQL statement and the cursor is closed, the precompiler 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 Oracle8 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, Oracle8 must still reallocate memory for a private SQL area and re-parse the SQL statement before executing it because RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES.

Nonetheless, when RELEASE_CURSOR=YES, the re-parse might not require extra processing because Oracle8 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 Oracle8 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 re-parsing.

When HOLD_CURSOR=YES and RELEASE_CURSOR=NO (the default), the link between the cache entry and the private SQL area is maintained after Oracle8 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.

Attention:

Using the defaults, HOLD_CURSOR=YES and RELEASE_CURSOR=NO, after executing a SQL statement with an earlier Oracle version, its parsed representation remains available. With Oracle8, under similar conditions, 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 re-parsed.

Parameter Interactions

Table D–1 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.

Table D–1   HOLD_CURSOR and RELEASE _CURSOR Interactions

HOLD_CURSOR	RELEASE_CURSOR	Links are ...
NO	NO	marked as reusable
YES	NO	maintained
NO	YES	removed immediately
YES	YES	removed immediately

E

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.

Topics are:

- What Is Syntactic and Semantic Checking?
- Controlling the Type and Extent of Checking
- Specifying SQLCHECK=SEMANTICS

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 so on are placed correctly in your SQL statement. For example, the following embedded SQL statements contain syntax errors:

```
* --  misspelled keyword WHERE
     EXEC SQL DELETE FROM EMP WERE DEPTNO = 20 END-EXEC.
* --  missing parentheses around column names COMM and SAL
     EXEC SQL
          INSERT INTO EMP COMM, SAL VALUES (NULL, 1500)
     END-EXEC.
```

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:

```
* --  nonexistent table, EMPP
     EXEC SQL DELETE FROM EMPP WHERE DEPTNO = 20 END-EXEC.
* --  undeclared host variable, EMP-NAME
     EXEC SQL SELECT * FROM EMP WHERE ENAME = :EMP-NAME END-EXEC.
```

The rules of SQL syntax and semantics are defined in the *Oracle8 SQL Reference*.

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 data manipulation statements and PL/SQL blocks. However, SQLCHECK cannot check dynamic SQL statements because they are not defined fully until run time.

You can specify the following values for SQLCHECK:

- SEMANTICS | FULL
- SYNTAX | LIMITED | NONE

The values SEMANTICS and FULL are equivalent, as are the values SYNTAX and LIMITED. The default value is SYNTAX.

Specifying SQLCHECK=SEMANTICS

When SQLCHECK=SEMANTICS, the precompiler checks the syntax and semantics of

- data manipulation statements such as INSERT and UPDATE
- PL/SQL blocks

However, the precompiler checks only the syntax of remote data manipulation statements (those using the AT *db_name* clause).

The precompiler gets the information for a semantic check from embedded DECLARE TABLE statements or, if you specify the option USERID, by connecting to Oracle8 and accessing the data dictionary. You need not connect to Oracle8 if every table referenced in a data manipulation statement or PL/SQL block is defined in a DECLARE TABLE statement.

If you connect to Oracle8 but some 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.

When checking data manipulation statements, the precompiler uses the Oracle8 set of syntax rules found in the *Oracle8 SQL Reference* but uses a stricter set of semantic rules. As a result, existing applications written for earlier versions of Oracle might not precompile successfully when SQLCHECK=SEMANTICS.

Specify SQLCHECK=SEMANTICS when precompiling new programs. If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK=SEMANTICS.

Enabling a Semantic Check

When SQLCHECK=SEMANTICS, the precompiler 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 precompiler can connect to an Oracle8 database that maintains definitions of tables and views referenced in your host program. After connecting to Oracle8, the precompiler 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.

To connect to Oracle8, specify the option USERID on the command line, using the syntax

```
USERID=username/password
```

where *username* and *password* comprise a valid Oracle8 userid. If you omit the password, you are prompted for it. If, instead of a username and password, you specify

```
USERID=/
```

the precompiler tries to connect to Oracle8 automatically with the userid

```
<prefix><username>
```

where *prefix* is the value of the Oracle8 initialization parameter OS_AUTHENT_PREFIX (the default value is OPS$) and *username* is your operating system user or task name.

If you try connecting to Oracle8 but cannot (for example, if the database is unavailable), the precompiler stops processing and issues an error message. If you omit the option USERID, the precompiler must get needed information from embedded DECLARE TABLE statements.

Using DECLARE TABLE

The precompiler can do a semantic check without connecting to Oracle8. 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], ...)
END-EXEC.
```

where *expr* is any expression that can be used as a default column value in the CREATE TABLE statement. *col_datatype* is an Oracle column declaration. Only integers can be used, not expressions. See "DECLARE TABLE (Oracle Embedded SQL Directive)" on page F-20.

If you use DECLARE TABLE to define a database table that already exists, the pre-compiler uses your definition, ignoring the one in the data dictionary.

F

Embedded SQL Commands and Precompiler Directives

This appendix contains descriptions of both SQL92 embedded SQL commands and directives as well as the Oracle8 embedded SQL extensions. These commands and directives are prefaced in your source code with the keywords, EXEC SQL.

Note: Only statements which differ in syntax from non-embedded SQL are described in this appendix. For details of the non-embedded SQL statements, see the *Oracle8 SQL Reference*.

This appendix contains the following sections:

- Summary of Precompiler Directives and Embedded SQL Commands

- About The Command Descriptions

- How to Read Syntax Diagrams

- ALLOCATE (Executable Embedded SQL Extension)

- CLOSE (Executable Embedded SQL)

- COMMIT (Executable Embedded SQL)

- CONNECT (Executable Embedded SQL Extension)

- DECLARE CURSOR (Embedded SQL Directive)

- DECLARE DATABASE (Oracle Embedded SQL Directive)

- DECLARE STATEMENT (Embedded SQL Directive)

- DECLARE TABLE (Oracle Embedded SQL Directive)

- DELETE (Executable Embedded SQL)

- DESCRIBE (Executable Embedded SQL)

- DESCRIBE (Executable Embedded SQL)
- EXECUTE ... END-EXEC (Executable Embedded SQL Extension)
- EXECUTE (Executable Embedded SQL)
- EXECUTE IMMEDIATE (Executable Embedded SQL)
- FETCH (Executable Embedded SQL)
- FETCH (Executable Embedded SQL)
- INSERT (Executable Embedded SQL)
- OPEN (Executable Embedded SQL)
- PREPARE (Executable Embedded SQL)
- ROLLBACK (Executable Embedded SQL)
- SAVEPOINT (Executable Embedded SQL)
- SELECT (Executable Embedded SQL)
- UPDATE (Executable Embedded SQL)
- VAR (Oracle Embedded SQL Directive)
- WHENEVER (Embedded SQL Directive)

Summary of Precompiler Directives and Embedded SQL Commands

Embedded SQL commands place DDL, DML, and Transaction Control statements within a procedural language program. Embedded SQL is supported by the Oracle Precompilers. Table F–1 provides a functional summary of the embedded SQL commands and directives.

The *type* column in Table F–1 is displayed in the format, *source/type*, where:

source	is either SQL92 standard SQL (S) or an Oracle extension (O)
type	is either an executable (E) statement or a directive (D)

Table F–1   Precompiler Directives and Embedded SQL Commands and Clauses

EXEC SQL Statement	Type	Purpose
ALLOCATE	O/E	To allocate memory for a cursor variable.
CLOSE	S/E	To disable a cursor, releasing the resources it holds.
COMMIT	S/E	To end the current transaction, making all database change permanent (optionally frees resources and disconnects from the database)
CONNECT	O/E	To log on to an Oracle8 instance.
DECLARE CURSOR	S/D	To declare a cursor, associating it with a query.
DECLARE DATABASE	O/D	To declare an identifier for a non-default database to be accessed in subsequent embedded SQL statements.
DECLARE STATEMENT	S/D	To assign a SQL variable name to a SQL statement.
DECLARE TABLE	O/D	To declare the table structure for semantic checking of embedded SQL statements by the Oracle Precompiler.
DELETE	S/E	To remove rows from a table or from a view's base table.
DESCRIBE	S/E	To initialize a descriptor, a structure holding host variable descriptions.
EXECUTE...END-EXEC	O/E	To execute an anonymous PL/SQL block.
EXECUTE	S/E	To execute a prepared dynamic SQL statement.
EXECUTE IMMEDIATE	S/E	To prepare and execute a SQL statement with no host variables.
FETCH	S/E	To retrieve rows selected by a query.

Table F–1   Precompiler Directives and Embedded SQL Commands and Clauses

EXEC SQL Statement	Type	Purpose
INSERT	S/E	To add rows to a table or to a view's base table.
OPEN	S/E	To execute the query associated with a cursor.
PREPARE	S/E	To parse a dynamic SQL statement.
ROLLBACK	S/E	To end the current transaction, discard all changes in the current transaction, and release all locks (optionally release resources and disconnect from the database).
SAVEPOINT	S/E	To identify a point in a transaction to which you can later roll back.
SELECT	S/E	To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.
UPDATE	S/E	To change existing values in a table or in a view's base table.
VAR	O/D	To override the default datatype and assign a specific Oracle8 external datatype to a host variable.
WHENEVER	S/D	To specify handling for error and warning conditions.

About The Command Descriptions

The directives, commands, and clauses appear alphabetically. The description of each contains the following sections:

Purpose   describes the basic uses of the command.

Prerequisites   lists privileges you must have and steps that you must take before using the command. Unless otherwise noted, most commands also require that the database be open by your instance.

Syntax   shows the keywords and parameters of the command.

Keywords and   describes the purpose of each keyword and parameter.
Parameters

Usage Notes   discusses how and when to use the command.

Examples   shows example statements of the command.

Related Topics   lists related commands, clauses, and sections of this manual.

How to Read Syntax Diagrams

Syntax diagrams are used to illustrate embedded SQL syntax. They are drawings that depict valid syntax.

Trace each diagram from left to right, in the direction shown by the arrows.

Commands and other keywords appear in UPPER CASE inside rectangles. Type them exactly as shown in the rectangles. Parameters appear in lower case inside ovals. Variables are used for the parameters. Operators, delimiters, and terminators appear inside circles.

If the syntax 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:
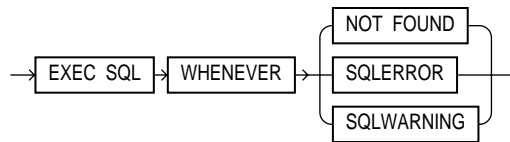
According to the diagram, all of the following statements are valid:

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

Statement Terminator

In all Pro*COBOL EXEC SQL diagrams, each statement is understood to end with the token *END-EXEC.*

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 is a required parameter:
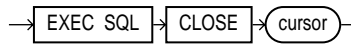
If there is a cursor named *EMPCURSOR*, then, according to the diagram, the following statement is valid:

```
EXEC SQL CLOSE EMPCURSOR END-EXEC.
```

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:
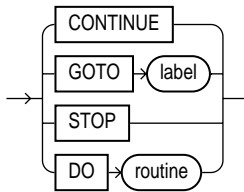
Optional Keywords and Parameters

If keywords and parameters appear in a vertical list above the main path, they are optional. In the following example, instead of traveling down a vertical line, you can continue along the main path:
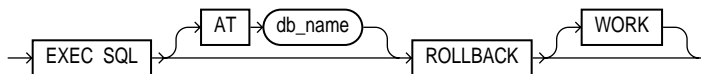
If there is a database named *oracle2*, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL ROLLBACK END-EXEC.
EXEC SQL ROLLBACK WORK END-EXEC.
EXEC SQL AT ORACLE2 ROLLBACK END-EXEC.
```

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.
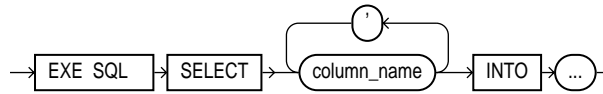
If DEBIT, CREDIT, and BALANCE are column names, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

Multi-part Diagrams

Read a multi-part diagram as if all the main paths were joined end-to-end. The following example is a two-part diagram:
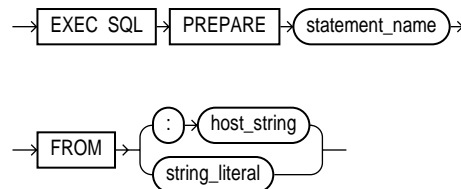
According to the diagram, the following statement is valid:

```
EXEC SQL PREPARE sqlstatement FROM :SQL-STRING END-EXEC.
```

Database Objects

The names of Oracle identifiers, 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.

ALLOCATE (Executable Embedded SQL Extension)

Purpose
To allocate a cursor variable to be referenced in a PL/SQL block.

Prerequistes
A cursor variable (see Chapter 6, "Using Embedded PL/SQL") of type SQL-CURSOR must be declared before allocating memory for the cursor variable.

Syntax

Keywords and Parameters

cursor_variable   is the cursor variable to be allocated.

Usage Notes
Whereas a cursor is static, a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query.

For more information on this command, see *PL/SQL User's Guide and Reference* and *Oracle8 SQL Reference.*

Example
This partial example illustrates the use of the ALLOCATE command in a Pro*COBOL embedded SQL program:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01  EMP-CUR       SQL-CURSOR.
```

```
01  EMP-REC.
    ...
EXEC SQL END DECLARE SECTION END-EXEC.
EXEC SQL ALLOCATE :EMP-CUR END-EXEC.
...
```

Related Topics

CLOSE (Executable Embedded SQL)

EXECUTE (Executable Embedded SQL)

FETCH (Executable Embedded SQL)

CLOSE (Executable Embedded SQL)

Purpose

To disable a cursor, freeing the resources acquired by opening the cursor, and releasing parse locks.

Prerequisites

The cursor or cursor variable must be open and MODE=ANSI.

Syntax
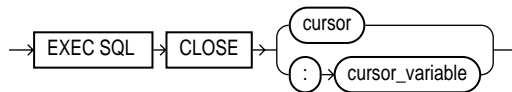
Keywords and Parameters

cursor	is a cursor to be closed.
cusor_variable	is a cursor variable to be closed.

Usage Notes

Rows cannot be fetched from a closed cursor. A cursor need not be closed to be reopened. The HOLD_CURSOR and RELEASE_CURSOR precompiler options alter the effect of the CLOSE command. For information on these options, see Chapter 7.

Example

This example illustrates the use of the CLOSE command:

```
EXEC SQL CLOSE EMP-CUR END-EXEC.
```

Related Topics

PREPARE (Executable Embedded SQL)

DECLARE CURSOR (Embedded SQL Directive)

OPEN (Executable Embedded SQL)

COMMIT (Executable Embedded SQL)

Purpose

To end your current transaction, making permanent all its changes to the database and optionally freeing all resources and disconnecting from the Oracle8 Server.

Prerequisites

To commit your current transaction, no privileges are necessary.

To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

If you are using Trusted Oracle in DBMS MAC mode, you can only commit an in-doubt transaction if your DBMS label matches the label the transaction's label and the creation label of the user who originally committed the transaction or if you satisfy one of the following criteria:

- If the transaction's label or the user's creation label is higher than your DBMS label, you must have READUP and WRITEUP system privileges.

- If the transaction's label or the user's creation label is lower than your DBMS label, you must have WRITEDOWN system privilege.

- If the transaction's label or the user's creation label is not comparable with your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

Syntax
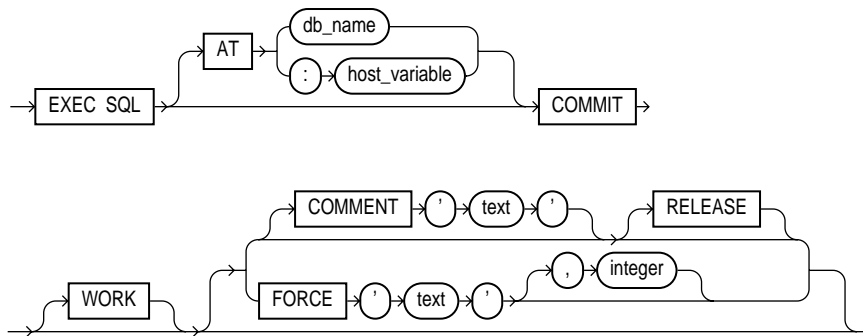
Keyword and Parameters

AT	identifies the database to which the COMMIT statement is issued. The database can be identified by either:

	dbname	is a database identifier declared in a previous DECLARE DATABASE statement.
	:*host_variable*	is a host variable whose value is a previously declared *db_name*.

If you omit this clause, Oracle8 issues the statement to your default database.

WORK	is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.
COMMENT	specifies a comment to be associated with the current transaction. The '*text*' is a quoted literal of up to 50 characters that Oracle8 stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.
RELEASE	frees all resources and disconnects the application from the Oracle8 Server.
FORCE	manually commits an in-doubt distributed transaction. The transaction is identified by the '*text*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the optional *integer* to explicitly assign the transaction a system change number (SCN). If you omit the *integer*, the transaction is committed using the current SCN.

Usage Notes

Always explicitly commit or rollback the last transaction in your program by using the COMMIT or ROLLBACK command and the RELEASE option. Oracle8 automatically rolls back changes if the program terminates abnormally.

The COMMIT command has no effect on host variables or on the flow of control in the program. For more information on this command, see "Using the COMMIT Statement" on page 8-4.

Example

This example illustrates the use of the embedded SQL COMMIT command:

```
EXEC SQL AT SALESDB COMMIT RELEASE END-EXEC.
```

Related Topics

ROLLBACK (Executable Embedded SQL)

SAVEPOINT (Executable Embedded SQL)

CONNECT (Executable Embedded SQL Extension)

Purpose

To logon to an Oracle8 database.

Prerequisites

You must have CREATE SESSION system privilege in the specified database.

If you are using Trusted Oracle in DBMS MAC mode, your operating system label must dominate both your creation label and the label at which you were granted CREATE SESSION system privilege. Your operating system label must also fall between the operating system equivalents of DBHIGH and DBLOW, inclusive.

If you are using Trusted Oracle in OS MAC mode, your operating system label must match the label of the database to which you are connecting.

Syntax

Keyword and Parameters

:user *:password*	specifies your username and password separately.
:user_password	is a single host variable containing the Oracle8 username and password seperated by a slash (/).
	To allow Oracle8 to verify your connection through your operating system, specify "/" as the *:user_password* value. To allow Oracle8 to verify your connection through your operating system, specify "/" as the *:user_password* value.
AT	identifies the database to which the connection is made. The database can be identified by either:
	db_name is a database identifer declared in a previous DECLARE DATABASE statement.
	:host_variable is a host variable whose value is a previously declared *db_name*.
USING	specifies the SQL*Net database specification string used to connect to a non-default database. If you omit this clause, you are connected to your default database.

| ALTER AUTHORIZATION | Change password to the following string. |
| :new_password | New password string. |

Usage Notes

A program can have multiple connections, but can only connect once

to your default database. For more information on this command, see "Embedding OCI (Oracle Call Interface) Calls" on page 4-34.

Example

The following example illustrate the use of CONNECT:

```
EXEC SQL CONNECT :USERNAME
        IDENTIFIED BY :PASSWORD
END-EXEC.
```

You can also use this statement in which the value of :*userid* is the value of :*username* and :*password* separated by a "/" such as 'SCOTT/TIGER':

```
EXEC SQL CONNECT :USERID END-EXEC.
```

Related Topics

COMMIT (Executable Embedded SQL)

DECLARE DATABASE (Oracle Embedded SQL Directive)

ROLLBACK (Executable Embedded SQL)

DECLARE CURSOR (Embedded SQL Directive)

Purpose

To declare a cursor, giving it a name and associating it with a SQL statement or a PL/SQL block.

Prerequisites

If you associate the cursor with an identifier for a SQL statement or PL/SQL block, you must have declared this identifier in a previous DECLARE STATEMENT statement.

Syntax

Keywords and Parameters

AT	i dentifies the database on which the cursor is declared. The database can be identified by either:
	db_name is a database identifier declared in a previous DECLARE DATABASE statement.
	:host_variable is a host variable whose value is a previously declared *db_name*.
	If you omit this clause, Oracle8 declares the cursor on your default database.
cursor	is the name of the cursor to be declared.
WITH HOLD	cursor remains open after a COMMIT or a ROLLBACK. The cursor must not be declared for UPDATE.
SELECT *command*	is a SELECT statement to be associated with the cursor. The following statement cannot contain an INTO clause.
statement_name *block_name*	identifies a SQL statement or PL/SQL block to be associated with the cursor. The *statement_name* or *block_name* must be previously declared in a DECLARE STATEMENT statement.

Usage Notes

You must declare a cursor before referencing it in other embedded SQL statements. The scope of a cursor declaration is global within its precompilation unit and the name of each cursor must be unique in its scope. You cannot declare two cursors with the same name in a single precompilation unit.

You can reference the cursor in the WHERE clause of an UPDATE or DELETE statement using the CURRENT OF syntax, provided that the cursor has been opened with an OPEN statement and positioned on a row with a FETCH statement. For more information on this command, see .

Example

This example illustrates the use of a DECLARE CURSOR statement:

```
EXEC SQL DECLARE EMPCURSOR CURSOR
    FOR SELECT ENAME, EMPNO, JOB, SAL
    FROM EMP
    WHERE DEPTNO = :DEPTNO
    FOR UPDATE OF SAL
END-EXEC.
```

Related Topics

CLOSE (Executable Embedded SQL)

DECLARE DATABASE (Oracle Embedded SQL Directive)

DECLARE STATEMENT (Embedded SQL Directive)

DELETE (Executable Embedded SQL)

FETCH (Executable Embedded SQL)

OPEN (Executable Embedded SQL)

PREPARE (Executable Embedded SQL)

SELECT (Executable Embedded SQL)

UPDATE (Executable Embedded SQL)

DECLARE DATABASE (Oracle Embedded SQL Directive)

Purpose

To declare an identifier for a non-default database to be accessed in subsequent embedded SQL statements.

Prerequisites

You must have access to a username on the non-default database.

Syntax


```
→ EXEC SQL → DECLARE → ( db_name ) → DATABASE ⊢
```
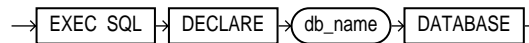
Keywords and Parameters

db_name   is the identifier established for the non-default database.

Usage Notes

You declare a *db_name* for a non-default database so that other embedded SQL statements can refer to that database using the AT clause. Before issuing a CONNECT statement with an AT clause, you must declare a *db_name* for the non-default database with a DECLARE DATABASE statement.

For more information on this command, see "Explicit Logons" on page 3-47.

Example

This example illustrates the use of a DECLARE DATABASE directive:

```
EXEC SQL DECLARE ORACLE3 DATABASE END-EXEC.
```

Related Topics

COMMIT (Executable Embedded SQL)

CONNECT (Executable Embedded SQL Extension)

DECLARE CURSOR (Embedded SQL Directive)

DECLARE STATEMENT (Embedded SQL Directive)

DELETE (Executable Embedded SQL)

EXECUTE (Executable Embedded SQL)

EXECUTE IMMEDIATE (Executable Embedded SQL)

INSERT (Executable Embedded SQL)

SELECT (Executable Embedded SQL)

UPDATE (Executable Embedded SQL)

DECLARE STATEMENT (Embedded SQL Directive)

Purpose
To declare an identifier for a SQL statement or PL/SQL block to be used in other embedded SQL statements.

Prerequisites
None.
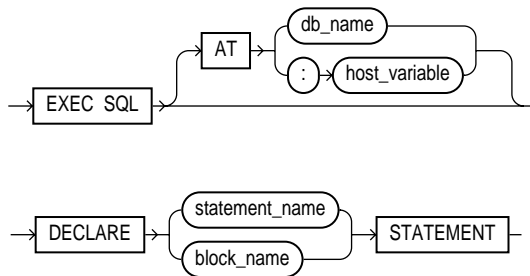
Syntax

Keywords and Parameters

AT	identifies the database on which the SQL statement or PL/SQL block is declared. The database can be identified by either:
	db_name is a database identifier declared in a previous DECLARE DATABASE statement.

	:host_variable	is a host variable whose value is a previously declared *db_name*.
		If you omit this clause, Oracle8 declares the SQL statement or PL/SQL block on your default database.
statement_name		is the declared identifier for the statement.
block_name		PL/SQL block

Usage Notes

You must declare an identifier for a SQL statement or PL/SQL block with a DECLARE STATEMENT statement only if a DECLARE CURSOR statement referencing the identifier appears physically (not logically) in the embedded SQL program before the PREPARE statement that parses the statement or block and associates it with its identifier.

The scope of a statement declaration is global within its precompilation unit, like a cursor declaration. For more information on this command, see "DECLARE" on page 11-20.

Example I

This example illustrates the use of the DECLARE STATEMENT statement:

```
EXEC SQL AT REMOTEDB
    DECLARE MYSTATEMENT STATEMENT
END-EXEC.
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING
END-EXEC.
EXEC SQL EXECUTE MYSTATEMENT END-EXEC.
```

Example II

In this example from a Pro*COBOL embedded SQL program, the DECLARE STATEMENT statement is required because the DECLARE CURSOR statement precedes the PREPARE statement:

```
EXEC SQL DECLARE MYSTATEMENT STATEMENT END-EXEC.
EXEC SQL DECLARE EMPCURSOR CURSOR FOR MYSTATEMENT END-EXEC.
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.
...
```

Related Topics

CLOSE (Executable Embedded SQL)

DECLARE DATABASE (Oracle Embedded SQL Directive)

FETCH (Executable Embedded SQL)

PREPARE (Executable Embedded SQL)

OPEN (Executable Embedded SQL)

DECLARE TABLE (Oracle Embedded SQL Directive)

Purpose

To define the structure of a table or view, including each column's datatype, default value, and NULL or NOT NULL specification for semantic checking by the precompiler when option SQLCHECK=SEMANTICS (or FULL).

Prerequisites

None.

Syntax
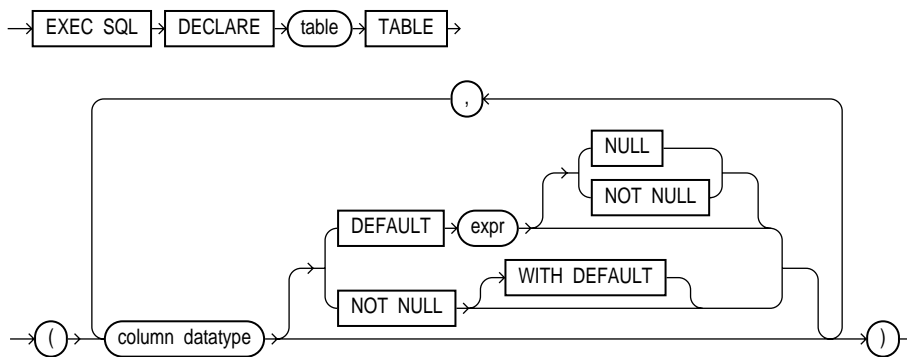
Keywords and Parameters

table	is the name of the declared table.
column	is a column of the *table*.
datatype	is the datatype of a *column*. For information on Oracle8 datatypes, see Chapter 4.
DEFAULT	specifies the default value of a *column*.
NULL	specifies that a *column* can contain nulls.
NOT NULL	specifies that a *column* cannot contain nulls.
WITH DEFAULT	is supported for compatibility with the IBM DB2 database.

Usage Notes

Datatypes can only use integers (not expressions) for length, precision, scale. For more information on using this command, see "Specifying SQLCHECK=SEMAN-TICS" on page E-3.

Example

The following statement declares the PARTS table with the PARTNO, BIN, and QTY columns:

```
EXEC SQL DECLARE PARTS TABLE
    (PARTNO  NUMBER  NOT NULL,
    BIN     NUMBER,
    QTY     NUMBER)
END-EXEC.
```

Related Topics

None.

DELETE (Executable Embedded SQL)

Purpose

To remove rows from a table or from a view's base table.

Prerequisites

For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

The DELETE ANY TABLE system privilege also allows you to delete rows from any table or any view's base table.

If you are using Trusted Oracle in DBMS MAC mode, your DBMS label must dominate the creation label of the table or view or you must meet one of the following criteria:

- If the creation label of the table or view is higher than your DBMS label, you must have READUP and WRITEUP system privileges.

- If the creation label of your table or view is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

In addition, for each row to be deleted, your DBMS label must match the row's label or you must meet one of the following criteria:

- If the row's label is higher than your DBMS label, you must have READUP and WRITEUP system privileges.

- If the row's label is lower than your DBMS label, you must have WRITEDOWN system privilege.

- If the row label is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

Syntax

Keywords and Parameters

AT		identifies the database to which the DELETE statement is ssued. The database can be identified by either:
	db_name	is a database identifier declared in a previous DECLARE DATABASE statement.
	:host_variable	is a host variable whose value is a previously declared *db_name*.
	If you omit this clause, the DELETE statement is issued to your default database.	
FOR *:host_integer*		limits the number of times the statement is executed if the WHERE clause contains array host variables. If you omit this clause, Oracle8 executes the statement once for each component of the smallest array.
schema		is the schema containing the table or view. If you omit *schema*, Oracle8 assumes the table or view is in your own schema.
table view		is the name of a table from which the rows are to be deleted. If you specify *view*, Oracle8 deletes rows from the view's base table.

dblink	is the complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see Chapter 2 of the *Oracle8 SQL Reference.* You can only delete rows from a remote table or view if you are using Oracle8 with the distributed option.
	If you omit *dblink*, Oracle8 assumes that the table or view is located on the local database.
part_name	name of partition in the table
alias	is an alias assigned to the table. Aliases are generally used in DELETE statements with correlated queries.
WHERE	specifies which rows are deleted:

	condition	deletes only rows that satisfy the condition. This condition can contain host variables and optional indicator variables. See the syntax description of *condition* in the *Oracle8 SQL Reference*.
	CURRENT OF	deletes only the row most recently fetched by the *cursor*. The *cursor* cannot be associated with a SELECT statement that performs a join, unless its FOR UPDATE clause specifically locks only one table.

If you omit this clause entirely, Oracle8 deletes all rows from the table or view.

Usage Notes

The host variables in the WHERE clause must be either all scalars or all arrays. If they are scalars, Oracle8 executes the DELETE statement only once. If they are arrays, Oracle8 executes the statement once for each set of array components. Each execution may delete zero, one, or multiple rows.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times Oracle8 executes the statement is determined by the smaller of the following values:

- the size of the smallest array
- the value of the :*host_integer* in the optional FOR clause

If no rows satisfy the condition, no rows are deleted and the SQLCODE returns a NOT_FOUND condition.

The cumulative number of rows deleted is returned through the SQLCA. If the WHERE clause contains array host variables, this value reflects the total number of rows deleted for all components of the array processed by the DELETE statement.

If no rows satisfy the condition, Oracle8 returns an error through the SQLCODE of the SQLCA. If you omit the WHERE clause, Oracle8 raises a warning flag in the fifth component of SQLWARN in the SQLCA. For more information on this command and the SQLCA, see "Using the SQL Communications Area" on page 9-19.

You can use comments in a DELETE statement to pass instructions, or *hints*, to the Oracle8 optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle8 Tuning.*

Example
This example illustrates the use of the DELETE statement within a Pro*COBOL embedded SQL program:

```
    EXEC SQL DELETE FROM EMP
       WHERE DEPTNO = :DEPTNO
       AND JOB = :JOB
END-EXEC.
EXEC SQL DECLARE EMPCURSOR CURSOR
       FOR SELECT EMPNO, COMM
       FROM EMP
END-EXEC.
EXEC SQL OPEN EMPCURSOR END-EXEC.
EXEC SQL FETCH EMPCURSOR
       INTO :EMP-NUMBER, :COMMISSION
END-EXEC.
EXEC SQL DELETE FROM EMP
       WHERE CURRENT OF EMPCURSOR
END-EXEC.
```

Related Topics
DECLARE DATABASE (Oracle Embedded SQL Directive)

DECLARE STATEMENT (Embedded SQL Directive)

DESCRIBE (Executable Embedded SQL)

Purpose

To initialize a descriptor to hold descriptions of host variables for a dynamic SQL statement or PL/SQL block.

Prerequisites

You must have prepared the SQL statement or PL/SQL block in a previous embedded SQL PREPARE statement.

Syntax
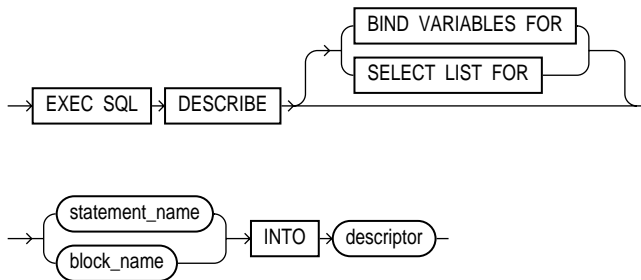
Keywords and Parameters

BIND VARIABLES	initializes the descriptor to hold information about the input variables for the SQL statement or PL/SQL block.
SELECT LIST	initializes the descriptor to hold information about the select list of a SELECT statement.
	The default is SELECT LIST FOR.
statement_name *block_name*	identifies a SQL statement or PL/SQL block previously prepared with a PREPARE statement.
descriptor	is the name of the descriptor to be initialized.

Usage Notes

You must issue a DESCRIBE statement before manipulating the bind or select descriptor within an embedded SQL program.

You cannot describe both input variables and output variables into the same descriptor.

The number of variables found by a DESCRIBE statement is the total number of placeholders in the prepare SQL statement or PL/SQL block, rather than the total number of uniquely named placeholders. For more information on this command, see "The DESCRIBE Statement" on page 11-26.

Example

This example illustrates the use of the DESCRIBE statement in a Pro*COBOL embedded SQL program:

```
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.
EXEC SQL DECLARE EMPCURSOR
    FOR SELECT EMPNO, ENAME, SAL, COMM
    FROM EMP
    WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
EXEC SQL DESCRIBE BIND VARIABLES FOR MYSTATEMENT
    INTO BINDDESCRIPTOR
END-EXEC.
EXEC SQL OPEN EMPCURSOR
    USING BINDDESCRIPTOR
END-EXEC.
EXEC SQL DESCRIBE SELECT LIST FOR MY-STATEMENT
    INTO SELECTDESCRIPTOR
END-EXEC.
EXEC SQL FETCH EMPCURSOR
    INTO SELECTDESCRIPTOR
END-EXEC.
```

Related Topics
PREPARE (Executable Embedded SQL)

EXECUTE ... END-EXEC (Executable Embedded SQL Extension)

Purpose
To embed an anonymous PL/SQL block into an Oracle Pro*COBOL program.

Prerequisites

None.

Syntax

Keywords and Parameters

AT	identifies the database on which the PL/SQL block is executed. The database can be identified by either:

	db_name	is a database identifier declared in a previous DECLARE DATABASE statement.
	:host_variable	is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the PL/SQL block is executed on your default database.

pl/sql_block	For information on PL/SQL, including how to write PL/SQL blocks, see the *PL/SQL User's Guide and Reference*.
END-EXEC	must appear after the embedded PL/SQL block. The keyword END-EXEC must be followed by the statement terminator for COBOL, ".".

Usage Notes

Since the Oracle Precompilers treat an embedded PL/SQL block like a single embedded SQL statement, you can embed a PL/SQL block anywhere in an Oracle Precompiler program that you can embed a SQL statement. For more information on embedding PL/SQL blocks in Oracle Precompiler programs, see Chapter 6, "Using Embedded PL/SQL".

Example

Placing this EXECUTE statement in an Oracle Precompiler program embeds a PL/SQL block in the program:

```
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.
```

Related Topics

EXECUTE IMMEDIATE (Executable Embedded SQL)

EXECUTE (Executable Embedded SQL)

Purpose

To execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block that has been previously prepared with an embedded SQL PREPARE statement.

Prerequisites

You must first prepare the SQL statement or PL/SQL block with an embedded SQL PREPARE statement.
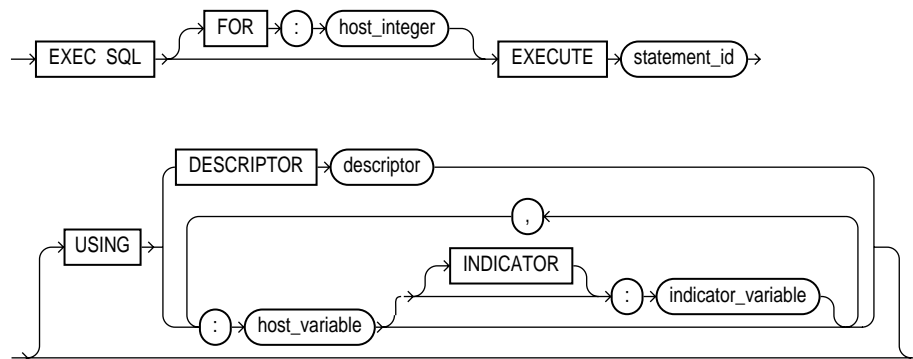
Syntax

Keywords and Parameters

FOR :*host_integer*	limits the number of times the statement is executed when the USING clause contains array host variables If you omit this clause, Oracle8 executes the statement once for each component of the smallest array.
statement_id	is a precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE command to associate the precompiler identifier with the statement or PL/SQL block.
USING	specifies a list of host variables with optional indicator variables that Oracle8 substitutes as input variables into the statement to be executed. The host and indicator variables must be either all scalars or all arrays.

Usage Note

For more information on this command, see Chapter 11, "Using Dynamic SQL".

Example

This example illustrates the use of the EXECUTE statement in a Pro*COBOL embedded SQL program:

```
EXEC SQL PREPARE MY-STATEMENT FROM MY-STRING END-EXEC.
EXEC SQL EXECUTE MY-STATEMENT USING :MY-VAR END-EXEC.
```

Related Topics

DECLARE DATABASE (Oracle Embedded SQL Directive)

PREPARE (Executable Embedded SQL)

EXECUTE IMMEDIATE (Executable Embedded SQL)

Purpose

To prepare and execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block containing no host variables.

Prerequisites

None.

Syntax

Keywords and Parameters

AT		identifies the database on which the SQL statement or PL/SQL block is executed. The database can be identified by either:
	db_name	is a database identifer declared in a previous DECLARE DATABASE statement.
	:host_variable	is a host variable whose value is a previously declared *db_name*.
		If you omit this clause, the statement or block is executed on your default database.
:host_string		is a host variable whose value is the SQL statement or PL/SQL block to be executed.
text		is a quoted text literal containing the SQL statement or PL/SQL block to be executed.
		The SQL statement can only be a DELETE, INSERT, or UPDATE statement.

Usage Notes

When you issue an EXECUTE IMMEDIATE statement, Oracle8 parses the specified SQL statement or PL/SQL block, checking for errors, and executes it. If any errors are encountered, they are returned in the SQLCODE component of the SQLCA.

For more information on this command, see "The EXECUTE IMMEDIATE Statement" on page 11-9.

Example

This example illustrates the use of the EXECUTE IMMEDIATE statement:

```
EXEC SQL
```

```
      EXECUTE IMMEDIATE 'DELETE FROM EMP WHERE EMPNO = 9460'
END-EXEC.
```

Related Topics

PREPARE (Executable Embedded SQL)

EXECUTE (Executable Embedded SQL)

FETCH (Executable Embedded SQL)

Purpose

To retrieve one or more rows returned by a query, assigning the select list values to host variables.

Prerequisites

You must first open the cursor with an the OPEN statement.

Syntax
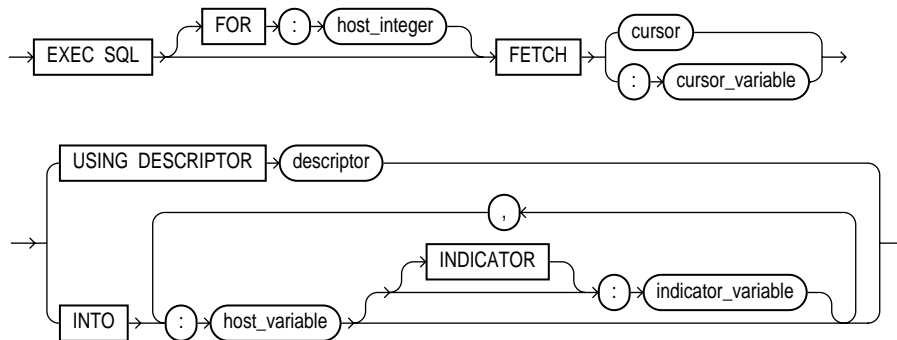
Keywords and Parameters

FOR *:host_integer*   limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle8 fetches enough rows to fill the smallest array.

cursor	is a cursor that is declared by a DECLARE CURSOR statement. The FETCH statement returns one of the rows selected by the query associated with the cursor.
:cursor_variable	is a cursor variable is allocated an ALLOCATE statement. The FETCH statement returns one of the rows selected by the query associated with the cursor variable.
INTO	specifies a list of host variables and optional indicator variables into which data is fetched. These host variables and indicator variables must be declared within the program.
USING	specifies the descriptor referenced in a previous DESCRIBE statement. Only use this clause with dynamic embedded SQL, method 4. Also, the USING clause does not apply when a cursor variable is used.

Usage Notes

The FETCH statement reads the rows of the active set and names the output variables which contain the results. Indicator values are set to -1 if their associated host variable is null. The first FETCH statement for a cursor also sorts the rows of the active set, if necessary.

The number of rows retrieved is specified by the size of the output host variables and the value specified in the FOR clause. The host variables to receive the data must be either all scalars or all arrays. If they are scalars, Oracle8 fetches only one row. If they are arrays, Oracle8 fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle8 fetches is determined by the smaller of the following values:

- the size of the smallest array

- the value of the :*host_integer* in the optional FOR clause

Of course, the number of rows fetched can be further limited by the number of rows that actually satisfy the query.

If a FETCH statement does not retrieve all rows returned by the query, the cursor is positioned on the next returned row. When the last row returned by the query has been retrieved, the next FETCH statement results in an error code returned in the SQLCODE element of the SQLCA.

Note that the FETCH command does not contain an AT clause. You must specify the database accessed by the cursor in the DECLARE CURSOR statement.

You can only move forward through the active set with FETCH statements. If you want to revisit any of the previously fetched rows, you must reopen the cursor and fetch each row in turn. If you want to change the active set, you must assign new values to the input host variables in the cursor's query and reopen the cursor.

Example

This example illustrates the FETCH command in a Pro*COBOL embedded SQL program:

```
    EXEC SQL DECLARE EMPCURSOR CURSOR FOR
        SELECT JOB, SAL FROM EMP WHERE DEPTNO = 30
    END-EXEC.
    ...
    EXEC SQL WHENEVER NOT FOUND GOTO ...
 LOOP.
    EXEC SQL FETCH EMPCURSOR INTO :JOB-TITLE1, :SALARY1 END-EXEC.
    EXEC SQL FETCH EMPCURSOR INTO :JOB-TITLE2, :SALARY2 END-EXEC.
    ...
    GO TO LOOP.
    ...
```

Related Topics

PREPARE (Executable Embedded SQL)

DECLARE CURSOR (Embedded SQL Directive)

OPEN (Executable Embedded SQL)

CLOSE (Executable Embedded SQL)

INSERT (Executable Embedded SQL)

Purpose

To add rows to a table or to a view's base table.

Prerequisites

For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

The INSERT ANY TABLE system privilege also allows you to insert rows into any table or any view's base table.

If you are using Trusted Oracle in DBMS MAC mode, your DBMS label must match the creation label of the table or view:

- If the creation label of the table or view is higher than your DBMS label, you must have WRITEUP system privileges.

- If the creation label of the table or view is lower than your DBMS label, you must have WRITEDOWN system privilege.

- If the creation label of your table or view is not comparable to your DBMS label, you must have WRITEUP and WRITEDOWN system privileges.

Syntax

```
                    ┌──────────┐
                    │  db_name │
        ┌──── AT ───┤          ├──── ┌── FOR ─(:)─ host_integer ──┐
EXEC SQL ┤          └(:)─ host_variable ┘                          │
```

```
                    ┌─ subquery ──────────────────────────────────────────────┐
                    │                                    ┌─(@)─ db_link ─┐      │
                    │  schema ─(.)─ ┌─ table ─┐          └─ PARTITION ─(()─ part_name ─()) ─┘
INSERT INTO ────────┤              └─ view ──┘
```

```
        ┌────,────┐
     (()─ column ─()) ──── VALUES ── (()─ expr ─())
                                     (()─ subquery ─())
```
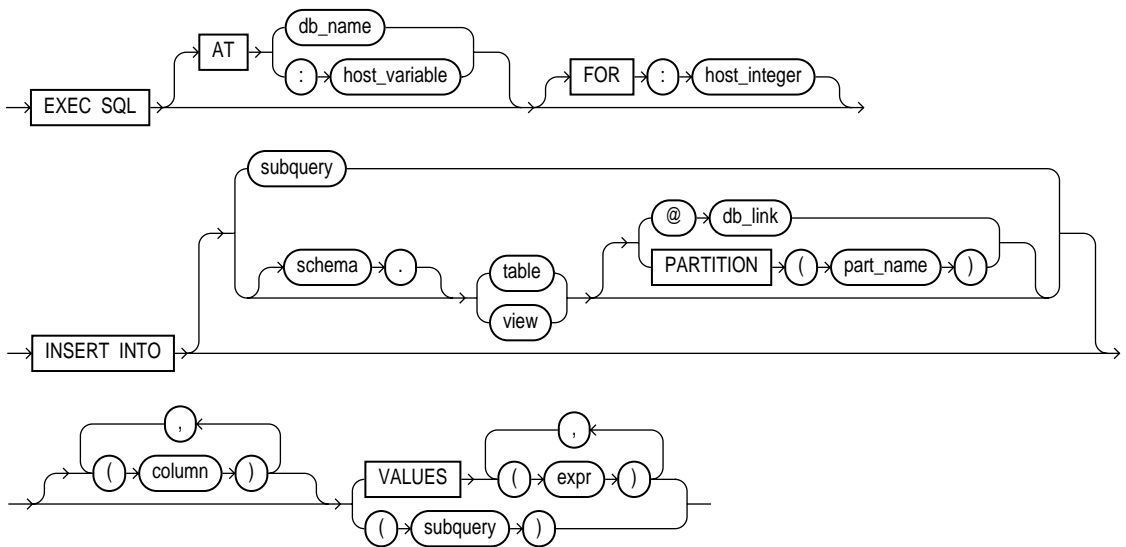
Keywords and Parameters

AT	identifies the database on which the INSERT statement is executed. The database can be identified by either:

db_name	is a database identifier declared in a previous DECLARE DATABASE statement.
:host_variable	is a host variable whose value is a previously declared *db_name*

If you omit this clause, the INSERT statement is executed on your default database.

FOR :*host_integer*	limits the number of times the statement is executed if the VALUES clause contains array host variables. If you omit this clause, Oracle8 executes the statement once for each component in the smallest array.
schema	is the schema containing the table or view. If you omit *schema*, Oracle8 assumes the table or view is in your own schema.
table view	is the name of the table into which rows are to be inserted. If you specify *view*, Oracle8 inserts rows into the view's base table.
db_link	is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see the *Oracle8 SQL Reference*. You can only insert rows into a remote table or view if you are using Oracle8 with the distributed option.
	If you omit *dblink*, Oracle8 assumes that the table or view is on the local database.
part_name	name of partition in the table
column	is a column of the table or view. In the inserted row, each column in this list is assigned a value from the VALUES clause or the query.
	If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. If you omit the column list altogether, the VALUES clause or query must specify values for all columns in the table.
VALUES	specifies a row of values to be inserted into the table or view. See the syntax description of *expr* in the *Oracle8 SQL Reference* Note that the expressions can be host variables with optional indicator variables. You must specify an expression in the VALUES clause for each column in the column list.
subquery	is a subquery that returns rows that are inserted into the table. The select list of this subquery must have the same number of columns as the column list of the INSERT statement. For the syntax description of a subquery, see "SELECT" in the *Oracle8 SQL Reference*.

Usage Notes

Any host variables that appear in the WHERE clause must be either all scalars or all arrays. If they are scalars, Oracle8 executes the INSERT statement once. If they are arrays, Oracle8 executes the INSERT statement once for each set of array components, inserting one row each time.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times Oracle8 executes the statement is determined by the smaller of the following values:

- size of the smallest array

- the value of the :*host_integer* in the optional FOR clause.

For more information on this command, see "The Basic SQL Statements" on page 5-7.

Example I
This example illustrates the use of the embedded SQL INSERT command:

```
EXEC SQL
    INSERT INTO EMP (ENAME, EMPNO, SAL)
    VALUES (:ENAME, :EMPNO, :SAL)
END-EXEC.
```

Example II
This example shows an embedded SQL INSERT command with a subquery:

```
EXEC SQL
    INSERT INTO NEWEMP (ENAME, EMPNO, SAL)
    SELECT ENAME, EMPNO, SAL FROM EMP
    WHERE DEPTNO = :DEPTNO
END-EXEC.
```

Related Topics
DECLARE DATABASE (Oracle Embedded SQL Directive)

OPEN (Executable Embedded SQL)

Purpose
To open a cursor, evaluating the associated query and substituting the host variable names supplied by the USING clause into the WHERE clause of the query.

Prerequisites
You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.

Syntax
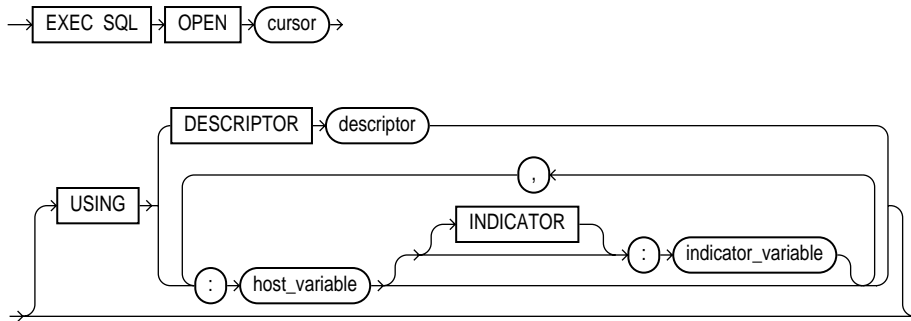
Keywords and Parameters

cursor	is the cursor to be opened.
USING	specifies the host variables to be substituted into the WHERE clause of the associated query.
:host_variable	specifies a host variable with an optional indicator variable to be substituted into the statement associated with the cursor.
DESCRIPTOR	specifies a descriptor that describes the host variables to be substituted into the WHERE clause of the associated query. The *descriptor* must be initialized in a previous DESCRIBE statement.
	The substitution is based on position. The host variable names specified in this statement can be different from the variable names in the associated query.

Usage Notes

The OPEN command defines the active set of rows and initializes the cursor just before the first row of the active set. The values of the host variables at the time of the OPEN are substituted in the statement. This command does not actually retrieve rows; rows are retrieved by the FETCH command.

Once you have opened a cursor, its input host variables are not reexamined until you reopen the cursor. To change any input host variables and therefore the active set, you must reopen the cursor.

All cursors in a program are in a closed state when the program is initiated or when they have been explicitly closed using the CLOSE command.

You can reopen a cursor without first closing it. For more information on this command, see "Opening a Cursor" on page 5-12.

Example

This example illustrates the use of the OPEN command in a Pro*COBOL embedded SQL program:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
     SELECT ENAME, EMPNO, JOB, SAL
     FROM EMP
     WHERE DEPTNO = :DEPTNO
END-EXEC.
EXEC SQL OPEN EMPCURSOR END-EXEC.
```

Related Topics

PREPARE (Executable Embedded SQL)

DECLARE CURSOR (Embedded SQL Directive)

FETCH (Executable Embedded SQL)

CLOSE (Executable Embedded SQL)

PREPARE (Executable Embedded SQL)

Purpose

To parse a SQL statement or PL/SQL block specified by a host variable and associate it with an identifier.

Prerequisites

None.

Syntax
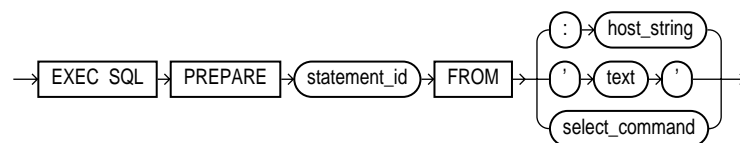
Keywords and Parameters

statement_id	is the identifier to be associated with the prepared SQL statement or PL/SQL block. If this identifier was previously assigned to another statement or block, the prior assignment is superseded.
:host_string	is a host variable whose value is the text of a SQL statement or PL/SQL block to be prepared.
text	is a string literal containing a SQL statement or PL/SQL block to be prepared.
select_command	is a SELECT command.

Usage Notes

Any variables that appear in the :*host_string* or *text* are placeholders. The actual host variable names are assigned in the USING clause of the OPEN command (input host variables) or in the INTO clause of the FETCH command (output host variables).

A SQL statement is prepared only once, but can be executed any number of times.

Example

This example illustrates the use of a PREPARE statement in a Pro*COBOL embedded SQL program:

```
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.
EXEC SQL EXECUTE MYSTATEMENT END-EXEC.
```

Related Topics

DECLARE CURSOR (Embedded SQL Directive)

OPEN (Executable Embedded SQL)

FETCH (Executable Embedded SQL)

CLOSE (Executable Embedded SQL)

ROLLBACK (Executable Embedded SQL)

Purpose

To undo work done in the current transaction. You can also use this command to manually undo the work done by an in-doubt distributed transaction.

Prerequisites

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Syntax
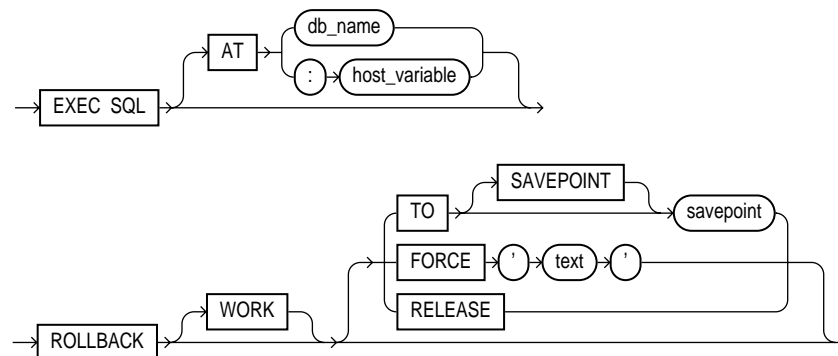
Keywords and Parameters

WORK	is optional and is provided for ANSI compatibility.
TO	rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.

FORCE	manually rolls back an in-doubt distributed transaction. The transaction is identified by the *text* containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING.
	ROLLBACK statements with the FORCE clause are not supported in PL/SQL.
RELEASE	frees all resources and disconnects the application from the Oracle8 Server. The RELEASE clause is not allowed with SAVEPOINT and FORCE clauses.

Usage Notes

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle8 treats as a single unit. A transaction begins with the first executable SQL statement after a COMMIT, ROLLBACK or connection to the database. A transaction ends with a COMMIT statement, a ROLLBACK statement, or disconnection (intentional or unintentional) from the database. Note that Oracle8 issues an implicit COMMIT statement before and after processing any Data Definition Language statement.

Using the ROLLBACK command without the TO SAVEPOINT clause performs the following operations:

- ends the transaction

- undoes all changes in the current transaction

- erases all savepoints in the transaction

- releases the transaction's locks

Using the ROLLBACK command with the TO SAVEPOINT clause performs the following operations:

- rolls back just the portion of the transaction after the savepoint.

- loses all savepoints created after that savepoint. Note that the named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.

- releases all table and row locks acquired since the savepoint. Note that other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

It is recommended that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit

the transaction and the program terminates abnormally, Oracle8 rolls back the last uncommitted transaction.

Example I

The following statement rolls back your entire current transaction:

```
EXEC SQL ROLLBACK END-EXEC.
```

Example II

The following statement rolls back your current transaction to savepoint SP5:

```
EXEC SQL ROLLBACK TO SAVEPOINT SP5 END-EXEC.
```

Distributed Transactions

Oracle8 with the distributed option allows you to perform distributed transactions, or transactions that modify data on multiple databases. To commit or roll back a distributed transaction, you need only issue a COMMIT or ROLLBACK statement as you would any other transaction.

If there is a network failure during the commit process for a distributed transaction, the state of the transaction may be unknown, or in-doubt. After consultation with the administrators of the other databases involved in the transaction, you may decide to manually commit or roll back the transaction on your local database. You can manually roll back the transaction on your local database by issuing a ROLL-BACK statement with the FORCE clause.

For more information on when to roll back in-doubt transactions, see *Oracle8 Distributed Database Systems.*

You cannot manually roll back an in-doubt transaction to a savepoint.

A ROLLBACK statement with a FORCE clause only rolls back the specified transaction. Such a statement does not affect your current transaction.

Example III

The following statement manually rolls back an in-doubt distributed transaction:

```
EXEC SQL ROLLBACK WORK FORCE '25.32.87' END-EXEC.
```

Related Topics

COMMIT (Executable Embedded SQL)

SAVEPOINT (Executable Embedded SQL)

SAVEPOINT (Executable Embedded SQL)

Purpose
To identify a point in a transaction to which you can later roll back.

Prerequisites
None.

Syntax
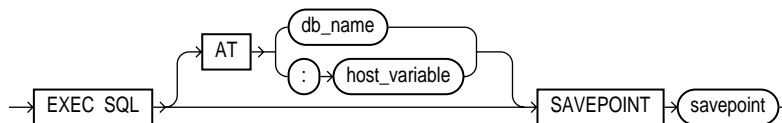
Keywords and Parameters

AT   identifies the database on which the savepoint is created. The database can be identified by either:

     db_name   is a database identifier declared in a previous DECLARE DATABASE statement.

     :host_variable   is a host variable whose value is a previously declared *db_name*.

     If you omit this clause, the savepoint is created on your default database.

savepoint   is the name of the savepoint to be created.

Usage Notes
For more information on this command, see "Using the SAVEPOINT Statement" on page 8-7.

Example
This example illustrates the use of the embedded SQL SAVEPOINT command:

```
EXEC SQL SAVEPOINT SAVE3 END-EXEC.
```

Related Topics
COMMIT (Executable Embedded SQL)

ROLLBACK (Executable Embedded SQL)

SELECT (Executable Embedded SQL)

Purpose
To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.

Prerequisites
For you to select data from a table or snapshot, the table or snapshot must be in your own schema or you must have SELECT privilege on the table or snapshot.

For you to select rows from the base tables of a view, the owner of the schema containing the view must have SELECT privilege on the base tables. Also, if the view is in a schema other than your own, you must have SELECT privilege on the view.

The SELECT ANY TABLE system privilege also allows you to select data from any table or any snapshot or any view's base table.

If you are using Trusted Oracle in DBMS MAC mode, your DBMS label must dominate the creation label of each queried table, view, or snapshot or you must have READUP system privileges.

Syntax
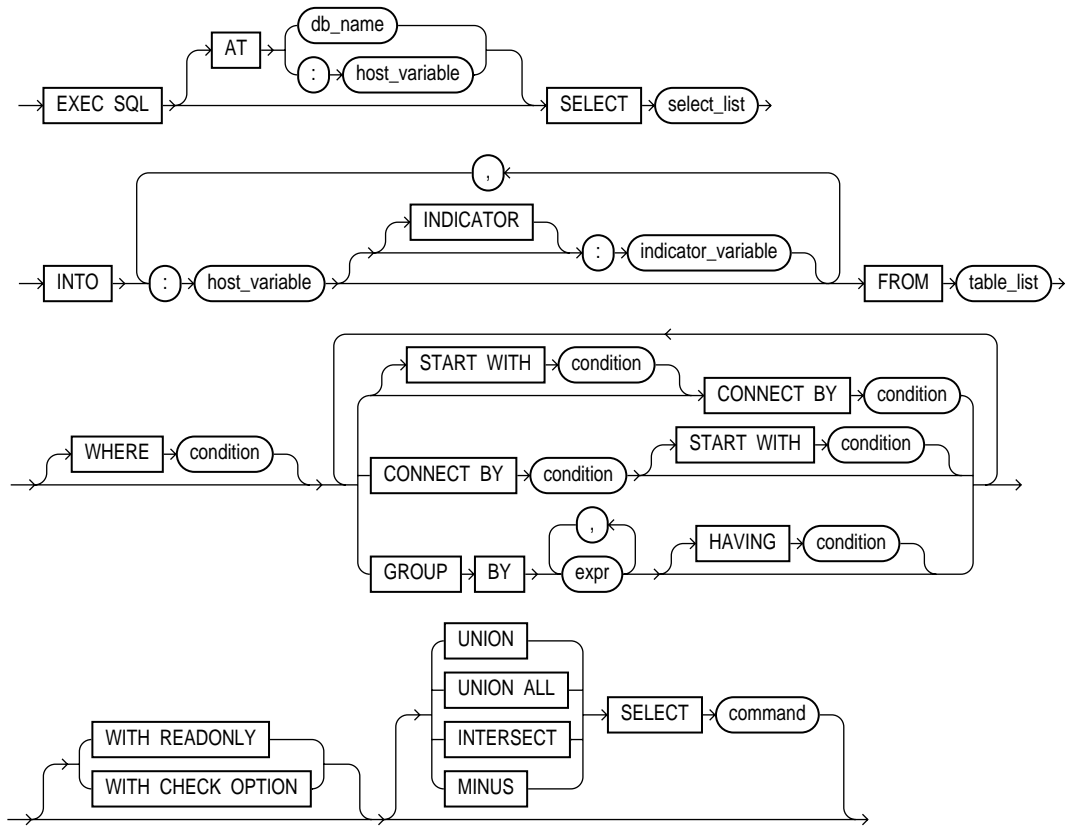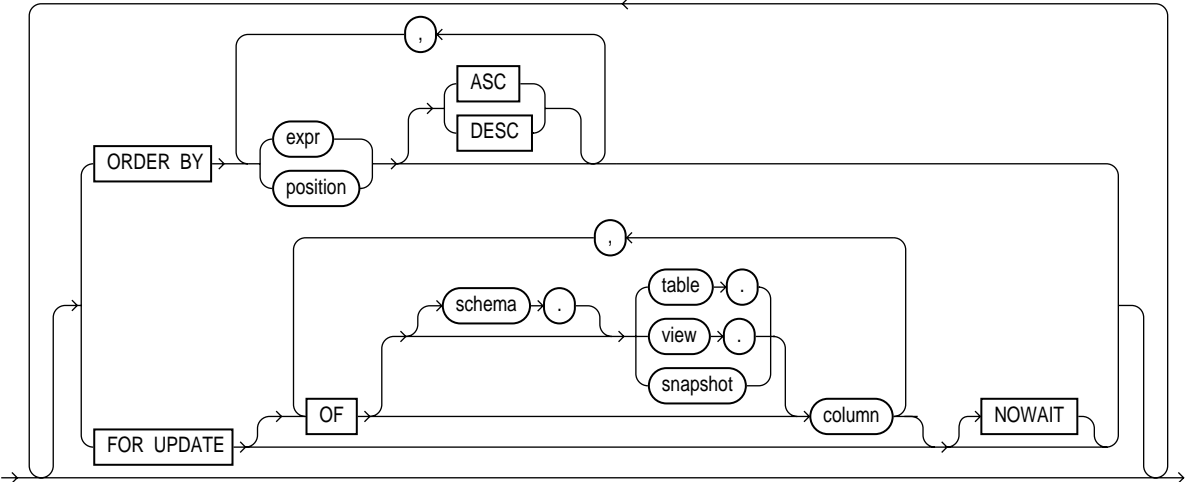
Keywords and Parameters

All other keywords and parameters are identical to the non-embedded SQL

AT	identifies the database to which the SELECT statement is issued. The database can be identified by either:	
	db_name	is a database identifier declared in a previous DECLARE DATABASE statement.
	:host_variable	is a host variable whose value is a previously declared *db_name*.
	If you omit this clause, the SELECT statement is issued to your default database.	
select_list	identical to the non-embedded SELECT command except that a host variables can be used in place of literals.	
INTO	specifies output host variables and optional indicator variables to receive the data returned by the SELECT statement. Note that these variables must be either all scalars or all arrays, but arrays need not have the same size.	
WHERE	restricts the rows returned to those for which the condition is TRUE. See the syntax description of *condition* in the *Oracle8 SQL Reference*. The *condition* can contain host variables, but cannot contain indicator variables. These host variables can be either scalars or arrays.	

SELECT command.

Usage Notes

If no rows meet the WHERE clause condition, no rows are retrieved and Oracle8 returns an error code through the SQLCODE component of the SQLCA.

You can use comments in a SELECT statement to pass instructions, or *hints*, to the Oracle8 optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle8 Tuning*.

Example

This example illustrates the use of the embedded SQL SELECT command:

```
EXEC SQL SELECT ENAME, SAL + 100, JOB
    INTO :ENAME, :SAL, :JOB
    FROM EMP
    WHERE EMPNO = :EMPNO
END-EXEC.
```

Related Topics

DECLARE DATABASE (Oracle Embedded SQL Directive)

DECLARE CURSOR (Embedded SQL Directive)

EXECUTE (Executable Embedded SQL)

FETCH (Executable Embedded SQL)

PREPARE (Executable Embedded SQL)

UPDATE (Executable Embedded SQL)

Purpose
To change existing values in a table or in a view's base table.

Prerequisites
For you to update values in a table or snapshot, the table must be in your own schema or you must have UPDATE privilege on the table.

For you to update values in the base table of a view, the owner of the schema containing the view must have UPDATE privilege on the base table. Also, if the view is in a schema other than your own, you must have UPDATE privilege on the view.

The UPDATE ANY TABLE system privilege also allows you to update values in any table or any view's base table.

If you are using Trusted Oracle in DBMS MAC mode, your DBMS label must match the creation label of the table or view:

- If the creation label of the table or view is higher than your DBMS label, you must have READUP and WRITEUP system privileges

- If the creation label of the table or view is lower than your DBMS label, you must have WRITEDOWN system privilege.

- If the creation label of your table or view is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

Syntax
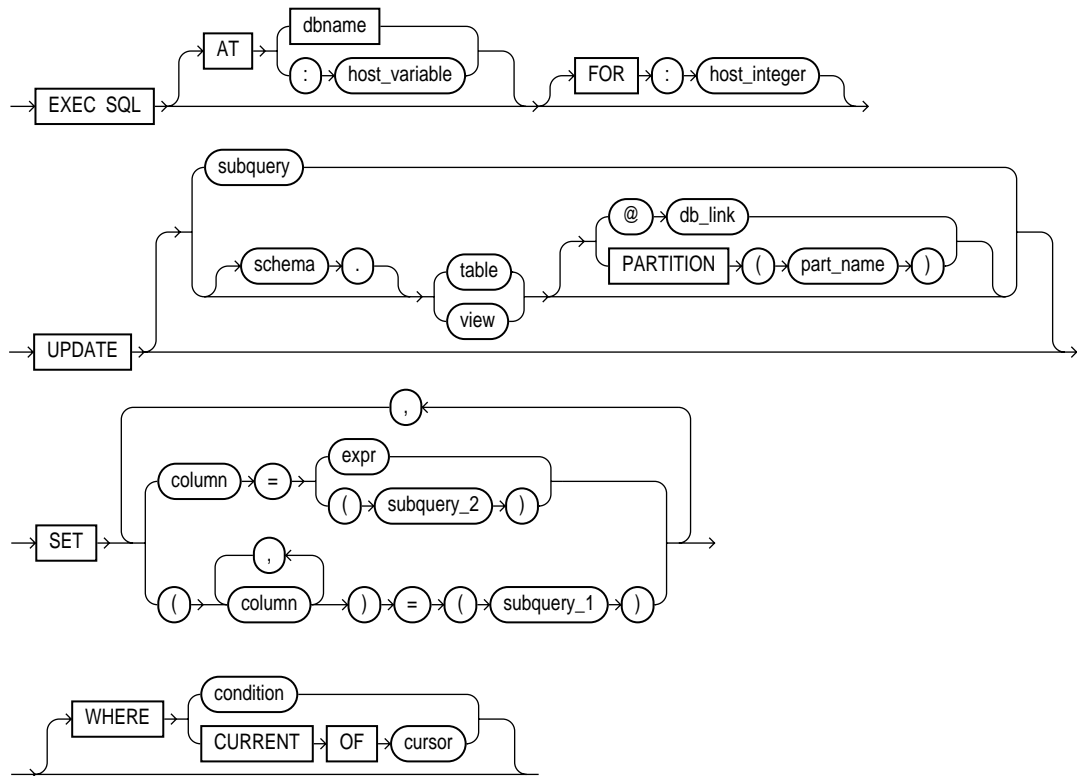
Keywords and Parameters

AT	identifies the database to which the UPDATE statement is issued. The database can be identified by either:

	dbname	is a database identifier declared in a previous DECLARE DATABASE statement.
	:host_variable	is a host variable whose value is a previously declared *dbname*.

If you omit this clause, the UPDATE statement is issued to your default database.

FOR :*host_integer*   limits the number of times the UPDATE statement is executed if the SET and WHERE clauses contain array host variables. If you omit this clause, Oracle8 executes the statement once for each component of the smallest array.

schema   is the schema containing the table or view. If you omit *schema*, Oracle8 assumes the table or view is in your own schema.

table view   is the name of the table to be updated. If you specify *view*, Oracle8 updates the view's base table.

dblink   is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see the *Oracle8 SQL Reference.* You can only use a database link to update a remote table or view if you are using Oracle8 with the distributed option.

part_name   name of partition in the table

alias   is a name used to reference the table, view, or subquery elsewhere in the statement.

column   is the name of a column of the table or view that is to be updated. If you omit a column of the table from the SET clause, that column's value remains unchanged.

expr   is the new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables. See the syntax of *expr* in the *Oracle8 SQL Reference*.

subquery_1   is a subquery that returns new values that are assigned to the corresponding columns. For the syntax of a subquery, see "SELECT" in the *Oracle8 SQL Reference*.

subquery_2   is a subquery that return a new value that is assigned to the corresponding column. For the syntax of a subquery, see "SELECT" in the *Oracle8 SQL Reference*.

WHERE   specifies which rows of the table or view are updated:

condition	updates only rows for which this condition is true. This condition can contain host variables and optional indicator variables. See the syntax of *condition* in the *Oracle8 SQL Reference*.
CURRENT OF	updates only the row most recently fetched by the *cursor*. The *cursor* cannot be associated with a SELECT statement that performs a join unless its FOR UPDATE clause explicitly locks only one table.

If you omit this clause entirely, Oracle8 updates all rows of the table or view.

Usage Notes

Host variables in the SET and WHERE clauses must be either all

scalars or all arrays. If they are scalars, Oracle8 executes the UPDATE statement only once. If they are arrays, Oracle8 executes the statement once for each set of array components. Each execution may update zero, one, or multiple rows.

Array host variables can have different sizes. In this case, the number of times Oracle8 executes the statement is determined by the smaller

of the following values:

- the size of the smallest array

- the value of the :*host_integer* in the optional FOR clause

The cumulative number of rows updated is returned through the third element of the SQLERRD component of the SQLCA. When arrays are used as input host variables, this count reflects the total number of updates for all components of the array processed in the UPDATE statement. If no rows satisfy the condition, no rows are updated and Oracle8 returns an error message through the SQLCODE element of

the SQLCA. If you omit the WHERE clause, all rows are updated and Oracle8 raises a warning flag in the fifth component of the SQLWARN element of the SQLCA.

You can use comments in an UPDATE statement to pass instructions, or *hints*, to the Oracle8 optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle8 Tuning*.

For more information on this command, see "The Basic SQL Statements" on page 5-7 andChapter 8, "Defining and Controlling Transactions".

Examples

The following examples illustrate the use of the embedded SQL UPDATE command:

```
EXEC SQL UPDATE EMP
    SET SAL = :SAL, COMM = :COMM INDICATOR :COMM-IND
    WHERE ENAME = :ENAME
END-EXEC.

EXEC SQL UPDATE EMP
    SET (SAL, COMM) =
        (SELECT AVG(SAL)*1.1, AVG(COMM)*1.1
        FROM EMP)
```

```
              WHERE ENAME = 'JONES'
       END-EXEC.
```

Related Topics
DECLARE DATABASE (Oracle Embedded SQL Directive)

VAR (Oracle Embedded SQL Directive)

Purpose
To perform *host variable equivalencing,* or to assign a specific Oracle8 external datatype to an individual host variable, overriding the default datatype assignment. There is an optional clause,CONVBUFSZ , that specifies the size of a buffer for character set conversion.

Prerequisites
The host variable must be previously declared in the embedded SQL program.

Syntax
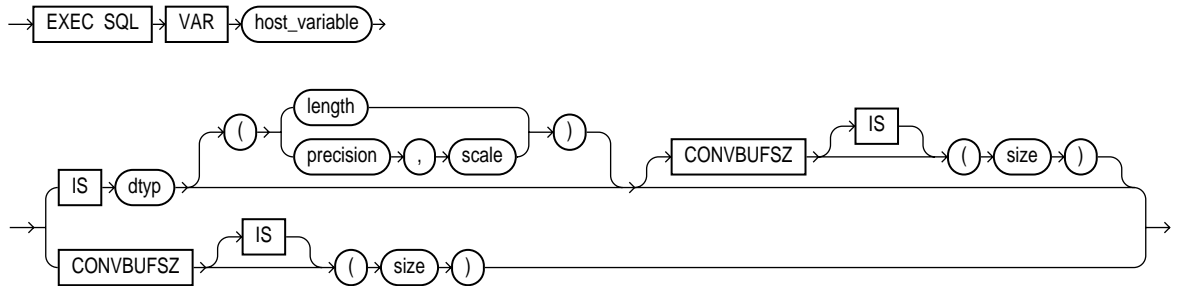
Keywords and Parameters

host_variable   is the host variable to be assigned an Oracle8 external datatype.

dtyp	is an Oracle8 external datatype recognized by the Oracle Precompilers (not an Oracle8 internal datatype). The datatype may include a length, precision, or scale. This external datatype is assigned to the *host_variable*. For a list of external datatypes, see "External Datatypes" on page 4-9..
size	is the size in bytes of a buffer in the Oracle8 runtime library used to perform conversion between character sets of the host_variable

Usage Notes

Host variable equivalencing is one kind of datatype equivalencing. Datatype equivalencing is useful for any of the following purposes:

- to automatically null-terminate a character host variable

- to store program data as binary data in the database

- to override default datatype conversion

For more information about Oracle datatypes, see "External Datatypes" on page 4-9 and "Sample Program 4: Datatype Equivalencing" on page 5-19.

Example

This example equivalences the host variable DEPT_NAME to

the datatype STRING and the host variable BUFFER to the

datatype RAW(200):

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    ...
    01 DEPT-NAME  PIC X(15).
* -- default datatype is CHAR
    EXEC SQL VAR DEPT-NAME IS STRING END-EXEC.
* -- reset to STRING
    ...
    01 BUFFER-VAR.
        05 BUFFER  PIC X(200).
* -- default datatype is CHAR
    EXEC SQL VAR BUFFER IS RAW(200) END-EXEC.
* -- refer to RAW
    ...
    EXEC SQL END DECLARE SECTION END-EXEC.
```

Related Topics

None.

WHENEVER (Embedded SQL Directive)

Purpose
To specify the action to be taken when an error or warning results from executing an embedded SQL program.

Prerequisites
None.

Syntax
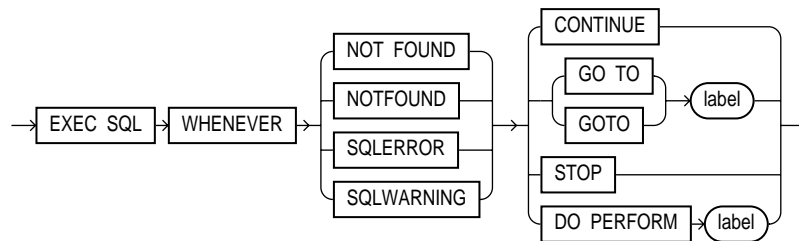
Keywords and Parameters

_{» Notes}

NOT FOUND | NOTFOUND	identifies any exception condition that returns an error code of +1403 to SQLCODE (or a +100 code when MODE=ANSI).
SQLERROR	identifies a condition that results in a negative return code.
SQLWARNING	identifies a non-fatal warning condition.
CONTINUE	indicates that the program should progress to the next statement.
GOTO | GO TO	indicates that the program should branch to the statement named by *label*.
STOP	stops program execution.
DO PERFORM	indicates that the program should perform a routine at *label*.

The WHENEVER command allows your program to transfer control to an error handling routine in the event an embedded SQL statement results in an error or warning.

The scope of a WHENEVER statement is positional, rather than logical. A WHENEVER statement applies to all embedded SQL statements that textually follow it in the source file, not in the flow of the program logic. A WHENEVER statement remains in effect until it is superseded by another WHENEVER statement checking for the same condition.

For more information on this command, see Chapter 8, "Defining and Controlling Transactions". Do not confuse the WHENEVER embedded SQL command with the WHENEVER SQL*Plus command.

Example

The following example illustrates the use of the WHENEVER command in a Pro*COBOL embedded SQL program:

```
    EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
    ...
    EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
    ...
 SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY "ORACLE ERROR DETECTED.".
    EXEC SQL ROLLBACK RELEASE END-EXEC.
    STOP RUN.
```

Related Topics

None

Index

O

X