Pro*COBOL Precompiler

Programmer's Guide

Release 9.0.1

June 2001

Part No. A89865-01

ORACLE

Pro*COBOL Precompiler, Release 9.0.1

Part No. A89865-01

Primary Authors: Jack Melnick and James W. Rawles

Contributors: Beethoven Cheng, Michael Chiocca, Nancy Ikeda, Maura Joglekar, Alex Keh, Thomas Kurian, Shiao-Yen Lin, Diana Lorentz, Lee Osborne, Jacqui Pons, Ajay Popat, Chris Racicot, Pamela Rothman, Simon Slack, Gael Stevens, and Eric Wan

Graphic Designer: Valarie Moore

Contents

3 Database Concepts

4 Datatypes and Host Variables

5
Embedded SQL

6 Embedded PL/SQL

7 Host Tables

8 Error Handling and Diagnostics

9 Oracle Dynamic SQL

10 ANSI Dynamic SQL

11 Oracle Dynamic SQL: Method 4

12 Multithreaded Applications

13 Large Objects (LOBs)

14 Precompiler Options

A New Features

B Operating System Dependencies

C Reserved Words, Keywords, and Namespaces

D Performance Tuning

E Syntactic and Semantic Checking

F Embedded SQL Statements and Precompiler Directives

Send Us Your Comments

Pro*COBOL Precompiler Programmer's Guide, Release 9.0.1

Part No. A89865-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. 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?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and 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) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

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 *Oracle9i SQL Reference* and *PL/SQL User's Guide and Reference* for more information on SQL and PL/SQL.

This preface contains these topics:

Audience

*Pro*COBOL Precompiler Programmer's Guide* is intended for anyone developing new COBOL applications or converting existing applications to run in the Oracle9*i* environment. 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
- Oracle9*i* concepts and terminology

In This Manual

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 key methodology, 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

---

See Also: For information about migrating your applications to Oracle9*i*, see *Oracle9i Database Migration*

Organization

This book presents a programmer's guide to Pro*COBOL in fourteen chapters and six appendixes:

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 are its key benefits and features.

Chapter 2, "Precompiler Concepts"

This chapter explains how embedded SQL programs work. Then the guidelines for programming in Pro*COBOL are presented. Compilation issues are discussed and the sample Oracle tables used in this guide are presented, as is the first of the demo programs, SAMPLE1.PCO.

Chapter 3, "Database Concepts"

This chapter describes transaction processing. You learn the basic techniques that safeguard the consistency of your database. You then learn how to connect to a database and how to connect to multiple distributed databases.

Chapter 4, "Datatypes and Host Variables"

The internal and external datatypes are defined at length. Then you are shown how to use the datatypes in your COBOL program. Then runtime contexts and ROWIDs are explained, followed by Globalization Support, datatype conversion and datatype equivalencing (with a sample program).

Chapter 5, "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, "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 in either PL/SQL or Java, host tables, and dynamic PL/SQL.

Chapter 7, "Host Tables"

This chapter looks at using host (COBOL) 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 8, "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 9, "Oracle Dynamic SQL"

This chapter shows you how to take advantage of dynamic SQL. You are taught three methods, from simple to complex, for writing flexible programs that let users build SQL statements interactively at run time.

Chapter 10, "ANSI Dynamic SQL"

ANSI Dynamic SQL, Method 4, is presented. This method supports all Oracle datatypes, while the older Oracle Method 4 does not support cursor variables, tables of group items, DML Returning Clause, and LOBs. ANSI Method 4 uses embedded SQL statements that set up descriptor areas in memory. ANSI SQL should be used for all new applications.

Chapter 11, "Oracle Dynamic SQL: Method 4"

This chapter shows you how to maintain existing applications that use dynamic SQL Method 4. Numerous examples are used to illustrate the method.

Chapter 12, "Multithreaded Applications"

How to write multithreaded applications is discussed in this chapter. (Your compiler must support multithreading.)

Chapter 13, "Large Objects (LOBs)"

This chapter presents large object datatypes (BLOBs, CLOBs, NCLOBs, and BFILEs). The embedded SQL commands that provide functionality comparable to OCI and PL/SQl are presented and used in sample code.

Chapter 14, "Precompiler Options"

This chapter details the requirements for running the Pro*COBOL precompiler and lists the precompiler options. You learn what happens during precompilation, how to issue the Pro*COBOL command, and how to specify the many useful precompiler options.

Appendix A, "New Features"

This appendix highlights the improvements and new features introduced in releases 8.1 and 8.0 of Pro*COBOL.

Appendix B, "Operating System Dependencies"

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

Appendix C, "Reserved Words, Keywords, and Namespaces"

This appendix refers you to a table of reserved words that have a special meaning to Pro*COBOL, and presents the 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 Statements and Precompiler Directives"

This appendix contains descriptions of precompiler directives, embedded SQL commands, and Oracle embedded SQL extensions. The purpose, prerequisites, syntax diagrams, keywords, parameters, usage notes, examples, and related topics are presented for each statement and directive.

Sample Pro*COBOL 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 complete sample program in this manual is available on-line in the demo directory. However, the exact filenames are system-dependent. For exact filenames, see your Oracle system-specific documentation. We present sample code developed for the Solaris operating system in this manual.

The Pro*COBOL Precompiler and 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
http://www.ansi.org

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
http://www.nist.gov

Compliance

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 element that violates ANSI format or syntax rules, except rules enforcing privileges. For a list of Oracle extensions to standard SQL, see the *Oracle9i 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 14-20.

Certification

The 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 and Technology
Gaithersburg, MD 20899
USA
`http://www.nist.gov`

MIA/SPIRIT

The Pro*COBOL Precompiler provides Globalization Support (formerly called National Language Support or NLS) of multibyte 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.

Conventions

This section describes the conventions used in the text and code examples of the this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	The C datatypes such as **ub4**, **sword**, or **OCINumber** are valid. When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles, emphasis, syntax clauses, or placeholders.	*Oracle9i Database Concepts* You can specify the *parallel_clause*. Run `U`*`old_release`*`.SQL` where *old_release* refers to the release you installed prior to upgrading.
UPPERCASE monospace (fixed-width font)	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, user names, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Specify the `ROLLBACK_SEGMENTS` parameter. Use the `DBMS_STATS.GENERATE_STATS` procedure.
lowercase monospace (fixed-width font)	Lowercase monospace typeface indicates executables and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, user names and roles, program units, and parameter values.	Enter `sqlplus` to open SQL*Plus. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (digits [ , precision ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
...	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , col`*`n`* `FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as it is shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.	`SELECT last_name, employee_id FROM employees;` `sqlplus hr/hr`

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

1

Introduction

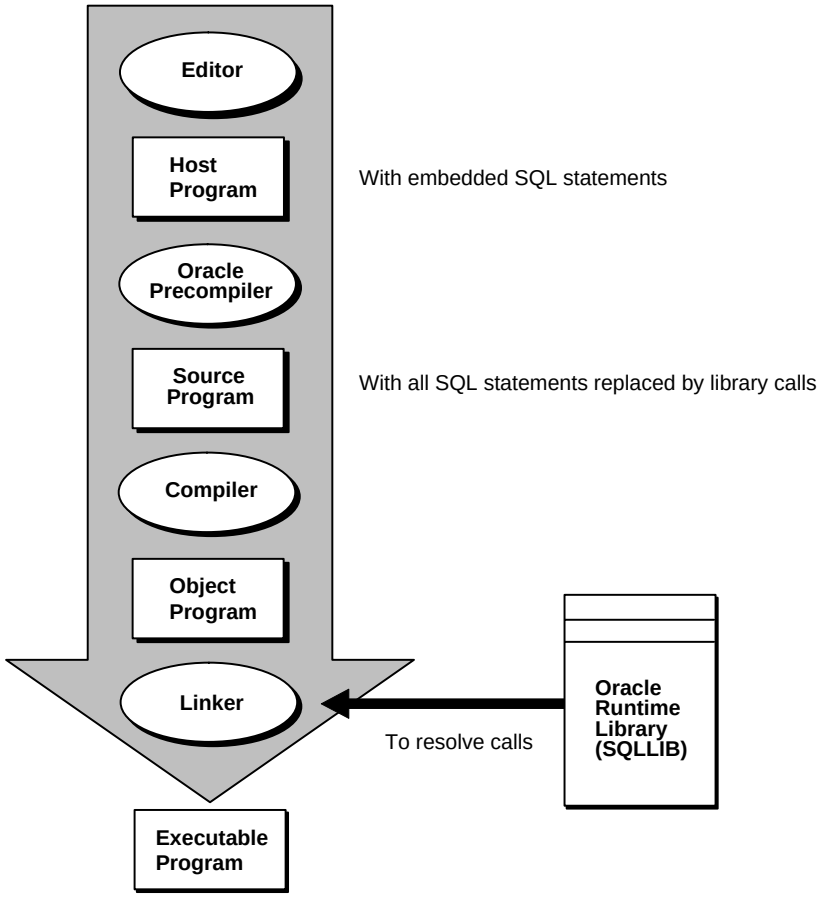
This chapter introduces you to the Pro*COBOL Precompiler. You look at its role in developing application programs that manipulate Oracle data and find out what it enables your applications to do. The following questions are answered:

- The Pro*COBOL Precompiler
- Advantages of the Pro*COBOL Precompiler
- The SQL Language
- The PL/SQL Language
- Pro*COBOL Features and Benefits

The Pro*COBOL Precompiler

The Pro*COBOL Precompiler is a programming tool that enables you to embed SQL statements in a host COBOL 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



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 again. However, Oracle will continue to issue patch releases for Pro*FORTRAN as bugs are reported and corrected.

Advantages of the Pro*COBOL Precompiler

The Pro*COBOL Precompiler lets you pack the power and flexibility of SQL into your application programs. You can embed SQL statements in 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. It enables close monitoring of resource usage, SQL statement execution, and various run-time indicators. With this information, you can adjust program parameters for maximum performance.

The SQL Language

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. Usually, only minor changes are required to switch from interactive to embedded SQL.

The PL/SQL Language

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.

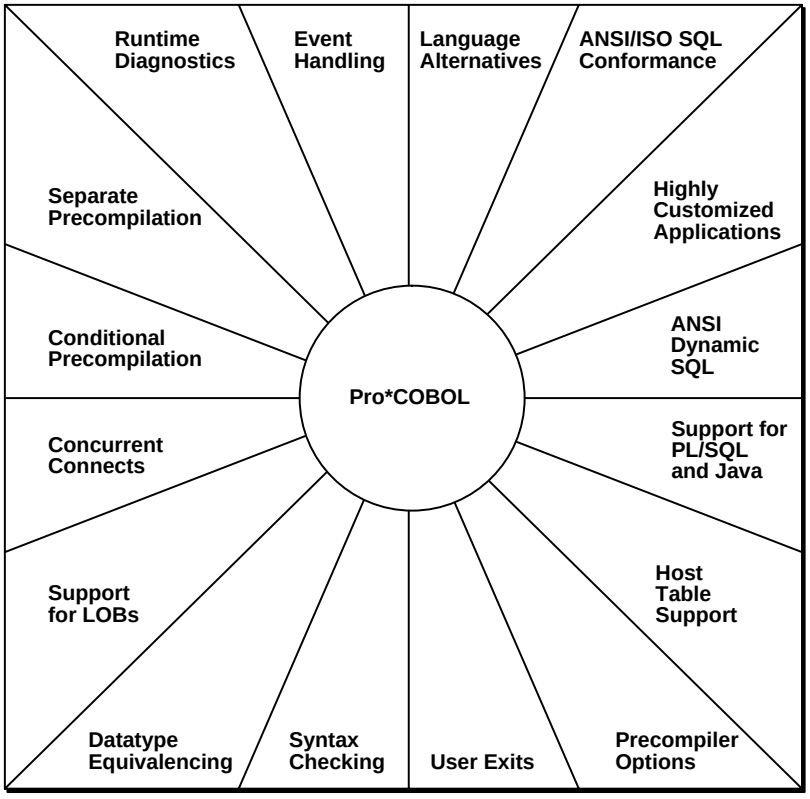
The main advantage of embedded PL/SQL is better performance. Unlike SQL, PL/SQL enables you to group SQL statements logically and send them to Oracle in a block rather than one by one. This reduces network traffic and processing overhead.

For more information about PL/SQL including how to embed it in an application program, see Chapter 6, "Embedded PL/SQL".

Pro*COBOL Features and Benefits

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

Figure 1–2 Pro*COBOL Features and Benefits



For example, the Pro*COBOL Precompiler enables you to:

- Write your application in COBOL.
- Conform to the ANSI/ISO embedded SQL standard.
- Take advantage of ANSI Dynamic SQL Method 4, 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 Oracle9*i* 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 Oracle9*i* interprets input data and formats output data.
- Precompile several program modules separately, and then link them into one executable program.
- Check the syntax and semantics of embedded SQL data manipulation statements and PL/SQL blocks.
- Access Oracle9*i* databases on multiple nodes concurrently, using Oracle Net (formerly called 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 through 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).
- Access Large Object (LOB) database types.

2

Precompiler Concepts

This chapter explains how embedded SQL programs do their work. It presents definitions of important words, explanations of basic concepts, and key rules.

Topics covered are:

Key Concepts of Embedded SQL Programming

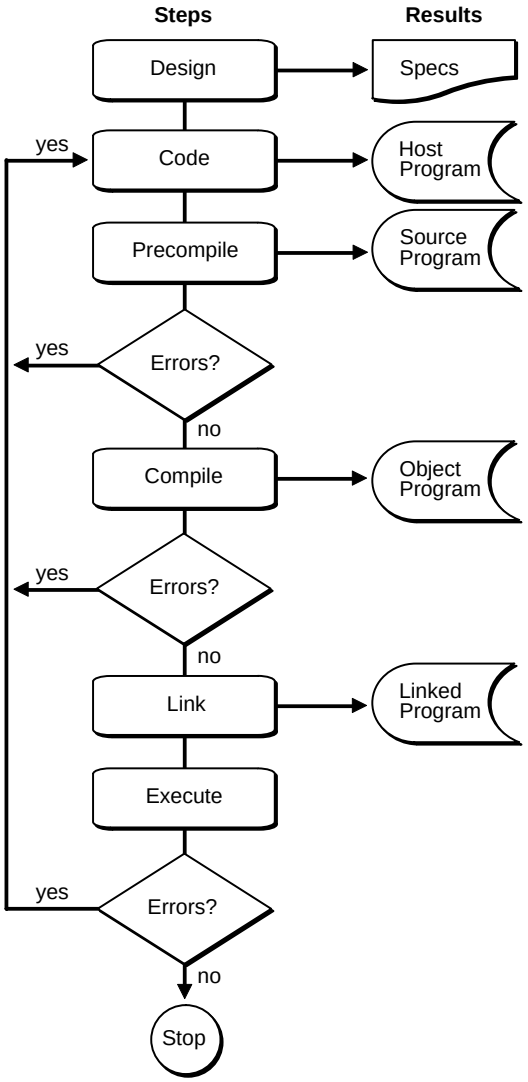
This section lays the conceptual foundation on which later chapters build.

Steps in Developing an Embedded SQL Application

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.

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

Figure 2–1 Application Development Process



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.

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 statements* and *directives*.

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.

Directives, on the other hand, do not result in calls to SQLLIB and do not operate on Oracle data.

You use directives to declare Oracle objects, communications areas, and SQL variables. They can be placed wherever COBOL declarations can be placed.

Appendix F, "Embedded SQL Statements and Precompiler Directives" contains a presentation of the most important statements and directives.Table 2–1 groups some examples of embedded SQL statements (not a complete list.)

Table 2–1 Embedded SQL Statements

Directives	
STATEMENT	**PURPOSE**
ARRAYLEN*	To use host tables with PL/SQL
BEGIN DECLARE SECTION*	To declare host variables
END DECLARE SECTION*	
DECLARE*	To name Oracle objects
INCLUDE*	To copy in files

Table 2–1 Embedded SQL Statements

VAR*	To equivalence variables
WHENEVER*	To handle runtime errors
Executable SQL	
STATEMENT	PURPOSE
ALLOCATE*	To define and control Oracle data
ALTER	
CONNECT*	
CREATE	
DROP	
GRANT	
NOAUDIT	
RENAME	
REVOKE	
TRUNCATE	
CLOSE*	To query and manipulate Oracle data
DELETE	
EXPLAIN PLAN	
FETCH*	
INSERT	
LOCK TABLE	
OPEN*	
SELECT	
UPDATE	
COMMIT	To process transactions
ROLLBACK	
SAVEPOINT	
SET TRANSACTION	
DESCRIBE*	To use dynamic SQL
EXECUTE*	
PREPARE*	

Table 2–1 Embedded SQL Statements

ALTER SESSION	To control sessions
SET ROLE	

*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 addition 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.
```

A period or any other terminator can follow a SQL statement. Either of the following is allowed:

```
     EXEC SQL ... END-EXEC,
     EXEC SQL ... 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, "Embedded SQL".

However, some applications are required to accept and process any valid SQL statement at run time. In this case you might not know until run time 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 9, "Oracle Dynamic SQL", Chapter 10, "ANSI Dynamic SQL", and Chapter 11, "Oracle Dynamic SQL: Method 4".

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, "Embedded PL/SQL".

Host Variables and Indicator Variables

A *host variable* is a scalar or table variable or group item declared in the COBOL 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. 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. A *NULL* is a missing, an unknown, or an inapplicable value. You use indicator variables to assign NULLs to input host variables and to detect NULLs in output variables or truncated values in output character host variables.

A host variable must *not* be:

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

In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable (to improve readability, you can precede the indicator variable with the optional keyword INDICATOR).

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 a host variable identifier, 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 4–9, "Conversions Between Internal and External Datatypes" shows the compatible datatypes between which Oracle9*i* converts automatically when necessary.

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, it does any conversion 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.

> **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 4-44.

Tables

Pro*COBOL lets you define table host variables (called *host tables*) and operate on them with a single SQL statement. Using the SELECT, FETCH, DELETE, INSERT,

and UPDATE statements, you can query and manipulate large volumes of data with ease.

For a complete discussion of host tables, see Chapter 7, "Host Tables".

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 the following error handling mechanisms:

- SQLCODE status variable
- SQLSTATE status variable
- SQL Communications Area (SQLCA)
- WHENEVER statement
- Oracle Communications Area (ORACA)

SQLCODE/SQLSTATE Status Variables

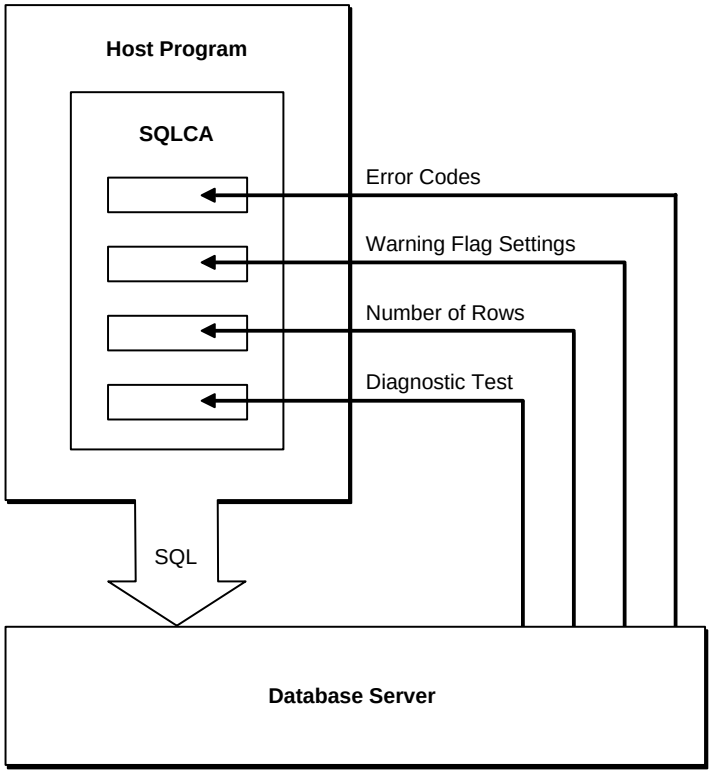
After executing a SQL statement, the Oracle Server returns a status code to a variable named SQLCODE or SQLSTATE. The status code indicates whether the SQL statement executed successfully or caused an error or warning condition.

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.

The SQLCA provides for diagnostic checking and event handling. At runtime, the SQLCA holds status information passed to your program by Oracle9*i*. After executing a SQL statement, Oracle8*i* sets SQLCA variables to indicate the outcome, as illustrated in Figure 2–2.

Figure 2–2 Updating the SQLCA



You can check to see if an INSERT, UPDATE, or DELETE statement succeeded and if it did, 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 8-6 shows you how to declare and use the SQLCA.

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 subprogram, branching to a labeled statement, performing a paragraph, 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.

Precompiler Options and Error Handling

Oracle returns the success or failure of SQL statements in status variables, SQLSTATE and SQLCODE. With precompiler option MODE=ORACLE, you use SQLCODE, declared by including SQLCA. With MODE=ANSI, either SQLSTATE or SQLCODE must be declared, but SQLCA is not necessary.

For more information, see Chapter 8, "Error Handling and Diagnostics".

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 Supported

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 Oracle system-specific documentation.

Coding Areas

The precompiler option FORMAT, specifies the format of your source code. If you specify FORMAT=ANSI (the default), you are conforming as much as possible to the ANSI standard. In this format, columns 1 through 6 can contain an optional sequence number, and column 7 (indicator area) can indicate comments or continuation lines.

Division headers, section headers, paragraph names, FD and 01 statements begin in columns 8 through 11 (area A). Other statements, including EXEC SQL and EXEC ORACLE statements, must be placed in area B (columns 12 through 72). These guidelines for source code format can be overridden by your compiler's rules.

If you specify FORMAT=TERMINAL, COBOL statements can begin in column 1 (the left-most column), or column 1 can be the indicator area. This format is also subject to the rules of your compiler.

Consult your COBOL compiler documentation for your own platform to determine the actual acceptable formats for COBOL statements.

> **Note:** In this manual, COBOL code examples use the FORMAT=TERMINAL setting. The online sample programs in the demo directory 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 the indicator area.

You can also place ANSI SQL-style comments starting with "-- " within SQL statements at the end of a line (but not after the last line of the SQL statement).

COBOL comments continue for the rest of the line after these two characters: "*>".

You can place C-style comments (/* ... */) in SQL statements.

The following example shows all four styles of comments:

```
     MOVE 12 TO DEPT-NUMBER. *> This is the software development group.
     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
```

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.
```

Copy Statements

Copy statements are not parsed by Pro*COBOL. Therefore, files included with the COPY command should not contain definitions of Host variables or contain embedded SQL statements. Instead, use the INCLUDE precompiler statement which is documented on "Using the INCLUDE Statement" on page 2-21. Be careful when using INCLUDE and also using DECLARE_SECTION=YES. Group items should be either placed all inside or all outside of a Declare Section.

Decimal-Point is Comma

Pro*COBOL supports the DECIMAL-POINT IS COMMA clause in the ENVIRONMENT DIVISION. If the DECIMAL-POINT IS COMMA clause appears in the source file, then the comma will be allowed as the symbol beginning the decimal part of any numeric literals in the VALUE clauses.

For example, the following is allowed:

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID.  FOO
    ENVIRONMENT DIVISION.
    CONFIGURATION   SECTION.
    SPECIAL-NAMES.
        DECIMAL-POINT IS COMMA.                *>  <-- **
    DATA DIVISION.
    WORKING-STORAGE SECTION.
...
    01  WDATA1          PIC      S9V999 VALUE  +,567. *>  <--- **
    01  WDATA2          PIC      S9V999 VALUE  -,234. *>  <--- **
...
```

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.

Division Headers that are Optional

The following division headers are optional:

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION
- DATA DIVISION

Note that the PROCEDURE DIVISION header is not optional. The following source can be precompiled:

```
*IDENTIFICATION DIVISION header is optional
 PROGRAM-ID.     HELLO.
*ENVIRONMENT DIVISION header is optional
 CONFIGURATION SECTION.
*DATA DIVISION header is optional
 WORKING-STORAGE SECTION.
 PROCEDURE       DIVISION.
     DISPLAY "Hello World!".
     STOP RUN.
```

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 *Oracle9i 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.

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).
```

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.

For SQL92 standards conformance, restrict the length of host variable names to 18 or fewer characters.

For a list of words that have restrictions on their use in applications, see Appendix C, "Reserved Words, Keywords, and Namespaces".

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 pseudo-type variables (these datatypes are prefixed with "SQL- ") 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.

Multibyte Datatypes

ANSI standard National Character Set datatypes are supported for handling multibyte character data. The PIC N or PIC G clause, if supported by your compiler, defines variables that store fixed-length NCHAR strings. You can store variable-length, multibyte National Character Set strings using COBOL group items consisting of a length field and a string field. See "VARCHAR Variables" on page 4-28.

The environmental variable NLS_NCHAR is available to specify a client-side Globalization Support National Character Set.

NULLs in SQL

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 and Section Names

You can associate standard COBOL paragraph and section 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 and section names in a WHENEVER ... DO or WHENEVER ... GOTO statement, as the next example shows:

```
PROCEDURE DIVISION.
MAIN.
    EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.
    ...
SQL-ERROR SECTION.
    ...
```

You must begin all paragraph names in area A.

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 2–2. Furthermore, COBOL enables the use of words instead of symbols, whereas SQL does not.

Table 2–2 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 following 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.
```

SQL statements may be ended by a comma, a period, or another COBOL statement.

The Declare Section

Passing data between the database server and your application program requires host variables and error handling. This section shows you how to meet these requirements.

Contents of a Declare Section

A *Declare Section*, 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

An Example

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

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

Precompiler Option DECLARE_SECTION

The Declare Section is optional. For backward compatibility with releases prior to 8.0, for which it was required, Pro*COBOL provides a command-line precompiler 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 is 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 14-5.

If DECLARE_SECTION is set to YES, then you must declare all program variables used in SQL statements inside the Declare Section. If DECLARE_SECTION is set to NO, then it is optional to use a Declare Section. In this case, declarations of host variables and indicator variables can be made either inside or outside a Declare Section. This optional behavior is a change from Release 8.0 and earlier releases. See "DECLARE_SECTION" on page 14-17 for details of the option.

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

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). 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.

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.

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 that contain them. 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.

Some compilers do not support the GLOBAL clause. Pro*COBOL supports nested programs by generating code that contains GLOBAL clauses. To avoid generating GLOBAL clauses unconditionally, specify the precompiler option NESTED=NO. NESTED (=YES or NO) defaults to YES and can be used in configuration files, or on the command line, but not inline (EXEC ORACLE statement).

See Also: "NESTED" on page 14-31.

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 enables 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 variables 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

For information on declaring the SQLCA for nested programs, (see "SQLCA Status Variable" on page 2-9), 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. This also applies to SQLDA and ORACA.

Nested Program Example

See SAMPLE13.PCO in the demo directory.

Conditional Precompilations

Conditional precompilation includes (or excludes) sections of code in your host program based on certain conditions. For example, you might want to include one section of code when precompiling under UNIX and another section when precompiling under VMS. Conditional precompilation lets you write programs that can run in different environments.

Conditional sections of code are marked by statements that define the environment and actions to take. You can code host-language statements as well as EXEC SQL statements in these sections. The following statements let you exercise conditional control over precompilation:

```
*   -- define a symbol
     EXEC ORACLE DEFINE symbol END-EXEC.
*    -- if symbol is defined
     EXEC ORACLE IFDEF symbol  END-EXEC.
*   -- if symbol is not defined
     EXEC ORACLE IFNDEF symbol END-EXEC.
*            -- otherwise
     EXEC ORACLE ELSE END-EXEC.
*           -- end this control block
     EXEC ORACLE ENDIF END-EXEC.
```

A conditional statement must be terminated with `END-EXEC`.

Note: The conditional compilation feature of your compiler may not be supported by Pro*COBOL.

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 Pro*COBOL is installed on your system. For example, predefined operating system symbols include CMS, MVS, UNIX, and VMS.

Separate Precompilations

You can precompile several COBOL program modules separately and 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 memory area for SQLCA, you must declare it globally. You can do this by modifying the SQLCA.COB file, changing the line

```
01  SQLCA.
```

to

```
01  SQLCA EXTERNAL.
```

Alternatively, you can include a hard-coded definition for SQLCA, copied from SQLCA.cob and make the aforementioned change. Note that you still have to include a definition of SQLCA in each precompiled unit.

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.

The linker resolves symbolic references in the object modules. If these references conflict, the link fails. Such conflicts 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 Support Services 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 manual.

Sample DEPT and EMP Tables

Most of the complete program examples in this guide use two sample database tables: DEPT and EMP. If they do not exist in your demo directory, create them before running the sample programs. 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 DEPT and EMP 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
```

Sample EMP Program: SAMPLE1.PCO

A good way to get acquainted with embedded SQL is to look at a program example. This program is SAMPLE1.PCO in the `demo` directory.

The 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 COMMISSION. The program uses the host indicator variable, COMM-IND to detect NULL values in column COMMISSION. See "Indicator Variables" on page 4-25.

The paragraph DISPLAY-INFO then displays the result.

The COBOL variables USERNAME, PASSWD, and EMP-NUMBER are declared using the VARYING clause, which enables you to use a variable-length string external Oracle datatype called VARCHAR. This datatype is explained in "VARCHAR Variables" on page 4-28.

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 8-6.

The BEGIN DECLARE SECTION and END DECLARE SECTION statements used are optional, unless you set the precompiler option DECLARE_SECTION to YES, or option MODE to ANSI. See "MODE" on page 2-2.

The WHENEVER statement is used to handle errors. For more details, see "WHENEVER Directive" on page 8-15.

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

```
*****************************************************************
* Sample Program 1:  Simple Query                               *
*                                                               *
* This program logs on to ORACLE, prompts the user for an       *
* employee number, queries the database for the employee's      *
* name, salary, and commission, then displays the result.       *
* The program terminates when the user enters a 0.              *
*****************************************************************

 ID 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.
     05  D-EMP-NUMBER  PIC 9(4).

 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 D-EMP-NUMBER.

    MOVE D-EMP-NUMBER TO 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, NVL(COMM, 0)
        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

Database Concepts

This chapter explains the CONNECT statement and its options, Oracle Net, and related network connection statements. Transaction processing is presented. You learn the basic techniques that safeguard the consistency of your database, including how to control whether changes to Oracle data are made permanent or undone.

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 (/) followed by an optional tnsnames.ora alias (@TNSALIAS).

The syntax for the CONNECT statement has an optional ALTER AUTHORIZATION clause. The complete 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  | IN {SYSDBA | SYSOPER} MODE} ]
END-EXEC.
```

The ALTER AUTHORIZATION clause is explained in "Changing Passwords at Runtime" on page 3-10. The SYSDBA and SYSOPER options are explained in "SYSDBA or SYSOPER Privileges" on page 3-11.

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 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.
    ...
```

```
01  USERNAME  PIC X(10).
01  PASSWD    PIC X(10).
        ...
    ...
PROCEDURE DIVISION.
LOGON.
    EXEC SQL WHENEVER SQLERROR GOTO LOGON-ERROR END-EXEC.
    MOVE "SCOTT" TO USERNAME.
    MOVE "TIGER" TO PASSWD.
    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.
```

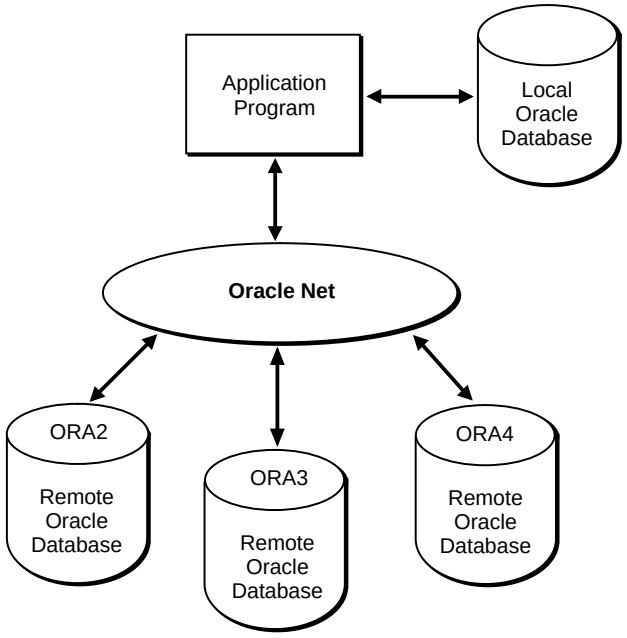
Default Databases and Connections

It is possible within a Pro*COBOL program to maintain more than one database connection at the same time.

Concurrent Logons

Pro*COBOL supports distributed processing through Oracle Net. 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 Oracle9*i* databases. ORA2, ORA3, and ORA4 are simply logical names used in CONNECT statements.

Figure 3–1 Connecting Through Oracle Net



By eliminating the boundaries in a network between different machines and operating systems, Oracle Net provides a distributed processing environment for Oracle tools. This section shows you how the Pro*COBOL supports distributed processing through Oracle Net. 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

Normally you would need only a single connection, achieved by `EXEC SQL CONNECT :USR-PWD END-EXEC`. The database that is connected to is determined by what USR-PWD contains. If it contains "SCOTT/TIGER", it will connect to the database defined as the default for the session and if it contains "SCOTT/TIGER@REMDB" it will connect through Oracle Net to the REMDB database as defined by your Oracle Net configuration. (An alternative is to use the USING clause to specify the Oracle Net connection string.) This is the default connection.

To make further concurrent connections to either the same or different databases you make use of the AT clause, that is, `EXEC SQL AT DB1 CONNECT :USR-PWD END-EXEC`. The name after the AT clause uniquely identifies a "nondefault" connection, and any SQL statements with the same name after the AT clause are executed against that connection. If the AT clause is omitted in an SQL statement then the statement is executed against the 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.

Using Username/Password

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 any valid Oracle connect string.

You can also log on automatically, as shown in "Automatic Logons" on page 3-9.

These are simplified subsets of the CONNECT statement. For all details, read the next sections in this chapter and also see "CONNECT (Executable Embedded SQL Extension)" on page F-17.

Named Database Connections

In the following example, you connect to a named database. Normally you use a named database connection only for multiple concurrent connections. The following example shows the syntax for a single connection:

```
* --  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 Oracle Net syntax for logging on to a nondefault database at a remote node.
- The undeclared identifier *DBNAME* names a nondefault connection; it is an identifier used by Oracle, *not* a host or program variable.

The USING clause specifies the network, machine, and database to be associated with *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 nondefault 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 ...
```

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 enables 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 nondefault 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 nondefault 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.
```

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$) and *username* is your operating system user or task name. For example, if the prefix is OPS$, your user name is TBARNES, and OPS$TBARNES is a valid Oracle userid, then you log on to Oracle as user OPS$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.

Changing Passwords at Runtime

Pro*COBOL provides client applications with a convenient way to change a user password at runtime through the optional ALTER AUTHORIZATION clause.

The syntax for the ALTER AUTHORIZATION clause is shown here:

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

Using this clause indicates that you want 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:

- The application will connect without issue.
- The application will fail to connect. This could be due to either of the following:
  - Password verification failed for some reason. In this case the password remains unchanged.
  - The account is locked. Changes to the password are not permitted.

Connect Without Alter Authorization

This section describes the possible outcomes of different variations of the 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:

- The application will connect without issue.
- 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 logons. At this point, the user is encouraged to change the password before the account becomes locked.
- 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.

SYSDBA or SYSOPER Privileges

Before Oracle release 8.1 you did not have to use this clause to have the SYSOPER or SYSDBA system privilege, but now you must.

Append the following optional string to the CONNECT statement after all other clauses if you want to log on with either SYSDBA or SYSOPER system privileges:

```
IN { SYSDBA | SYSOPER } MODE
```

For example:

```
EXEC SQL CONNECT ... IN SYSDBA MODE END-EXEC.
```

Here are the restrictions that apply to this option:

- This option is not supported when using the AUTO_CONNECT=YES precompiler option setting.
- The option is not permitted when using the ALTER AUTHORIZATION keywords in the CONNECT statement.

Using Links

Database links are supported through the Oracle9*i* distributed database option. For example, a distributed query allows a single SELECT statement to access data on one or more nondefault databases.

The distributed query facility depends on database links, which assign a name to a CONNECT statement rather than to the connection itself. At runtime, the embedded SELECT statement is executed by the specified database server, which connects implicitly to the nondefault database(s) to get the required data.

For more information, see *Oracle Net Services Administrator's Guide.*

Key Terms

Before delving into the subject of transactions, you should know the terms defined in this section.

The jobs or tasks that the database 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 the database. Oracle9*i* enables user sessions to work simultaneously and share computer resources. To do this, Oracle9*i* 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.

Oracle9*i* 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 table 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 the server breaks the deadlock. The server signals an error to the participating transaction that had completed the least amount of work, and the "deadlock detected while waiting for resource" error code is returned to SQLCODE in the SQLCA.

When a table is queried by one user and updated by another at the same time, the database generates a **read consistent** view of the table's data for the query. That is, once a query begins (and proceeds), the data read by the query does not change. As update activity continues, the database takes **snapshots** of the table's data and records changes in a **rollback segment**. The database uses information in the rollback segment to build read consistent query results and to undo changes if necessary.

How Transactions Guard a Database

The database is transaction oriented; 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. The database 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, it shows 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 *Oracle9i SQL Reference*.

Beginning and Ending Transactions

You begin a transaction with the first executable SQL statement (other than CONNECT) in your program. When one transaction ends, the next executable SQL statement automatically begins another transaction. Thus, every executable statement is part of a transaction. Because they cannot be rolled back and need not be committed, declarative SQL statements are not considered part of a transaction.

You end a transaction in one of the following ways:

- Code a COMMIT or ROLLBACK statement, with or without the RELEASE option. This *explicitly* makes permanent or undoes changes to the database.
- Code a data definition statement (ALTER, CREATE, or GRANT, for example) 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 interrupt.

If your program fails in the middle of a transaction, Oracle9*i* detects the error and rolls back the transaction. If your operating system fails, Oracle9*i* restores the database to its former (pre-transaction) state.

Using the COMMIT Statement

The COMMIT statement is used 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 declared using the FOR UPDATE clause or referenced elsewhere in the code with the CURRENT OF clause. If MODE=ANSI | ANSI14 or CLOSE_ON_COMMIT=YES is used, then all explicit cursors are closed.
- 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 3-22.

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, Oracle9*i* rolls them back. In the following example, you commit your transaction and disconnect:

```
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all 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

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 Oracle. When MODE=ANSI, Oracle 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 F-24.

CLOSE_ON_COMMIT Precompiler Option

The precompiler option CLOSE_ON_COMMIT is available to override the default behavior of MODE=ANSI (if you specify MODE=ANSI on the command line, any cursors not declared with the WITH HOLD clause are closed on commit):

```
CLOSE_ON_COMMIT = {YES | NO}
```

The default is NO. This option must be entered only on the command line or in a configuration file.

> **Note:** Use this option carefully; applications may be slowed if cursors are opened and closed many times because of the need to re-parse for each OPEN statement. See "CLOSE_ON_COMMIT" on page 14-14.

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 declared using the FOR UPDATE clause or referenced elsewhere in the code with the CURRENT OF clause. If MODE={ANSI | ANSI14}, then *all* explicit cursors are closed.

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:

```
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.
DISPLAY 'PROCESSING ERROR.'.
DISPLAY 'ERROR CODE : ', SQLCODE.
DISPLAY 'MESSAGE :', SQLERRMC.
EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
STOP RUN.
```

Oracle9*i* rolls back transactions if your program terminates abnormally.

Statement-Level Rollbacks

Before executing any SQL statement, Oracle9*i* marks an implicit savepoint (not available to you). Then, if the statement fails, Oracle9*i* 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, Oracle9*i* 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.

Oracle9*i* can also roll back single SQL statements to break deadlocks. Oracle9*i* signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Using the SAVEPOINT Statement

The SAVEPOINT embedded SQL statement marks and names the current point in processing 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 data to its former state, recover, and then reexecute the function.

To undo part of a transaction, you can 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. With it, 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 SAVEPOINT 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.

Using the RELEASE Option

Oracle9*i* 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 using the RELEASE embedded SQL statement.

Normal termination occurs when your program runs its course, closes open cursors, explicitly commits or rolls back work, disconnects, and returns control to the user. Your program will exit gracefully if the last SQL statement it executes is either

```
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

or

```
EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
```

where the token WORK is optional. Otherwise, locks and cursors acquired by your user session are held after program termination until Oracle9*i* recognizes that the user session is no longer active. This might cause other users in a multiuser environment to wait longer than necessary for the locked resources.

Using the SET TRANSACTION Statement

You can 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 multitable, multiquery, 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, a store manager checks 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, Oracle9*i* implicitly (automatically) locks many data structures for you. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking. Explicit locking lets you share or deny access to a table for the duration of a transaction or ensure multitable and multiquery read consistency.

With the SELECT FOR UPDATE OF statement, you can explicitly lock specific rows of a table to make sure they do not change before an update or delete is executed. However, Oracle9*i* 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, you can meanwhile optionally specify the FOR UPDATE clause, which has the effect of acquiring an exclusive lock on all rows defined by the cursor. This is useful, for example, when you want to base an update on existing rows in a table and want to ensure that they are not meanwhile changed by anyone else.

Note that if you refer to a cursor with the CURRENT OF clause, that the precompiler will automatically add the FOR UPDATE clause to the cursor definition and the word OF 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 OF part of the clause and simply code:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20
       FOR UPDATE
END-EXEC.
```

For an example, see "Using the CURRENT OF Clause" on page 5-16.

Restrictions

You cannot use FOR UPDATE with multiple tables, but you must use FOR UPDATE OF to identify a column in the table that you want locked. Row locks obtained by a FOR UPDATE statement are cleared by a COMMIT, which explains why the cursor is closed for you. If you try to fetch from a FOR UPDATE cursor after a commit, Oracle9*i* generates a Fetch out of Sequence error.

Fetching Across Commits

If you want to mix commits and fetches, do not use the CURRENT OF clause. Instead, select the ROWID of each row, and 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 can receive inconsistent results if another user modifies a row after you read it but before you update or delete it.

Using the LOCK TABLE Statement

Use the LOCK TABLE statement locks 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 *Oracle9i Application Developer's Guide - Fundamentals*

The optional keyword NOWAIT tells Oracle9*i* not to wait for a table if it has been locked by another user. Control is immediately returned to your program so that 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, Oracle9*i* 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. Consequently, 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.

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, the server 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 *Oracle9i Database Concepts.*

Guidelines for Transaction Processing

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. Thus, 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 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 rollback operations inside the block affect the whole transaction. In the following example, the rollback operation 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.
...
```

X/Open Applications

For instructions on using the XA interface in X/Open applications, see your Transaction Processing (TP) Monitor user's guide and *Oracle9i Application Developer's Guide - Fundamentals.*

4

Datatypes and Host Variables

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

The Oracle9i Datatypes

Oracle9*i* recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle9*i* stores data in database columns.

For complete descriptions of the Oracle internal (also called *built-in*) datatypes, see *Oracle9i SQL Reference*.

Oracle9*i* also uses internal datatypes to represent database pseudocolumns. An external datatype specifies how data is stored in a host variable.

Internal Datatypes

Table 4–1 summarizes the information about each Oracle built-in datatype.

Table 4–1 Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length and Default
CHAR (size)	Fixed-length character data of length *size* in characters or bytes, depending on the national character set	Fixed for every row in the table (with trailing blanks.) Column size is the number of characters for a fixed-width national character set or the number of bytes required to store one character, with an upper limit of 2000 bytes for each row. Default size is 1 character or 1 byte for each row, depending on the national character set. Consider the character set (one-byte or multibyte) before setting *size*.
VARCHAR2 (size)	Fixed-length character data of length *size* in characters or bytes, depending on the national character set. A maximum size must be specified.	Variable for each row. Column size is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum size is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes for each row. Default size is 1 character or 1 byte, depending on the national character set.

Table 4–1 Summary of Oracle Built-In Datatypes (Cont.)

NCHAR (size)	Fixed-length character data of length *size* in characters or bytes, depending on national character set.	Fixed for every row in the table (with trailing blanks). Column *size* is the number of bytes for a national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 2000 bytes for each row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 (size)	Variable-length character data of length *size* in characters or bytes, depending on national character set. A maximum *size* must be specified.	Variable for each row. Column *size* is the number of bytes for a national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes for each row. Default is 1 character or 1 byte, depending on the character set.
CLOB	Single-byte character data	Up to 2^32 - 1 bytes, or 4 gigabytes.
NCLOB	Single-byte or fixed-length multibyte national character set (NCHAR) data	Up to 2^32 - 1 bytes, or 4 gigabytes.
LONG	Variable-length character data	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, for each row. Provided for backward compatibility.
NUMBER(p,s)	Variable-length numeric data.: Maximum precision *p* and / or scale *s* is 38	Variable for each row. The maximum space required for a given column is 21 bytes for each row.
DATE	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E.	Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-YY) specified by NLS_DATE_FORMAT parameter.
BLOB	Unstructured binary data	Up to 2^32 - 1 bytes, or 4 gigabytes.

Table 4–1 Summary of Oracle Built-In Datatypes (Cont.)

BFILE	Binary data stored in an external file	Up to 2^32 - 1 bytes, or 4 gigabytes.
RAW (size)	Variable-length raw binary data	Variable for each row in the table, up to 2000 bytes for each row. A maximum *size* must be specified. Provided for backward compatibility.
LONG RAW	Variable-length raw binary data	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, for each row. Provided for backward compatibility.
ROWID	Binary data representing row addresses	Fixed at 10 bytes (extended `ROWID`) or 6 bytes (restricted `ROWID`) for each row in the table.

External Datatypes

The external datatypes include all the internal datatypes plus several datatypes found in other supported host languages. Use the datatype names in datatype equivalencing, and the datatype codes in dynamic SQL Method 4. The following table lists external datatypes.

Table 4–2 External Datatypes

Name	Code	Description
CHAR	1	<= 65535-byte, variable-length character string ()
	96	<= 65535-byte, fixed-length character string ()
CHARF	96	<= 65535-byte, fixed-length character string
CHARZ	97	<= 65535-byte, fixed-length, null-terminated string ()
DATE	12	7-byte, fixed-length date/time value
DECIMAL	7	COBOL packed decimal
DISPLAY	91	COBOL numeric character string with leading sign
DISPLAY TRAILING	152	COBOL numeric with trailing sign
FLOAT	4	4-byte or 8-byte floating-point number
INTEGER	3	2-byte, 4-byte, or 8-byte signed integer. (8-byte on 64-bit platforms).
LONG	8	<= 2147483647-byte, fixed-length string
LONG RAW	24	<= 217483647-byte, fixed-length binary data

Table 4–2 External Datatypes

Name	Code	Description
LONG VARCHAR	94	<= 217483643-byte, variable-length string
LONG VARRAW	95	<= 217483643-byte, variable-length binary data
NUMBER	2	Internal Oracle Format Number represented in binary coded decimal format.
OVERPUNCH LEADING	172	COBOL numeric character string with embedded leading sign
OVERPUNCH TRAILING	154	COBOL numeric character string with embedded trailing sign (equivalent to declarations of the form PIC S9(n)V9(m) DISPLAY)
RAW	23	<= 65535-byte, fixed-length binary data ()
ROWID	11	fixed-length binary value (system-specific)
STRING	5	<= 65535-byte, null-terminated character string ()
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 ()
VARNUM	6	variable-length binary number
VARRAW	15	<= 65533-byte, variable-length binary data

Notes:

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

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

CHAR

CHAR behavior depends on the settings of the option PICX. See "PICX" on page 14-34.

CHARF

By default, the CHARF datatype represents 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 14-34.

On Input. Oracle9*i* 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, Oracle9*i* generates an error. If the input value is all-blank, then a string of spaces is stored.

On Output. Oracle9*i* 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, then the original value of the variable is not overwritten.

If the output value is longer than the declared length of the host variable, Oracle9*i* truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle9*i* sets it to the original length of the output value.

CHARZ

The CHARZ datatype represents fixed-length, null-terminated character strings. On most platforms, the maximum length of a CHARZ value is 65535 bytes. You usually will not need this external type in Pro*COBOL.

DATE

The DATE datatype represents dates and times in 7-byte, fixed-length fields. As Table 4–3 shows, the century, year, month, day, hour (in 24-hour format), minute, and second are stored in that order from left to right.

Table 4–3 Date Format

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

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). Pro*COBOL also

supports five additional datetime datetypes, as described in "Datetime and Interval Datatype Descriptors" on page 4-13.

DECIMAL

The DECIMAL datatype represents 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, the value is truncated to the declared length.

DISPLAY

The DISPLAY datatype represents 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

The FLOAT datatype represents numbers that have a fractional part or that exceed the capacity of the INTEGER datatype. FLOAT relates to the COBOL datatypes COMP-1 (4-byte floating point) and COMP-2 (8-byte floating point).

Oracle9*i* can represent numbers with greater precision than floating point implementations because the internal format of Oracle9*i* 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

The INTEGER datatype represents numbers that have no fractional part. An integer is a signed, 2-byte, 4-byte, or 8-byte binary number. (8-byte on 64-bit platforms.) The order of the bytes in a word is platform-dependent. You must specify a length for input and output host variables. On output, if the column has a fractional part, the digits after the decimal point are truncated.

LONG

The LONG datatype represents 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

The LONG RAW datatype represents 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 Oracle9*i* 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

The LONG VARCHAR datatype represents 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

The LONG VARRAW datatype represents 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.

NUMBER

The NUMBER datatype represents the internal Oracle NUMBER format which cannot be represented by a COBOL datatype.

OVER-PUNCH

OVER-PUNCH is the default signed numeric for the COBOL language. Digits are held in ASCII or EBCDIC format in radix 10, with one digit for each byte of computer storage. The sign is held in the high order nibble of one of the bytes. It is called OVER-PUNCH 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 over-punch.

RAW

The RAW datatype represents 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 Oracle9*i* 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

The ROWID datatype is the database row identifier in COBOL. To support both logical and physical ROWIDs (as well as ROWIDs of non-Oracle tables) the Universal ROWID (UROWID) was defined. Use the SQL-ROWID pseudotype for this datatype (see "Universal ROWIDs" on page 4-34).

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, Oracle9*i* 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, and hen 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 7-19.

> **Note:** If you need full portability or your application communicates with a non-Oracle database through 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 through 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 terminated by a LOW-VALUE character.This datatype is usually not used in Pro*COBOL.

UNSIGNED

The UNSIGNED datatype represents unsigned integers.This datatype is usually not used in Pro*COBOL.

VARCHAR

The VARCHAR datatype represents 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

The VARCHAR2 datatype represents 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 multibyte characters, its maximum length is less than *n* characters.

On Input. Oracle9*i* reads the number of bytes specified for the input host variable, strips any trailing blanks, and then stores the input value in the target database column.

If the input value is longer than the defined width of the database column, Oracle9*i* generates an error. If the input value is all SPACES, Oracle9*i* treats it like a NULL.

Oracle9*i* can convert a character value to a NUMBER column value if the character value represents a valid number. Otherwise, Oracle9*i* generates an error.

On Output. Oracle9*i* returns the number of bytes specified for the output host variable, blank-padding if necessary, and then assigns the output value to the target host variable. If a NULL is returned, Oracle9*i* fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, Oracle9*i* truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle9*i* sets it to the original length of the output value.

Oracle9*i* can convert NUMBER column values to character values. The length of the character host variable determines precision. If the host variable is too short for the number, scientific notation is used. For example, if you select the column value 123456789 into a host variable of length 6, Oracle9*i* returns the value `1.2E08` to the host variable.

VARNUM

The VARNUM datatype is similar in format to NUMBER and is usually not used in Pro*COBOL.

VARRAW

The VARRAW datatype represents 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.

SQL Pseudocolumns and Functions

SQL recognizes the pseudocolumns listed in Table 4–4, which return specific data items.

Table 4–4 Pseudocolumns and Internal Datatypes

Pseudocolumn	Internal Datatype
CURRVAL	NUMBER
LEVEL	NUMBER
NEXTVAL	NUMBER
ROWID	ROWID
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 pseudocolumn values from a dummy table.

In addition, SQL recognizes the functions without parameters listed in Table 4–5, which also return specific data items.

Table 4–5 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 pseudocolumns and functions follow. For details, see the *Oracle9i 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.

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 can 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, the sequence named *partno* assigns 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 roll back the transaction. A reference to NEXTVAL stores the current sequence number in CURRVAL.

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.

Datetime and Interval Datatype Descriptors

The OCI datetime and interval datatypes supported by Pro*COBOL are briefly summarized here.

> **See Also:** *Oracle9i SQL Reference f*or more a more complete discussion datetime datatype descriptors

ANSI DATE

The ANSI DATE is based on the DATE, but contains no time portion. (Therefore, it also has no time zone.) ANSI DATE follows the ANSI specification for the DATE datatype. When assigning an ANSI DATE to a DATE or a timestamp datatype, the time portion of the Oracle DATE and the timestamp are set to zero. When assigning a DATE or a timestamp to an ANSI DATE, the time portion is ignored.

You are encouraged to instead use the TIMESTAMP datatype which contains both date and time.

TIMESTAMP

The TIMESTAMP datatype is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, plus the hour, minute, and second values. It has no time zone. The TIMESTAMP datatype has the form:

```
TIMESTAMP(fractional_seconds_precision)
```

where *fractional_seconds_precision* (which is optional) specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE (TSTZ) is a variant of TIMESTAMP that includes an explicit time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The TIMESTAMP WITH TIME ZONE datatype has the form:

```
TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE
```

where *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data.

TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ) is another variant of TIMESTAMP that includes a time zone displacement in its value. Storage is in the same format as for TIMESTAMP. This type differs from TIMESTAMP WITH TIME ZONE in that data stored in the database is normalized to the database time zone, and the time zone displacement is not stored as part of the column data. When users retrieve the data, Oracle returns it in the users' local session time zone.

The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The TIMESTAMP WITH LOCAL TIME ZONE datatype has the form:

```
TIMESTAMP(fractional_seconds_precision) WITH LOCAL TIME ZONE
```

where *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. The INTERVAL YEAR TO MONTH datatype has the form:

```
INTERVAL YEAR(year_precision) TO MONTH
```

where the optional *year_precision* is the number of digits in the YEAR datetime field. The default value of *year_precision* is 2.

INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds. The INTERVAL DAY TO SECOND datatype has the form:

```
INTERVAL DAY (day_precision) TO SECOND(fractional_seconds_precision)
```

where:

- *day_precision* is the number of digits in the DAY datetime field. It is optional. Accepted values are 0 to 9. The default is 2.

fractional_seconds_precision is the number of digits in the fractional part of the SECOND datetime field. It is optional. Accepted values are 0 to 9. The default is 6.

> **Note:** To avoid unexpected results in your DML operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and SESSIONTIMEZONE. If the time zones have not been set manually, Oracle uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, Oracle uses UTC as the default value.

Host Variables

Host variables are the key to communication between your host program and the server. Typically, a host program inputs data to the server, and the server outputs data to the program. The server 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 Pro*COBOL. COBOL datatypes must be compatible with the source / target database column.

The supported COBOL variable declarations, descriptions, corresponding external datatypes, and Oracle datatype codes are shown in Table 4–6.

Table 4–6 Host Variable Declarations

Variable Declaration	Description	External Datatype	Type Code
PIC X...X	fixed-length string of 1-byte characters (1)	CHARF	96
PIC X(*n*)	*n*-length string of 1-byte characters		
PIC X...X VARYING	variable-length string of 1-byte characters (1,2)	VARCHAR	9
PIC X(*n*) VARYING	variable-length (*n* max.) string of 1-byte characters (2)		
PIC N...N	fixed-length string of multibyte NCHAR	CHARF	96
PIC G...G	characters (1,3)		
PIC N(*n*)	*n*-length string of multibyte NCHAR characters		
PIC G(n)	(3)		
PIC N...N VARYING	variable-length string of multibyte characters (2,3)	VARCHAR	9
PIC N(n) VARYING			
PIC G...G VARYING	variable-length (*n* max.) string of multibyte		
PIC G(n) VARYING	characters (2,3)		
PIC S9...9 BINARY	integer (4,5,7)	INTEGER	3
PIC S9(*n*) BINARY			
PIC S9...9 COMP			
PIC S9(*n*) COMP			
PIC S9...9 COMP-4			
PIC S9(*n*) COMP-4			
PIC S9...9 COMP-5	byte-swapped integer (4,5,6,7)	INTEGER	3
PIC S9(*n*) COMP-5			
COMP-1	floating-point number (5)	FLOAT	4
COMP-2			
PIC S9...9[V9...9] COMP-3	packed-decimal (4,5)	DECIMAL	7
PIC S9(*n*)[V9(*n*)] COMP-3			
PIC S9...9[V9...9] PACKED-DECIMAL			
PIC S9(*n*)[V9(*n*)] PACKED-DECIMAL			

Table 4–6 Host Variable Declarations

Variable Declaration	Description	External Datatype	Type Code
PIC S9...9[V9...9] DISPLAY SIGN LEADING SEPARATE PIC S9(*n*)[V9(*m*)] DISPLAY SIGN LEADING SEPARATE	display leading (8,11)	DISPLAY	91
PIC S9...9[V9...9] DISPLAY SIGN TRAILING SEPARATE PIC S9(*n*)[V9(*m*)] DISPLAY SIGN TRAILING SEPARATE	display trailing (8)	DISPLAY TRAILING	152
PIC 9...9 DISPLAY PIC 9(n)[V9(m)] DISPLAY	unsigned display(9)	UNSIGNED DISPLAY	153
PIC S9...9[V9...9] DISPLAY SIGN TRAILING PIC S9(n)[V9(m)] DISPLAY SIGN TRAILING	over-punch trailing (9)	OVER-PUNCH TRAILING	154
PIC S9...9[V9...9] DISPLAY SIGN LEADING PIC S9(n)[V9(m)] DISPLAY SIGN LEADING	over-punch leading (9)	OVER-PUNCH LEADING	172
SQL-CURSOR	cursor variable		
SQL-CONTEXT	runtime context		
SQL-ROWID	universal ROWID	UROWID	104
SQL-BFILE	external binary file	BFILE	112
SQL-BLOB	binary LOB	BLOB	113
SQL-CLOB	character LOB	CLOB	114

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 4-28.
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 the integer is *n* to 18. Typically it is 9 on 32-bit machines and 16 on 64-bit machines. This may vary, depending upon your system.
8. Both DISPLAY and SIGN are optional.
9. DISPLAY is optional
10. If TRAILING is omitted, the embedded sign position is operating-system dependent.
11. LEADING is optional.

In Table 4–6 and Table 4–7 the symbols '[' and ']' denote that an optional entry is contained inside. The symbols '{' and '}' denote that a choice must be made between tokens separated by the symbol ' | '.

Table 4–7, "Compatible Oracle Internal Datatypes" shows all the COBOL datatypes that can be converted to and from each internal datatype.

Table 4–7 Compatible Oracle Internal Datatypes

Internal Datatype	Notes	COBOL Datatype	Description
CHAR(*x*)	(1)	PIC X(n)	character string
VARCHAR2(*y*)	(1)	PIC X...X	*n*-character string
		PIC X(*n*) VARYING	variable-length string
		PIC X...X VARYING	
		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...9[V9...9] COMP-3	packed decimal
		PIC S9(*n*)[V9(*n*)] COMP-3	
		PIC S9...9[V9...9] DISPLAY	display
		PIC S9(*n*)[V9(*n*)] DISPLAY	
NCHAR(u)	(2)	PIC {N...N | G...G}	national character string
NVARCHAR2(v)	{2}	PIC { N(n) | G(n)}	n-national character string
BLOB		SQL-BLOB	binary LOB
CLOB		SQL-CLOB	character LOB
NCLOB		SQL-NCLOB	national character LOB
BFILE		SQL-BFILE	external binary file

Table 4–7 Compatible Oracle Internal Datatypes

Internal Datatype	Notes	COBOL Datatype	Description
NUMBER		PIC S9...9 COMP	integer
NUMBER (*p*,*s*)	(3)	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 (4)
		PIC [X(*n*) | N(*n*) | G(n)]	*n*-character string (4)
		PIC X...X VARYING	variable-length string
		PIC X(*n*) VARYING	*n*-byte variable-length string
DATE	(5)	PIC X(*n*)	*n*-byte character string
LONG			
RAW	(1)	PIC X...X	character string
		PIC X(n)	
		PIC X(n) VARYING	*n*-byte variable-length string
		PIC X...X VARYING	
LONG RAW			
ROWID	(6)	SQL-ROWID	universal rowid

Notes:

1. <= *x* < =2000 bytes, default is 1. 1<=*y* <=4000 bytes, default is 1.

2. 1<=u<=2000 bytes, default is 1. 1<=v<=4000 bytes, default is 1.

3. *p* ranges from 2 to 38. *s* ranges from -84 to 127.
4. Strings can be converted to NUMBERs only if they consist of convertible characters — 0 to 9, period (.), +, -, E, e. The Globalization Support (formerly called National Language Support or NLS) settings for your system might change the decimal point from a period (.) to a comma (,).
5. 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.
6. When converted to a string type, a ROWID requires from 18 to 4000 bytes. ROWID can also be converted to a character type. Oracle recommends the use of SQL-ROWID for all new programs.

Example Declarations

The following example declares several host variables for later use:

```
    ...
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

You can initialize host variables, except pseudotype 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.

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

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

Host variables are used 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 a 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 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 :DEPARTURE.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 "VARCHAR" on page 14-42.

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.
  ...
     EXEC SQL SELECT DHR, DMIN INTO :DEPARTURE.HOUR, :DEPARTURE.MINUTE
         FROM TIMETABLE
         WHERE ...
```

Restrictions

A host variable cannot substitute for a column, table, or other object in a SQL statement and must not be an Oracle9*i* reserved word. See Appendix C, "Reserved Words, Keywords, and Namespaces"for a list of reserved words and keywords.

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 for character columns) in output host variables.

Using Indicator Variables

Here are the values indicator variables can take on.

On Input The values your program can assign to an indicator variable have the following meanings:

-1	Oracle will assign a NULL to the column, ignoring the value of the host variable.
>=0	Oracle will assign the value of the host variable to the column.

On Output T he values Oracle can assign to an indicator variable have the following meanings:

-1	The column value is NULL, so the value of the host variable is indeterminate.
0	Oracle assigned an intact column value to the host variable.
>0	Oracle assigned a truncated column value to the host variable. The integer returned by the indicator variable is the original length of the column value, and SQLCODE in SQLCA is set to zero.
-2	Oracle assigned a truncated column variable to the host variable, but the original column value could not be determined (a LONG column, for example).

Declaring Indicator Variables

An indicator variable must be explicitly declared as PIC S9(4) COMP and must not be a 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.

Use in Where Clauses

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.
```

Avoid Error Messages

If you SELECT or FETCH a NULL into a host variable that has no indicator, Oracle9*i* issues an error message.

You can disable the error message by also specifying UNSAFE_NULL=YES on the command line. For more information, see Chapter 14, "Precompiler Options".

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, Oracle9*i* issues an error message.

However, when MODE={ANSI | ANSI14 | ANSI13}, no error is generated. Values for indicator variables are discussed in Chapter 5, "Embedded SQL".

Indicator Variables for Multibyte NCHAR Variables

Indicator variables for multibyte 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 multibyte 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.

VARCHAR Variables

COBOL string datatypes are fixed length. However, Pro*COBOL lets you declare a variable-length string pseudotype called VARCHAR. A VARCHAR variable is a pseudotype that enables you to specify the exact length of the data stored in the database and to specify the exact length of the data to be passed to the database.

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.
```

Note: PIC N and PIC G are not allowed in definitions that use VARYING. To see how to correctly use PIC N and PIC G in VARCHAR variables, see "Implicit VARCHAR Group Items" on page 4-29

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 9,999 bytes.

The advantage of using VARCHAR variables is that you can explicitly set and reference the length field. With input host variables, Pro*COBOL reads the value of the length field and uses that many characters of the string field. With output host variables, Pro*COBOL 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 multibyte NCHAR characters, use these formats (length is expressed in double-byte characters):

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

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) 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 14, "Precompiler Options"

Referencing VARCHAR Variables

In SQL statements, you reference a VARCHAR variable using the group name prefixed 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.

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 multibyte Globalization Support (formerly called 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)

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

Default for PIC X

The default datatype of PIC X variables is CHARF (was VARCHAR2 before release 8.0.) The 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 14-34 for more details.

Effects of the PICX Option

The PICX option determines how Pro*COBOL treats data in character strings. The PICX option enables your program to use ANSI fixed-length strings or to maintain compatibility with previous versions of the database 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 VARCHAR2 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,

Pro*COBOL re-appends trailing blanks up to the length of the database column. However, if you insert into a variable-length VARCHAR2 column, Pro*COBOL never appends blanks.

When PICX=CHARF, trailing blanks are never stripped.

Host input variables for multibyte Globalization Support 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.

Make sure that the input value is not trailed by extraneous characters. 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, Pro*COBOL 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.

Varying-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, Pro*COBOL uses exactly the number of characters given by the length field, counting any spaces.

On Output

When you use a VARCHAR variable as an output host variable, Pro*COBOL 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 Pro*COBOL 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 multibyte NCHAR data are *not* padded at all. The length of the buffer is set to the length in characters, not bytes.

Universal ROWIDs

There are two kinds of table organization used in the database server: *heap tables* and i*ndex-organized tables*.

Heap tables are the default. This is the organization used in all tables before Oracle9. The physical row address (ROWID) is a permanent property that is used to identify a row in a heap table. The external character format of the physical ROWID is an 18-byte character string in base-64 encoding.

An index-organized table does not have physical row addresses as permanent identifiers. A *logical* ROWID is defined for these tables. When you use a SELECT ROWID ... statement from an index-organized table the ROWID is an opaque structure that contains the primary key of the table, control information, and an optional physical "guess". You can use this ROWID in a SQL statement containing a clause such as "WHERE ROWID = ..." to retrieve values from the table.

The *universal* ROWID was introduced in the Oracle 8.1 release. Universal ROWID can be used for both physical ROWID and logical ROWID. You can use universal ROWIDs to access data in heap tables, or index-organized tables, since the table organization can change with no effect on applications. The column datatype used for ROWID is UROWID(*length*), where `length` is optional.

Use the universal ROWID in all new applications.

For more information on universal ROWIDs, see *Oracle9i Database Concepts*.

Declare a universal ROWID, which uses the pseudotype SQL-ROWID, this way:

```
 01  MY-ROWID SQL-ROWID.
```

Memory for the universal ROWID is allocated with the ALLOCATE statement:

```
     EXEC SQL ALLOCATE :MY-ROWID END-EXEC.
```

Use MY-ROWID in SQL DML statements like this:

```
     EXEC SQL SELECT ROWID INTO :MY-ROWID FROM MYTABLE WHERE ... END-EXEC.
...
     EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID END-EXEC.
...
```

Free the memory when you no longer need it with the FREE directive:

```
     EXEC SQL FREE :MY-ROWID END-EXEC.
```

You also have the option of using a character host variable of width between 18 and 4000 as the host bind variable for universal ROWID. Character-based universal ROWIDs are supported for heap tables only for backwards compatibility. Because a universal ROWID can be variable length there can be truncation when it is selected. For a more complete discussion of this variable see *Oracle9i Database Concepts*.

Use the character variable like this:

```
 01  MY-ROWID-CHAR PIC X(4000) VARYING.
...
     EXEC SQL ALLOCATE :MY-ROWID-CHAR END-EXEC.
     EXEC SQL SELECT ROWID INTO :MY-ROWID-CHAR FROM MYTABLE WHERE ... END-EXEC.
...
     EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID-CHAR END-EXEC.
...
     EXEC SQL FREE :MY-ROWID-CHAR END-EXEC.
```

For an example of a *positioned update* using the universal ROWID, see "Positioned Update" on page 5-18.

Subprogram SQLROWIDGET

The Oracle subprogram SQLROWIDGET enables you to retrieve the ROWID of the last row inserted, updated, or selected. SQLROWIDGET requires a context or NULL and a ROWID as its arguments.

To use the default context, pass the figurative constant NULL as the first parameter in the call to SQLROWIDGET.

Note: The universal ROWID must be declared and allocated before the call. The context, if used must be declared and allocated before the call. Here is the syntax:

```
CALL "SQLROWIDGET" USING NULL rowid.
```

or

```
CALL "SQLROWIDGET" USING context rowid.
```

where

context (IN)

is the runtime context variable, of pseudotype SQL-CONTEXT, or the figurative constant NULL for the default context. For a discussion of runtime contexts, see "Embedded SQL Statements and Directives for Runtime Contexts" on page 12-8.

rowid (OUT)

is a universal ROWID variable, of pseudotype SQL-ROWID. When a normal execution finishes, this will point to a valid universal ROWID. In case of an error, *rowid* is undefined.

Here is a sample showing its use with the default context:

```
 01  MY-ROWID SQL-ROWID.
 ...
     EXEC SQL ALLOCATE :MY-ROWID END-EXEC.

* INSERT, or UPDATE or DELETE Goes here:
 ...
     CALL "SQLROWIDGET" USING NULL MY-ROWID.
* MY-ROWID now has the universal rowid descriptor for the last row
 ...
     EXEC SQL FREE :MY-ROWID END-EXEC.
 ...
```

If your compiler does not allow using the figurative constant NULL in a CALL statement, you can declare a variable with picture S9(9) COMP VALUE 0 and use that with the BY VALUE clause in the call to SQLROWIDGET, as follows:

```
 01 NULL-CONTEXT  PIC S9(9) COMP VALUE ZERO.
 01 MY-ROWID SQLROWID.
....
        CALL "SQLROWIDGET" USING BY VALUE NULL-CONTEXT BY REFERENCE MY-ROWID.
```

Globalization Support

Although the widely-used 7-bit 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 Oracle9*i* deal with such dissimilar languages?

Oracle9*i* provides Globalization Support (formerly called National Language Support or NLS), which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments. With Globalization Support, number and date formats adapt automatically to the language conventions specified for a user session. Thus, Globalization Support enables users around the world to interact with Oracle9*i* in their native languages.

You control the operation of language-dependent features by specifying various Globalization Support parameters. You can set default parameter values in the initialization file. Table 4–8 shows what each Globalization Support parameter specifies.

Table 4–8 Globalization Support Parameters

Globalization Support 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 Globalization Support 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 Globalization Support parameters during a session, you use the ALTER SESSION statement as follows:

```
ALTER SESSION SET nls_parameter = value
```

Pro*COBOL fully supports all the Globalization Support features that allow your applications to process multilingual data stored in an Oracle9*i* 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 Globalization Support parameters to the TO_CHAR, TO_DATE, and TO_NUMBER functions. For more information about Globalization Support, see the *Oracle9i Globalization and National Language Support Guide*.

Multibyte Globalization Support Character Sets

Pro*COBOL extends support for multibyte Globalization Support character sets through

- Recognition of multibyte 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 multibyte characters.
- The NLS_NCHAR environment variable. Equate it to the client-side character set used in PIC N or PIC G.

NLS_LOCAL=YES Restrictions

When the precompiler option NLS_LOCAL is YES, the runtime library (SQLLIB) performs blank-padding and blank-stripping for Globalization Support multibyte datatypes.

When NLS_LOCAL=YES, multibyte NCHAR features are not supported within a PL/SQL block. These features include N-quoted character literals and fixed-length character variables.

These restrictions then apply:

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

No Odd-Byte Widths. Oracle9*i* CHAR columns should not be used to store multibyte 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 multibyte NCHAR host variable.

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

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

Functions should not be used on columns that store multibyte Globalization Support data.

Character Strings in Embedded SQL

A multibyte Globalization Support 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 14, "Precompiler Options".

Blank Padding

When a Pro*COBOL character variable is defined as a multibyte Globalization Support 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 4-31.

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

Output host variables are blank padded with multibyte 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 Globalization Support data, since they can only be specified using dynamic SQL or datatype equivalencing, neither of which is supported for Globalization Support data.

Indicator Variables

You can use indicator variables with multibyte Globalization Support 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.

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 Oracle9*i*. Oracle9*i* uses the codes to convert between internal and external datatypes.

Before assigning a SELECTed column value to an output host variable, Oracle9*i* 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, Oracle9*i* 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 `123465543` (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 Globalization Support parameters in the Oracle9*i* 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 Globalization Support, see the *Oracle9i Globalization and National Language Support Guide*.

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

Table 4–9 Conversions Between Internal and External Datatypes

	Internal							
External	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHAR	I/O	I/O (2)	I/O	I(3)	I/O	I/O (3)	I/O (1)	I/O
CHARF	I/O	I/O (2)	I/O	I (3)	I/O	I/O (3)	I/O (1)	I/O
CHARZ	I/O	I/O (2)	I/O	I (3)	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	I/O (3)	I/O (1)	I/O
LONG RAW	O(6)		I (5,6)	I/O		I/O		O (6)
LONG VARCHAR	I/O	I/O(2)	I/O	I (3,5)	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)
NUMBER	I/O (4)		I		I/O			I/O (4)
RAW	I/O (6)		I (5,6)	I/O		I/O		I/O (6)

Table 4–9 Conversions Between Internal and External Datatypes

ROWID	I		I				I/O	I
STRING	I/O	I/O (2)	I/O	I (3.5)	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	I/O (3)		I/O
VARCHAR2	I/O	I/O (2)	I/O	I (3)	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. 1. On input, host string must be in Oracle'BBBBBBBB.RRRR.FFFF' format.
1. On output, column value is returned in same format.
2. 2. On input, host string must be the default DATE character format.
3. On output, column value is returned in same format
4. 3. On input, host string must be in hex format.
5. On output, column value is returned in same format.
6. 4. On output, column value must represent a valid number.
7. 5. On input, length must be less than or equal to 2000.
8. 6. On input, column value is stored in hex format.
9. On output, column value must be in hex format.
10. 7. On input, host string must be a valid operating system label in text format.
11. On output, column value is returned in same format.
12. 8. On input, host string must be a valid operating system label in raw format.
13. On output, column value is returned in same format.

Legend:

I = input only

O = output only

I/O = input or output

Explicit Control Over DATE String Format

When you select a DATE column value into a character host variable, Oracle9*i* must convert the internal binary value to an external character value. So, Oracle9*i*

implicitly calls the SQL function TO_CHAR, which returns a character string in the default date format. The default is set by the Oracle9*i* 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. Oracle9*i* 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 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 4–10 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 *Oracle9i SQL Reference* for the list of date format model elements. Note that all separately compiled units linked together must use the same DATE_FORMAT value.

Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle9*i* interprets input data and the way Oracle9*i* formats output data. You can equivalence supported COBOL datatypes to external datatypes on a variable-by-variable basis.

Usefulness of Equivalencing

Datatype equivalencing is useful in several ways. For example, suppose you want to use a variable-length string in a COBOL program. You can declare a PIC X host variable, then equivalence it to the external datatype VARCHAR2.

In addition, you can use datatype equivalencing to override default datatype conversions. Unless Globalization Support parameters in the initialization file specify otherwise, if you select a DATE column value into a character host variable, Oracle9*i* returns a 9-byte string formatted as follows:

```
DD-MON-YY
```

However, if you equivalence the character host variable to the DATE external datatype, Oracle9*i* 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 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	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 4–11 on page 4-48 shows which parameters to use with each external datatype.

The CONVBUFSZ clause is explained in "CONVBUFSZ Clause in VAR Statement" on page 4-47.

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-94.

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 Oracle9*i* 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 runtime library used to perform conversion of the specified host variable between character sets.

When you have not used the CONVBUFSZ clause, the 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*, Oracle9*i* null-terminates the value for you.

Table 4–11 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 TRAILING	n/a	required	required	none
UNSIGNED DISPLAY	n/a	required	required	none

Table 4–11 Parameters for Host Variable Equivalencing

External Datatype	Length	Precision	Scale	Default Length
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
LONG RAW	optional	n/a	n/a	declared length of variable
LONG VARCHAR	required (note 1)	n/a	n/a	none
LONG VARRAW	required (note 1)	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 65533 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.s

When PICX=CHARF, specifying the datatype CHAR in a VAR statement equivalences the host-language datatype to the fixed-length ANSI datatype CHAR (Oracle9*i* 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 Oracle9*i* internal format. Keep in mind that Oracle9*i* 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 4–1 gives some examples of returned VARNUM values.

Table 4–12 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 4-43.

If no Oracle9*i* 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, Oracle9*i* must convert the internal binary value to an external character value. In this case, Oracle9*i* 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, Oracle9*i* 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, Oracle9*i* issues an error message.

For more information about datatype conversion, see "Sample Program 4: Datatype Equivalencing" on page 4-51.

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 displayed on the terminal screen.

```
*****************************************************************
* Sample Program 4:  Datatype Equivalencing                     *
*                                                               *
* This program simulates the storage and retrieval of bitmap    *
* images into table IMAGE, which is created in the SCOTT        *
* account after logging on to ORACLE.  Datatype equivalencing   *
* allows an ORACLE external type of LONG RAW to be specified    *
* for the programs representation of the images.                *
*****************************************************************

 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  INEMPNO           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.
    05  D-INEMPNO     PIC 9(4).
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")
        GO TO SIGN-OFF-EXIT.
    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, LET'S DISPLAY SOME.".

DISP-LOOP.
    MOVE 0 TO INEMPNO.
    DISPLAY " ".
    DISPLAY "ENTER EMPLOYEE NUMBER (0 TO QUIT):  "
        WITH NO ADVANCING.

    ACCEPT D-INEMPNO.

    MOVE D-INEMPNO TO INEMPNO.
    IF (INEMPNO = 0)
        GO TO SIGN-OFF.
    EXEC SQL WHENEVER NOT FOUND GOTO NO-EMP END-EXEC.
    EXEC SQL SELECT EMP.EMPNO, ENAME, SAL, NVL(COMM, 0), BITMAP
        INTO :EMP-NUMBER, :EMP-NAME, :SALARY,
             :COMMISSION:COMM-IND, :BUFFER
        FROM EMP, IMAGE
        WHERE EMP.EMPNO = :INEMPNO
          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.
    EXEC SQL DROP TABLE IMAGE END-EXEC.
SIGN-OFF-EXIT.
    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.
```

5

Embedded SQL

This chapter describes 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

Using Host Variables

Use host variables to pass data and status information to your program from the database, and to pass data to the database.

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.

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
```

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.

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 multibyte Globalization Support 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.
```

But this is less flexible.

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 COMMISSION 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.

Without an indicator variable there is no way to know whether or not a NULL was returned. The value of the host variable is undefined. If you do not use indicator variables, set the precompiler option UNSAFE_NULL=YES. Oracle therefore advises that UNSAFE_NULL=YES only be used to upgrade existing programs and that indicator variables be used for all new programs.

When UNSAFE_NULL=NO, if you select or fetch NULLs into a host variable that lacks an indicator variable, Oracle issues an error message.

For more information, see "UNSAFE_NULL" on page 14-41.

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. In all cases a WARNING will be signalled (see "Warning Flags" on page 8-9). if an

indicator variable is used with a character string, when a value is truncated, the indicator variable is set to the length of the value in the database. Note that no warning is flagged if a numeric value is truncated.

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 statements which manipulate data in database tables (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, you want to know how many rows have been updated as well as whether it succeeded. 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 "ANSI SQLSTATE Variable" on page 8-3.

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. 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 multirow SELECT statements. For a detailed discussion of each statement and its clauses, see the *Oracle9i 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 an message if more than one row is returned.

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. 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, but this is not recommended because a table's definition can change.

DML Returning Clause

The INSERT, UPDATE, and DELETE statements can have an optional *DML returning clause* which returns column value expressions *expr*, into host variables *hv*, with host indicator variables *iv*. The returning clause has this syntax:

```
{RETURNING | RETURN} {expr [,expr]}
    INTO {:hv [[INDICATOR]:iv] [, :hv [[INDICATOR]:iv]]}
```

The number of expressions must equal the number of host variables. This clause eliminates the need for selecting the rows after an INSERT or UPDATE, and before a DELETE when you need to record that information for your application. The `DML returning clause` eliminates inefficient network round-trips, extra processing,

and server memory. You will also note, for example, when a trigger inserts default values or a primary key value.

The returning_clause is not allowed with a subquery. It is only allowed after the VALUES clause.

For example, our INSERT could have a clause at its end such as:

```
RETURNING EMPNO, ENAME, DEPTNO INTO :NEW-EMP-NUMBER, :NEW-EMP-NAME, :DEPT
```

See the DELETE, INSERT, and UPDATE entries in the appendix Appendix F, "Embedded SQL Statements and Precompiler Directives".

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 5-11.

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

Cursors

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 that use the INTO clause.

The set of rows retrieved is called the *results 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*.

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 (more precisely, a directive) 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, 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 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 3-15, and "CLOSE_ON_COMMIT" on page 14-14.

If MODE is specified at a higher level than CLOSE_ON_COMMIT, then MODE takes precedence. The defaults are MODE=ORACLE and CLOSE_ON_ COMMIT=NO. If you specify MODE=ANSI then any cursors not using the WITH HOLD clause will be closed on COMMIT. The application will run more slowly because cursors are closed and re-opened many times. Setting CLOSE_ON_ COMMIT=NO when MODE=ANSI results in performance improvement. To see how macro options such as MODE affect micro options such as CLOSE_ON_ COMMIT, see "Precedence of Option Values" on page 14-4.

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 source file A.PCO, then open it in source file B.PCO.

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.

For users of MODE=ANSI or CLOSE_ON_COMMIT=YES, the WITH HOLD clause can be used in a DECLARE section to override the behavior defined by the two options. With these options set, the behavior will be for all cursors to be closed when a COMMIT is issued. This can have performance implications due to the overhead of re-opening the cursor to continue processing. The careful use of WITH HOLD can speed up programs that need to conform to the ANSI standard for precompilers in most respects.

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. 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.

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 their alphabetized entries in "Using Pro*COBOL Precompiler Options" on page 14-12.

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 MODE=ANSI, 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 ...
LOOP.
    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
    ...
    GO TO LOOP.
    ...
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 (if MODE=ANSI then the optional SQLSTATE variable will also be set.) 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 (the default when MODE=ORACLE), issuing a COMMIT or ROLLBACK will only close cursors declared using the FOR UPDATE clause or referred to by the CURRENT OF clause. Other cursors are unaffected by a COMMIT or ROLLBACK and if open, remain open. However, when CLOSE_ON_COMMIT=YES (the default when MODE=ANSI), issuing a COMMIT or ROLLBACK closes *all* cursors. For more information, see "CLOSE_ON_COMMIT" on page 14-14.

Using the CURRENT OF Clause

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

The FOR UPDATE OF clause is optional when you declare a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement. The CURRENT OF clause signals the precompiler to add a FOR UPDATE clause if necessary. For more information, see"Mimicking the CURRENT OF Clause" on page 7-19.

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'
    END-EXEC.
    ...
    EXEC SQL OPEN EMPCURSOR END-EXEC.
    EXEC SQL WHENEVER NOT FOUND DO ...
LOOP.
    EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME, :SALARY
    END-EXEC.
    ...
    EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
        WHERE CURRENT OF EMPCURSOR
```

```
END-EXEC.
GO TO LOOP.
```

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 an error.

You cannot use the CURRENT OF clause with a cursor declared with a join since internally, the CURRENT OF mechanism uses the ROWID pseudocolumn and there is no way to specify which table the ROWID relates to. For an alternative, see "Mimicking the CURRENT OF Clause" on page 7-19. Finally, you cannot use the CURRENT OF clause in dynamic SQL.

A Typical Sequence of Statements

The following example shows the typical sequence of cursor control statements using the CURRENT OF clause and the FOR UPDATE OF clause:

```
* --  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 GOTO NO-MORE
     END-EXEC.
* --  Fetch and process data in a loop.
 LOOP.
       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.
       GO TO LOOP.
     ...
```

```
 MO-MORE.
* --  Disable the cursor.
       EXEC SQL CLOSE EMPCURSOR END-EXEC.
       EXEC SQL COMMIT WORK RELEASE END-EXEC.
     STOP RUN.
```

Positioned Update

The following skeletal example demonstrates positioned update using the universal ROWID, which is defined in "Universal ROWIDs" on page 4-34:

```
...
 01  MY-ROWID SQL-ROWID.
...
     EXEC SQL ALLOCATE :MY-ROWID END-EXEC.
     EXEC SQL DECLARE C CURSOR FOR
        SELECT ROWID, ... FROM MYTABLE FOR UPDATE OF ... END-EXEC.
     EXEC SQL OPEN C END-EXEC.
     EXEC SQL FETCH C INTO :MY-ROWID ... END-EXEC.
* Process retrieved data.
...
     EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID END-EXEC.
 ...
 NO-MORE-DATA:
     EXEC SQL CLOSE C END-EXEC.
     EXEC SQL FREE :MY-ROWID END-EXEC.
...
```

The PREFETCH Precompiler Option

The precompiler option PREFETCH allows for more efficient queries by pre-fetching rows. This decreases the number of server round-trips needed and reduces memory required. The number of rows set by the PREFETCH option value in a configuration file or on the command line is used for all queries involving explicit cursors, subject to the standard precedence rules.

When used inline, the PREFETCH option must precede any of these cursor statements:

- EXEC SQL OPEN *cursor*
- EXEC SQL OPEN *cursor* USING *host_var_list*
- EXEC SQL OPEN *cursor* USING DESCRIPTOR *desc_name*

When an OPEN is executed, the value of PREFETCH gives the number of rows to be pre-fetched when the query is executed. You can set the value from 0 (no pre-fetching) to 9999. The default value is 1.

Note: The PREFETCH precompiler option is specifically designed for enhancing the performance of single row fetches. PREFETCH values have no effect when doing array fetches, regardless of which value is assigned.

Sample Program 2: Cursor Operations

This program logs on to Oracle, declares and opens a cursor, fetches the names, salaries, 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.

```
*****************************************************************
* Sample Program 2:  Cursor Operations                          *
*                                                               *
* This program logs on to ORACLE, declares and opens a cursor,  *
* fetches the names, salaries, and commissions of all           *
* salespeople, displays the results, then closes the cursor.    *
*****************************************************************

 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  COMMISSION    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 " ".
    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.

    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.
```

6

Embedded PL/SQL

This chapter shows you how to improve performance by embedding PL/SQL transaction processing blocks in your program. This chapter has the following sections:

Embedding PL/SQL

Pro*COBOL treats a PL/SQL block like a single embedded SQL statement. 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 anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.

VARCHAR Variables

When entering a PL/SQL block, Oracle9*i* automatically checks the length fields of VARCHAR host variables. Therefore, 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.

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. Further, 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 14, "Precompiler Options".

Advantages of PL/SQL

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

- Better performance
- Integration with Oracle9*i*
- Cursor FOR loops
- Procedures and functions
- Packages
- PL/SQL tables
- User-defined records

For more information about PL/SQL, see *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, Oracle9*i* 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 the server.

Integration with Oracle9*i*

PL/SQL is tightly integrated with the server. For example, most PL/SQL datatypes are native to the 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 and 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, you can write your own, such 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 and 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 6–1 on page 6-16 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 a 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. The following example "packages" 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. The following example declares 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. The following example declares 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. Thus, 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 Pro*COBOL. To connect to the server, you must also specify the option USERID. For more information, see "Using Pro*COBOL Precompiler Options" on page 14-12.

Host Variables and PL/SQL

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

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

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

PL/SQL Examples

The following example illustrates the use of host variables with PL/SQL. The program prompts the user for an employee number and 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 'Connected to Oracle'.
PERFORM
```

```
     DISPLAY 'Employee Number (0 to end)? 'WITH NO ADVANCING
     ACCEPT 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 PL/SQL Example

In the following example the user is prompted for a bank account number, transaction type, and transaction amount. The account is then debited or credited. If the account does not exist, an exception is raised. When the transaction is complete its status is displayed.

```
     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
           IF :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 that you can use the VARCHAR pseudotype to declare variable-length character strings. If the VARCHAR is an input host variable, you must tell Pro*COBOL what length to expect. Therefore, set the length field to the actual length of the value stored in the string field.

If the VARCHAR is an output host variable, Pro*COBOL 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. Therefore, 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.
```

Indicator Variables and PL/SQL

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 rule:

- 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. 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.
```

Host Tables and PL/SQL

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 COMP-3.
 01  MEDIAN-SALARY  PIC S9(6)V99 COMP-3.
     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 PL/SQL and Java Subprograms" on page 6-21.

The interface between Host Tables and PL/SQL strictly controls datatypes. The default external type for PIC X is CHARF (fixed length character string) and this can only be mapped to PL/SQL tables of type CHAR.

Table 6–1 shows the legal conversions between row values in a PL/SQL table and elements in a host table. The most notable fact is that you cannot pass a PIC X variable to a table of type VARCHAR2 without using datatype equivalencing to equivalence the variable to VARCHAR2, or using PICX=VARCHAR2 on the command line.

Table 6–1 Legal Datatype Conversions

HostTable \ PL/SQL 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 VARCHAR			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.

The following example uses 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(9) 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 the server for execution, a much smaller host table is sent along. This saves time and, in a networked environment, reduces network traffic.

Optional Keyword EXECUTE to ARRAYLEN Statement

The use of host tables used in a dynamic SQL Method 2 statement (see "Using Method 2" on page 9-12) may have two different interpretations based on the presence or absence of the keyword to EXECUTE the ARRAYLEN statement. .

If the EXECUTE keyword is absent:

- The PL/SQL block will be executed multiple times, with the actual number determined by the minimum dimension of ARRAYLEN used.
- The host array cannot be bound to a PL/SQL table.

If EXECUTE is present:

- The host table must be bound to an index table.
- The PL/SQL block will be executed once.
- All host variables specified in the EXEC SQL EXECUTE statement must:
  - Be specified in an ARRAYLEN ... EXECUTE statement, or
  - Be a scalar.

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.
      ...
          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.
        ...
```

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 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.

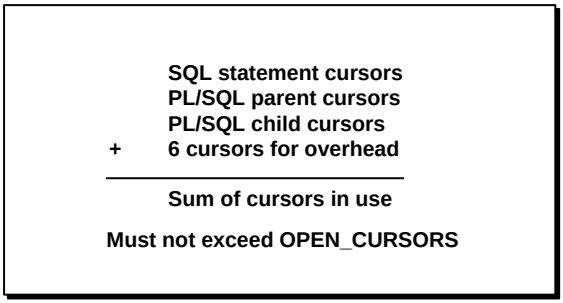
```
        ...
01  DYNSTMT   PIC X(80) VARYING.
01  INTTAB    PIC S9(9) COMP OCCURS 3 TIMES.
01  INTTAB2   PIC S9(9) COMP OCCURS 3 TIMES.
01  DIM       PIC S9(9) COMP VALUE 3.

            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.
...
```

Cursor Usage in Embedded PL/SQL

The maximum number of cursors your program can simultaneously use is determined by the database initialization parameter OPEN_CURSORS. Normally, to prevent OPEN_CURSORS being exceeded, the precompiler allows management of statement cursors. The precompiler options HOLD_CURSOR, RELEASE_CURSOR and MAXOPENCURSORS are used. (For more details on this subject, see "Embedded PL/SQL Considerations" on page D-12.) While executing an embedded PL/SQL block there will be one cursor, the parent cursor, associated with the entire PL/SQL block and a separate child cursor for each statement executed during the execution of the PL/SQL block. Because the PL/SQL block is passed to the server for execution, only the parent cursor can be tracked by the precompiler runtime library. Thus, it is possible for applications that use a lot of cursors in this way to exceed OPEN_CURSORS. Figure 6–1 shows how to calculate the maximum number of cursors used.

Figure 6–1 Maximum Cursors in Use



Developers should be aware of this situation and plan for this in the setting of OPEN_CURSORS and MAXOPENCURSORS.

If there are problems with this, you may wish to free all child cursors after a SQL statement is executed.

This can be achieved by setting RELEASE_CURSOR=YES and HOLD_CURSOR=NO. Because the use of the first setting for the entire program is likely to have an impact on performance, you can set these options in line 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
```

Stored PL/SQL and Java Subprograms

Unlike anonymous blocks, PL/SQL subprograms (procedures and functions) and Java methods can be compiled separately, stored in the database, and invoked.

A subprogram explicitly created using an Oracle tool such as SQL*Plus is called a *stored* subprogram. Once compiled and stored in the data dictionary, it is a database object can be re-executed without being re-compiled.

When a subprogram within a PL/SQL block or stored subprogram is sent to the database by your application, it is called an **inline** subprogram and is compiled by the database. Pro*COBOL sends the statement to the server for execution.

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 **standalone** 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).

If an embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement fails, Oracle9*i* generates a warning, not an error. For the full syntax of the CREATE statement see the *Oracle9i SQL Reference.*

Calling a Stored PL/SQL or Java Subprogram

To call a stored subprogram from your host program, you can use either an anonymous PL/SQL block or the CALL embedded SQL statement.

Anonymous PL/SQL Block

The following example calls a standalone 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.

You must use precompiler option SQLCHECK=SEMANTICS with an embedded PL/SQL block.

CALL Statement

The concepts presented above for the embedded PL/SQL block holds true for the CALL statement. The CALL embedded SQL statement has the form:

```
 EXEC SQL
   CALL [schema.][package.]stored_proc[@db_link](arg1, ...)
   [INTO :ret_var[[INDICATOR]:ret_ind]]
END-EXEC.
```

where:

schema
the schema containing the procedure

package
the package containing the procedure

stored_proc
is the Java or PL/SQL stored procedure to be called

db_link
is the optional remote database link

arg1...
is the list of arguments (variables, literals, or expressions) passed,

ret_var
is the optional host variable which receives the result

ind_var
the optional indicator variable for ret_var.

You can use either SQLCHECK=SYNTAX, or SQLCHECK=SEMANTICS with the CALL statement.

CALL Example

If you have created a PL/SQL function `fact` (stored in the package `mathpkg`) that takes an integer as input and returns its factorial in an integer:

```
EXEC SQL CREATE OR REPLACE PACKAGE BODY mathpkg as
  function fact(n IN INTEGER) RETURN INTEGER AS
    BEGIN
      IF (n <= 0) then return 1;
      ELSE return n * fact(n - 1);
      END IF;
    END fact;
  END mathpkge;
END-EXEC.
```

then to use `fact` in a Pro*COBOL application:

```
...
         01 N      PIC S9(4) COMP.
         01 FACT   PIC S9(9) COMP.
...
     EXEC SQL CALL mathpkge.fact(:N) INTO :FACT END-EXEC.
...
```

For more information about this statement, see "CALL (Executable Embedded SQL)" on page F-13. For a complete explanation of passing arguments and other issues, see *Oracle9i Application Developer's Guide - Fundamentals, "External Routines"* chapter.

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 9, "Oracle Dynamic SQL", Chapter 10, "ANSI Dynamic SQL"and Chapter 11, "Oracle Dynamic SQL: Method 4".

Subprograms Restriction

In dynamic SQL Method 4, a host table cannot be bound to a PL/SQL procedure with a parameter of type TABLE.

Sample Program 9: Calling a Stored Procedure

Before trying the sample program, you must create a PL/SQL package named *calldemo*, by running the following script, titled CALLDEMO.SQL, which is supplied with Pro*COBOL. The script can be found in the Pro*COBOL demo library. Check your system-specific Oracle documentation for exact spelling of the the name 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 the database, prompts the user for a department number and 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 and 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.

```
*****************************************************************
* Sample Program 9: Calling a Stored Procedure
*
* This program connects to ORACLE, prompts the user for a
* department number, then calls a PL/SQL stored procedure named
* GET_EMPLOYEES, which is stored in package CALLDEMO.  The
* procedure declares three PL/SQL tables ast 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.  The program calls the procedure repeatedly,
* displaying each batch of employee data, until no more data
* is found.
* Use option picx=varchar2 when precompiling this sample program.
*****************************************************************

 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.

    05  D-DEPT-NUM   PIC 9(2).

    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 D-DEPT-NUM.
```

```
    MOVE D-DEPT-NUM TO 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 that the datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Further, 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 through *database links*. Typically, database links are established by your DBA and stored in the data dictionary. A database link tells your program where the remote database is located, the path to it, and what username and password to use. The following example uses 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;
```

Cursor Variables

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 database server, using PL/SQL. See *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 a database. This gives you a convenient way to centralize data retrieval.

First, you declare the cursor variable. After declaring the variable, you use these statements to control a cursor variable:

- ALLOCATE
- OPEN ... FOR
- FETCH
- CLOSE
- FREE

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 and 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.
- Increased 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 runtime.

The AT clause cannot be used in an ALLOCATE statement.

> **Caution:** 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 database 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 first calling the *open_emp_cur* stored procedure from your Pro*COBOL program and then issuing a 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 as you would from a static cursor. 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 that their datatypes are compatible.

The FETCH statement must be executed on the client side. The following example fetches 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 this 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. The following example closes the cursor variable *CUR-VAR* when the last row is processed:

```
 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.
```

Freeing a Cursor Variable

To free memory allocated for the cursor variable, *CUR-VAR*, use the FREE statement after the CLOSE:

```
*    Free the cursor variable memory.
     EXEC SQL
         FREE :CUR-VAR
     END-EXEC.
```

Restrictions on Cursor Variables

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, FREE, and CLOSE commands. The DECLARE CURSOR command does *not* apply to cursor variables.
- You cannot use the AT clause with the ALLOCATE command.

Sample Program 11: Cursor Variables

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, SAMPLE11.PCO, that uses the cursor variable declared in the SAMPLE11.SQL example to fetch employee names, salaries, and commissions from the EMP table:

```
      ***************************************************************
      * Sample Program 11:  Cursor Variable Operations              *
      *                                                             *
      * This program logs on to ORACLE, allocates and opens a cursor  *
      * variable fetches the names, salaries, and commissions of all  *
      * salespeople, displays the results, then closes the cursor.    *
      ***************************************************************

       IDENTIFICATION DIVISION.
       PROGRAM-ID. CURSOR-VARIABLES.
       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  HOST              PIC X(15) VARYING.
       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(4) COMP.
    05  EMP-NAM-IND   PIC S9(4) COMP.
    05  EMP-JOB-IND   PIC S9(4) COMP.
    05  EMP-MGR-IND   PIC S9(4) COMP.
    05  EMP-DAT-IND   PIC S9(4) COMP.
    05  EMP-SAL-IND   PIC S9(4) COMP.
    05  EMP-COM-IND   PIC S9(4) COMP.
    05  EMP-DEP-IND   PIC S9(4) 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.
    05  D-EMP-DEP     PIC 9(2).


PROCEDURE DIVISION.

BEGIN-PGM.
    EXEC SQL
        WHENEVER SQLERROR DO PERFORM SQL-ERROR
    END-EXEC.
    PERFORM LOGON.
    EXEC SQL
        ALLOCATE :EMP-CUR
    END-EXEC.
    DISPLAY "Enter department number (0 to exit):  "
        WITH NO ADVANCING.
    ACCEPT D-EMP-DEP.
    MOVE D-EMP-DEP TO EMP-DEP.
    IF EMP-DEP <= 0
        GO TO SIGN-OFF
    END-IF.
    MOVE EMP-DEP TO D-DEP-NUM.
    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 GOTO CLOSE-UP
    END-EXEC.
    MOVE SPACES TO EMP-NAM-ARR.
    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.
    IF EMP-COM-IND = 0
        MOVE EMP-COM TO D-EMP-COM
        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.

CLOSE-UP.
```

```
        EXEC SQL
            CLOSE :EMP-CUR
        END-EXEC.
        EXEC SQL
            FREE :EMP-CUR
        END-EXEC.
    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.
```

7

Host Tables

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

The main sections are:

Host Tables

A *host table* (also known as an array) is a set of related data items, called *elements*, associated with a single variable. An indicator variable defined as a table is called an *indicator table*. An indicator table can be associated with any host table that is NULLABLE.

Advantages of Host Tables

Host tables can ease programming and can offer greatly improved performance. When writing an application, you are usually faced with the problem of storing and manipulating large amounts of data. Host tables simplify the task of accessing multiple return values.

Host tables let you manipulate multiple rows with a single SQL statement. Thus, communications 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 host tables your program must do 300 individual INSERTs—one for each employee. With host tables, only one INSERT need be done.

Tables in Data Manipulation 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 for 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.

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.
    ...
```

The maximum number of host table elements in a SQL statement that is accessible in one fetch is 32K (or possibly greater, depending on your platform and the available memory). If you try to access a number that exceeds the maximum, you get a "parameter out of range" runtime error. If the statement is an anonymous PL/SQL block, the number of elements accessible is limited to 32512 divided by the size of the datatype.

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. Pro*COBOL treats a SQL statement containing host tables of dimension *n* like the same statement executed *n* times with *n* different scalar host variables, but more efficiently.

Using Indicator Tables

You can use indicator tables to assign NULLs to elements in input host tables and to detect NULLs or truncated values (of character columns only) in output host tables. The following example shows how to conduct an 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.

When using host table SELECT and FETCH, it is recommended that you use indicator variables. That way you can test for NULLs in the associated output host table.

If a NULL is selected or fetched into a host variable that has no associated indicator variable, your program stops processing, sets *sqlca.sqlerrd(3)* to the number of rows processed, and returns an error.

NULL is selected by default, but you can switch it off by using the UNSAFE_NULL = YES option.

When DBMS=V7 or V8, your program does not consider truncation to be an error.

Host Group Item Containing Tables

Note: If you have a host group item containing tables, then 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.
```

Oracle Restrictions

Mixing scalar host variables with host tables in the VALUES, SET, INTO, or WHERE clause is *not* allowed. If any of the host variables is a host table, all must be host tables.

You cannot use host tables with the CURRENT OF clause in an UPDATE or DELETE statement.

ANSI Restriction and Requirements

The array interface is an Oracle extension to the ANSI/ISO embedded SQL standard. However, when you precompile with MODE=ANSI, array SELECTs and FETCHes are still allowed. The use of arrays can be flagged using the FIPS flagger precompiler option, if desired.

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 define the host tables with that number of elements. In the following example, you select directly into three host tables. The table was defined with 50 rows, with the knowledge that the select will return no more than 50 rows.

```
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 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 reexecute the SELECT statement, it just returns the first 50 rows again, even if more are eligible. You must either define a larger table or declare a cursor for use with the FETCH statement.

If a SELECT INTO statement returns more rows than the size of the table you defined, Oracle9*i* issues an error message unless you specify SELECT_ERROR=NO. For more information about the option, see "SELECT_ERROR" on page 14-37.

Batch Fetches

Use batch fetches when the size of data you are processing is large (greater than about 100 rows) as well as when you do not know how many rows will be returned.

If you do not know the maximum number of rows a select will return, you can declare and open a cursor, and 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:

```
...
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 DECLARE EMPCURSOR CURSOR FOR
    SELECT EMPNO, SAL FROM EMP
    END-EXEC.
    ...
    EXEC SQL OPEN EMPCURSOR END-EXEC.
    ...
    EXEC SQL WHENEVER NOT FOUND DO PERFORM END-IT.
LOOP.
```

```
     EXEC SQL FETCH EMPCURSOR INTO :EMP-NUMBER, :SALARY END-EXEC.
* --  process batch of rows
     ...
     GO TO LOOP.
 END-IT.
...
```

Do not forget to check how many rows were actually returned in the last fetch and to process them. See "Sample Program 3: Fetching in Batches" on page 7-10 for a complete example.

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.

Number of Rows Fetched

Each fetch returns, at most, the number of entries in the table. Fewer rows are returned in the following cases:

- The end of the active set is reached. The "no data found" warning code is returned to SQLCODE in the SQLCA. For example, this happens if you fetch into a table of number of entries 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 a table of number of entries 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 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 on Using Host Tables

Using host tables in the WHERE clause of a SELECT statement is allowed only in a sub-query. (For an example, see "The WHERE Clause" on page 7-18.) Also, since Pro*COBOL always takes the smallest dimension of table, do not mix simple host variables with host tables in the INTO clause of a SELECT or FETCH statement because only one row will be retrieved. If any of the host variables is a table, then all must be tables.

Table 7–1 shows which uses of host tables are valid in a SELECT INTO statement.

Table 7–1 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, Oracle recommends that you use indicator tables. This is because this makes it NULLs easier to find 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, Oracle9*i* stops processing, sets SQLERRD(3) to the number of rows processed, and issues an error message:

Fetching Truncated Values

If you select or fetch a truncated column value into a host table that lacks an indicator table, Oracle9*i* sets SQLWARN(2).

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 doing table selects and fetches, you can use indicator tables. That way, if Oracle9*i* 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.

Sample Program 3: Fetching in Batches

The following host table sample program can be found in the demo directory.

```
*****************************************************************
* Sample Program 3:  Host Tables                                *
*                                                               *
* This program logs on to ORACLE, declares and opens a cursor,  *
* fetches in batches using host tables, and prints the results. *
*****************************************************************

 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.
```

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 "The FOR Clause" on page 7-16.

An example of inserting with host tables follows:

```
 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.
* -- populate the host tables
     EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
         VALUES (:EMP-NAME, :EMP-NUMBER, :SALARY)
     END-EXEC.
```

The number of rows inserted will be available in SQLERRD(3).

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

```
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.
```

Restrictions on Host Tables

Mixing simple host variables with host tables in the VALUES clause of an INSERT, UPDATE, or DELETE statement causes only the first element of any host table to be processed because simple host variables are treated as host tables of dimension one and Pro*COBOL always uses the smallest declared dimension. You receive a warning when this occurs.

Updating with Tables

You can also use host tables as input variables in an UPDATE statement, as the following example shows:

```
 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.
 ...
* --  populate the host tables
     EXEC SQL
         UPDATE EMP SET SAL = :SALARY WHERE EMPNO = :EMP-NUMBER
     END-EXEC.
```

The number of rows updated by issuing this statement is available in SQLERRD(3). This is not necessarily the number of rows in the host table. The number does *not* include rows processed by an update cascade (which causes subsequent updates.)

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 in UPDATE

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 7-19.

Table 7–2 shows which uses of host tables are valid in an UPDATE statement:

Table 7–2 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. Doing so 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 in DELETE

You cannot use host tables with the CURRENT OF clause in a DELETE statement. For an alternative, see "Mimicking the CURRENT OF Clause" on page 7-19.

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:

```
 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.
* --  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 number of entries of the indicator table cannot be smaller than the number of entries of the host table.

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 (See information on Oracle dynamic SQL in Chapter 11, "Oracle Dynamic SQL: Method 4".
- 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. Internally, the value is treated as an unsigned quantity. An attempt to pass a negative value through the use of a signed host variable will result in unpredictable behavior.

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 receive an error message.

The FOR clause is not allowed in SELECT statements because its meaning is unclear. Does it mean "execute this SELECT statement *n* times"? Or, does it mean "execute this SELECT statement once, but return *n* rows"? The problem in the former case is that each execution might return multiple rows. In the latter case, it is better to declare a cursor and use the FOR clause in a FETCH statement, as follows:

```
EXEC SQL FOR :LIMIT FETCH 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.
```

The WHERE Clause

Pro*COBOL treats a SQL statement containing host tables of number of entries *n* like the same SQL statement executed *n* times with *n* different scalar variables (the individual table elements). The precompiler issues an error message only when such treatment is ambiguous:

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 following 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 following 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

The CURRENT OF clause enables you to do UPDATEs or DELETEs of the most recent row in the cursor. Use of the CURRENT OF clause causes the FOR UPDATE clause to be added to the cursor. Adding this clause has the effect of locking all rows identified by the cursor in exclusive mode. Note that you cannot use CURRENT OF with host tables. Instead, append FOR UPDATE to the definition of the cursor and explicitly select the ROWID column, then use that value to identify the current row during the update or delete. An example follows:

```
    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 DECLARE EMPCURSOR CURSOR FOR
    SELECT ENAME, JOB, ROWID FROM EMP
    FOR UPDATE
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-ID
    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.
```

Tables of Group Items as Host Variables

Pro*COBOL allows the use of tables of group items (also called *records*) in embedded SQL statements. The tables of 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.

For example, given the following declaration:

```
01    TABLES.
      05   EMP-TABLE           OCCURS 20 TIMES.
           10    EMP-NUMBER    PIC S9(4) COMP.
           10    EMP-NAME      PIC X(10).
           10    DEPT-NUMBER   PIC S9(4) COMP.
```

the following statement is valid:

```
EXEC SQL INSERT INTO EMP(EMPNO, ENAME, DEPTNO)
     VALUES(:EMP-TABLE)
END-EXEC.
```

Assuming that the group item has been filled with data already, the statement bulk inserts 20 rows consisting of the employee number, employee name, and department number into the EMP table.

Make sure that the order of the group items corresponds to the order in the SQL statement.

When using tables of group items, it is also possible to specify individual elementary items of the group. For example, the following statement is also valid. Twenty rows of employee numbers are inserted into the EMPNO column of the EMP table:

```
EXEC SQL INSERT INTO EMP (EMPNO)
     VALUES (:EMP-TABLE.EMP-NUMBER)
END-EXEC.
```

When using VARCHAR=YES, if the group item declaration resembles a VARCHAR host variable, then the group item is treated like an elementary item. Therefore, referencing this group item in SQL statements must be done using the group name, but not the elementary item names.

To use an indicator variable, set up a second table of a group item that contains an indicator variable for each variable in the group item:

```
01     TABLES-IND.
       05   EMP-TABLE-IND  OCCURS 20 TIMES.
            10   EMP-NUMBER-IND       PIC S9(4) COMP.
            10   EMP-NAME-IND         PIC S9(4) COMP.
            10   DEPT-NUMBER_IND      PIC S9(4) COMP.
```

The host indicator table of a group item can be used as follows:

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
    VALUES (:EMP-TABLE:EMP-TABLE-IND)
END-EXEC.
```

If the exact characteristics of the data are known, it is convenient to specify an elementary item indicator for a group item:

```
05    EMP-TABLE-IND     PIC S9(4) COMP
                        OCCURS 20 TIMES.
```

Host tables of group items cannot have group items that are tables. For example:

```
01   TABLES.
     05   EMP-TABLE              OCCURS 20 TIMES.
          10  EMP-NUMBER         PIC S9(4) COMP OCCURS 10 TIMES.
          10  EMP-NAME           PIC X(10).
          10  DEPT-NUMBER        PIC S9(4) COMP.
```

EMP-TABLE *cannot* be used as a host variable because EMP-NUMBER is a table.

Host tables of nested group items are not allowed. For example:

```
01   TABLES.
     05   TEAM-TABLE                  OCCURS 20 TIMES
          10   EMP-TABLE
               15   EMP-NUMBER        PIC S9(4) COMP.
               15   EMP-NAME          PIC X(10).
          10   DEPT-TABLE.
               15   DEPT-NUMBER       PIC S9(4) COMP.
               15   DEPT-NAME         PIC X(10).
```

TEAM-TABLE cannot be used as a host variable because its members (EMP-TABLE and DEPT-TABLE) are group items themselves.

Finally, the restrictions that apply to host tables in Pro*COBOL also apply to tables of group items:

- Multi-dimensional and variable-length tables are not allowed.
- If multiple tables are used in a single SQL statement, their dimensions should be the same.
- Host tables in SQL statements must not be subscripted.

Sample Program 14: Tables of Group Items

This program logs on, declares and opens a cursor, fetches in batches using a table of group items. Read the initial comments for details.

```
*****************************************************************
* Sample Program 14:  Tables of group items                     *
*                                                               *
* This program logs on to ORACLE, declares and opens a cursor,  *
* fetches in batches using a table of group items , and prints  *
* the results.  This sample is identical to sample3 except that *
* instead of using three separate host tables of five elements  *
* each, it uses a five-element table of three group items.      *
* The output should be identical.                               *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID. TABLE-OF-GROUP-ITEMS.
 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-TABLE OCCURS 5 TIMES.
    05  EMP-NUMBER    PIC S9(4) COMP.
    05  SALARY        PIC S9(6)V99
                      DISPLAY SIGN LEADING SEPARATE.
    05  EMP-NAME      PIC X(10) VARYING.
    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-REC-TABLE
    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.
```

8

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 ANSI status variables SQLCODE and SQLSTATE, or the Oracle SQLCA (SQL Communications Area) structure. You also learn how to use the WHENEVER statement and how to diagnose problems using the Oracle ORACA (Oracle Communications Area) structure.

The following topics are discussed:

Why Error Handling is Needed

A significant part of every application program must be devoted to error handling. The main benefit of error handling is that it enables 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 must trap errors because the precompiler will continue regardless of the errors encountered unless you halt processing.

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 two general methods of error handling:

- The Oracle-specific method with SQLCA and optional ORACA.
- ANSI SQL92 standard method with SQLSTATE status variable.

The precompiler MODE option governs ANSI/ISO compliance. When MODE={ANSI | ANSI14}, you declare the SQLSTATE status variable as PIC X(5). Additionally, the ANSI SQL89 SQLCODE status variable is still supported, but it is deprecated and not recommended for new programs. When MODE={ORACLE | ANSI13}, you *must* include the SQLCA through an EXEC SQL INCLUDE statement. It is possible to use both methods in one program but usually not necessary.

For detailed information on mixing methods see "Status Variable Combinations" on page 8-37.

SQLCA

The SQLCA is a record-like, host-language data structure which includes Oracle warnings, error numbers and error text. Oracle9*i* updates the SQLCA after every *executable* SQL or PL/SQL statement. (SQLCA values are undefined after a declarative statement.) By checking 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

When you use the WHENEVER statement to implicitly check the status of your SQL statements, Pro*COBOL automatically inserts error checking code after each executable statement. Alternatively, you can explicitly write your own code to test the value of the SQLCODE member of the SQLCA structure. Include SQLCA by using the embedded SQL INCLUDE statement:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

ORACA

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA, which contains cursor statistics, SQL statement text, certain option settings and system statistics. Include ORACA by using the embedded SQL INCLUDE statement:

```
EXEC SQL INCLUDE ORACA END-EXEC.
```

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 8-23.

ANSI SQLSTATE Variable

When MODE=ANSI, you can declare the ANSI SQLSTATE variable inside the Declare Section for implicit or explicit error checking. If the option DECLARE_SECTION is set to NO, then you can also declare it outside of the Declare Section.

> **Note:** When MODE=ANSI, you can also declare the SQLCODE variable with a picture S9(9) COMP. While it can be used instead of or with the SQLSTATE variable, this is not recommended for new programs. You can also use the SQLCA with the SQLSTATE variable. When MODE=ANSI14, then SQLSTATE is not supported and you must declare either SQLCODE or include SQLCA. You cannot declare both SQLCODE and SQLCA for any setting of mode.

Declaring SQLSTATE

This section describes how to declare SQLSTATE. SQLSTATE must be declared as a five-character alphanumeric string as 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.
```

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 8–1 shows the coding scheme:

Figure 8–1 SQLSTATE Coding Scheme

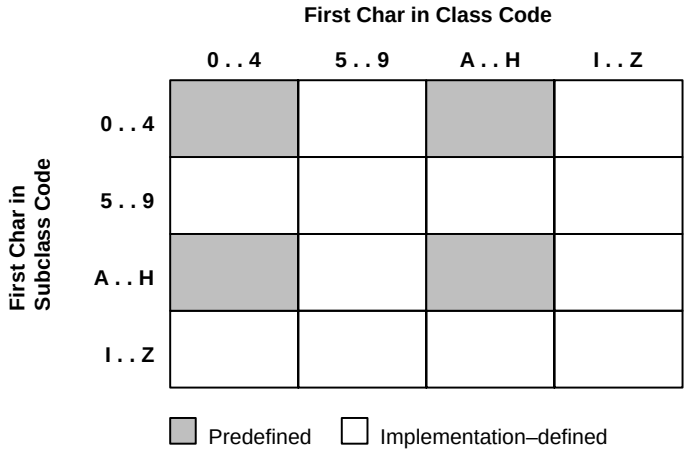


Table 8–1 shows the classes predefined by SQL92.

Table 8–1 Predefined 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

Table 8–1 Predefined Classes

Class	Condition
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
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 8–4, "SQLSTATE Codes" shows how errors map to SQLSTATE status codes. In some cases, several errors map to the status code. In other cases, no error maps to the status code (so the last column is empty). Status codes in the range 60000..99999 are implementation-defined.

Using the SQL Communications Area

Oracle9*i* 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 Oracle9*i* 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.

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 if you want to use the WHENEVER SQLWARNING statement, you must declare the SQLCA. The SQLCA must also be included when using multibyte NCHAR host variables.

> **Note:** When your application uses Oracle 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 3-3

Contents of the SQLCA

The SQLCA contains runtime information about the execution of SQL statements, such as error codes, warning flags, event information, rows-processed count, and diagnostics.

Figure 8–2 shows all the variables in the SQLCA.

Figure 8–2 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 SQLWARNO             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 Oracle9*i* to communicate with the program.

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.

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, Oracle9*i* must parse it; that is, examine it to make sure it follows syntax rules and refers to valid database objects. If Oracle9*i* 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, Oracle9*i* sets SQLERRD(5) to zero. Oracle9*i* 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 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 in "Getting the Full Text of Error Messages" on page 8-13.

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	Oracle9*i* executed the statement without detecting an error or exception.
> 0	Oracle9*i* executed the statement but detected an exception. This occurs when Oracle9*i* 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. Oracle9*i* 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 *Oracle9i Database Error 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.

SQLWARN

This table of single characters has eight elements. They are used as warning flags. Oracle9*i* 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 Oracle9*i* assigns a truncated column value to an output host character variable.

> **Note:** Figure 8–2, "SQLCA Variable Declarations for Pro*COBOL" on page 8-8 illustrates SQLWARN implementation in Pro*COBOL as a group item with elementary PIC X items named SQLWARN0 through SQLWARN7.

Descriptions of the fields in SQLWARN follow:

SQLWARN(1)	This flag is set if another warning flag is set.
SQLWARN(2)	This flag is set if a truncated column value was assigned to an output host variable. This applies only to character data. Oracle9*i* 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(3)	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(4)	This flag is set if the number of columns in a query select list does not equal the number of host variables in the INTO clause of the SELECT or FETCH statement. The number of items returned is the lesser of the two.
SQLWARN(5)	This flag is no longer in use.

SQLWARN(6)	This flag is set when an EXEC SQL CREATE {PROCEDURE | FUNCTION | PACKAGE | PACKAGE BODY} statement fails because of a PL/SQL compilation error.
SQLWARN(7)	This flag is no longer in use.
SQLWARN(8)	This flag is no longer in use.

SQLEXT

This string field is reserved for future use.

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

Regardless of the setting of MODE, you can use SQLGLM to get the full text of error messages if you have explicitly declared SQLCODE and not included SQLCA. 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 a database, you can call SQLGLM using the syntax

```
CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH
```

where the parameters are:

Parameter	Datatype	Parameter Definition
MSG-TEXT	PIC X(n)	The field in which to store the error message. (Oracle9*i* blank-pads to the end of this field.)
MAX-SIZE	PIC S9(9) COMP	An integer that specifies the maximum size of the MSG-TEXT field in bytes.
MSG-LENGTH	PIC S9(9) COMP	An integer variable in which Oracle9*i* stores the actual length of the error message.

All parameters must be passed by reference. This is usually the default parameter passing convention; you need not take special action.

The maximum length of an 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:

```
     ...
*    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, 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 Oracle from DB2, Pro*COBOL provides DSNTIAR. The DSNTIAR 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 listed in the following table.

Table 8–2 DSNTIAR 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)

WHENEVER Directive

By default, Pro*COBOL ignores 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 Oracle9*i* 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 Oracle9*i* automatically check the SQLCA for any of the following conditions.

Conditions

SQLWARNING

SQLWARN(0) is set because Oracle9*i* 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 Oracle9 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.

Note: You have to have included SQLCA for this to work.

SQLERROR

SQLCODE has a negative value if Oracle9*i* returns an error.

NOT FOUND or NOTFOUND

SQLCODE has a value of +1403 (or +100 when MODE={ANSI | ANSI14 | ANSI13} or when END_OF_FETCH=100) when the end of fetch has been reached. This can happen when all the rows that meet the search criteria have been fetched or no rows meet that criteria.

You may use the END_OF_FETCH option to override the value use by the MODE macro option.

```
END_OF_FETCH = 100 | 1403 (default 1403)
```

For more details, see "END_OF_FETCH" on page 14-19

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 CALL

Your program calls a nested subprogram. When the end of the subprogram is reached, control transfers to the statement that follows the failed SQL statement.

DO PERFORM

Your program transfers control to a COBOL section or paragraph. When the end of the section is reached, control transfers to the statement that follows the failed SQL statement.

```
EXEC SQL
    WHENEVER <condition> DO PERFORM <section_name>
END-EXEC.
```

GOTO or GO TO

Your program branches to the specified paragraph or section.

STOP

Your program stops running and uncommitted work is rolled back.

Be careful. The STOP action displays no messages before logging off.

Coding the WHENEVER Statement

Code the WHENEVER statement using the following syntax:

```
EXEC SQL
    WHENEVER <condition> <action>
END-EXEC.
```

DO PERFORM

When using the WHENEVER ... DO PERFORM statement, the usual rules for PERFORMing a paragraph or section 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 PERFORM statements are used to handle specific errors:

```
 PROCEDURE DIVISION.
 MAIN SECTION.
 MSTART.
     ...
     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.
     ...
 MEXIT.
     STOP RUN.
 INS-ERROR SECTION.
 INSSTART.
*    Check for "duplicate key value" Oracle9 error
     IF SQLCA.SQLCODE = -1
     ...
*    Check for "value too large" Oracle9 error
     ELSE IF SQLCA.SQLCODE = -1401
     ...
     ELSE
     ...
     END-IF.
     ...
 INSEXIT.
     EXIT.
*
 DEL-ERROR SECTION.
 DSTART.
*    Check for the number of rows processed.
     IF SQLCA.SQLERRD(3) = 0
     ...
     ELSE
     ...
     END-IF.
     ...
 DEXIT.
     EXIT.
```

Notice how the paragraphs check variables in the SQLCA to determine a course of action.

DO CALL

This clause calls an action subprogram. Here is the syntax of this clause:

```
     EXEC SQL
```

```
    WHENEVER <condition> DO CALL <subprogram_name>
    [USING <param1> ...]
END-EXEC.
```

The following restrictions or rules apply:

- You cannot use the RETURNING, ON_EXCEPTION, or OVER_FLOW phrases in the USING clause.
- You may have to enter the subprogram name followed by the keyword COMMON in the PROGRAM-ID statement of your COBOL source code.
- You must use a WHENEVER CONTINUE statement in the action subprogram.
- The action subprogram name may have to be in double quotes in the DO CALL clause of the WHENEVER directive.

Here is an example of a program that can call the error subprogram SQL-ERROR from inside the subprogram LOGON, or inside the MAIN program, without having to repeat code in two places, as when using the DO PERFORM clause:

```
IDENTIFICATION DIVISION.
           PROGRAM-ID. MAIN.
           ENVIRONMENT DIVISION.
       ...
           PROCEDURE DIVISION.
           BEGIN-PGM.
               EXEC SQL
                   WHENEVER SQLERROR DO CALL "SQL-ERROR"
               END-EXEC.
               CALL "LOGON".
       ...
           IDENTIFICATION DIVISION.
           PROGRAM-ID. LOGON.
           DATA DIVISION.
           WORKING-STORAGE SECTION.
           01  USERNAME          PIC X(15) VARYING.
           01  PASSWD            PIC X(15) VARYING.
           PROCEDURE DIVISION.
               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.
        END PROGRAM LOGON.
    ...
        IDENTIFICATION DIVISION.
        PROGRAM-ID. SQL-ERROR COMMON.
        PROCEDURE DIVISION.
            EXEC SQL
                WHENEVER SQLERROR CONTINUE
            END-EXEC.
            DISPLAY " ".
            DISPLAY SQLERRMC.
            EXEC SQL
                ROLLBACK WORK RELEASE
            END-EXEC.
            END PROGRAM SQL-ERROR.
        END PROGRAM MAIN.
```

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

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 8–3 shows the host-language datatypes available for the parameters in the SQLGLS argument list.

Table 8–3 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. A complete table of the function code for each SQL command is found in *Oracle Call Interface Programmer's Guide*.

There are no SQL function codes for these statements:

- CONNECT
- COMMIT
- FETCH
- ROLLBACK
- RELEASE

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 Oracle9*i*-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 resources such as the SQL Statement Executor and the *cursor cache*, an area of memory reserved for cursor management.

Contents of the ORACA

The ORACA contains option settings, system statistics, and extended diagnostics. Figure 8–3 shows all the variables in the ORACA:

Figure 8–3 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 nonzero values enables 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 runtime library does the consistency checking and can issue error messages, which are listed in *Oracle9i Database 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	Allow DEBUG operations to be enabled.

ORAHCHF

If the master DEBUG flag (ORADBGF) is set, this flag tells the 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	Enable heap consistency checking (the default).
1	Disable heap consistency checking.

ORASTXTF

This flag lets you specify when the text of the current SQL statement is saved. It has the following settings:

0	Never save the SQL statement text (the default).
1	Save the SQL statement text on SQLERROR only.
2	Save the SQL statement text on SQLERROR or SQLWARNING.
3	Always save the SQL statement text.

The SQL statement text is saved in the ORACA 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 Oracle9*i*. 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 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 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, "Performance Tuning".

ORACA Example Program

The following program prompts for a department number, inserts the name and salary of each employee in that department into one of two tables, and 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.
```

How Errors Map to SQLSTATE Codes

The following table describes SQLSTATE the codes, what they signify, and the returned errors.

Table 8–4 SQLSTATE Codes

Code	Condition	Oracle9*i* 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	
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	

Table 8–4 SQLSTATE Codes

Code	Condition	Oracle9*i* Error
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
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	

Table 8–4 SQLSTATE Codes

Code	Condition	Oracle9*i* 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
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	

Table 8–4 SQLSTATE Codes

Code	Condition	Oracle9*i* Error
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	
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

Table 8–4 SQLSTATE Codes

Code	Condition	Oracle9*i* Error
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	multithreaded server and detached process errors	ORA-00100 .. 00120
		ORA-00440 .. 00569
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	Oracle 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

Table 8–4 SQLSTATE Codes

Code	Condition	Oracle9*i* Error
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
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 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

Table 8–4 SQLSTATE Codes

Code	Condition	Oracle9*i* Error
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
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	

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).
- The ASSUME_SQLCODE setting.

Table 8–5 and Table 8–6 describe the resulting behavior of each status variable combination when ASSUME_SQLCODE=NO and when ASSUME_ SQLCODE=YES, respectively.

For both Tables: 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 8–5 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 8–5 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 8–6 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.

Table 8–6 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	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.
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	

9

Oracle 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:

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, usernames, and views

Requirements for Dynamic SQL Statements

To represent a dynamic SQL statement, a character string must contain the text of a valid DML or DDL SQL statement, but *not* contain the EXEC SQL clause, host-language delimiter or statement terminator.

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, Oracle9*i* 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 Oracle9*i* *parses* the SQL statement. That is, Oracle9*i* 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, Oracle9*i* *binds* the host variables to the SQL statement. That is, Oracle9*i* gets the addresses of the host variables so that it can read or write their values.

If the statement is a query, you define the SELECT variables and then Oracle9*i* FETCHes them until all rows are retrieved. The cursor is then closed.

Then Oracle9*i executes* the SQL statement. That is, Oracle9*i* 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 the 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.

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 9–1 shows:

Table 9–1 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 place-holders 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 (regardless of whether you have set 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 can be parsed just once by calling PREPARE once, and executed many times with different values for the host variables. This is not true when RELEASE_CURSOR=YES is also specified, because the statement has to be prepared again before each execution.

Note: SQL data definition statements such as CREATE are executed once the PREPARE is completed.

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 Oracle Method 4" on page 9-24). 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 9–1, "Choosing the Right Method" on page 9-7, will help you choose the correct method.

Avoiding Common Errors

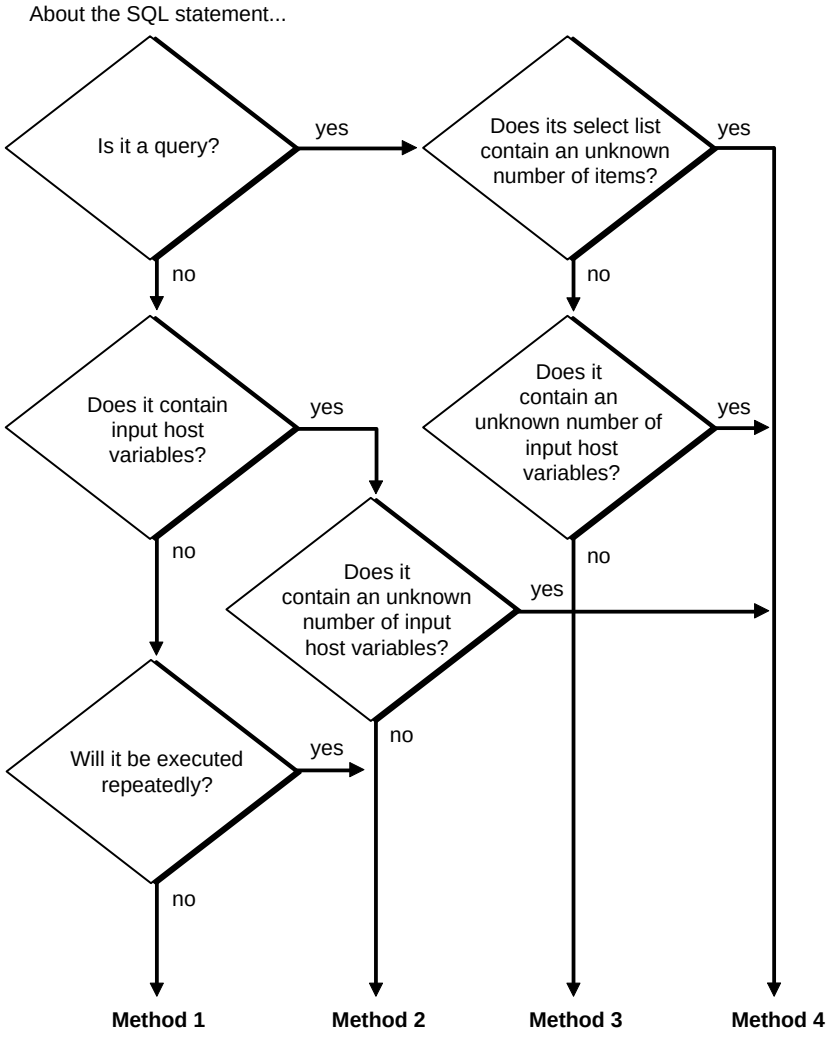
If you use a character array to store the dynamic SQL statement, blank-pad the array before storing the SQL statement. That way, you clear extraneous characters. This is especially important when you reuse the array for different SQL statements. As a rule, always initialize (or re-initialize) the host string before storing the SQL statement.

Do not null-terminate the host string. Oracle9*i* does not recognize the null terminator as an end-of-string marker. Instead, Oracle9*i* 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 9–1 Choosing the Right Method



Using Method 1

The simplest kind of dynamic SQL statement results only in "success" or "failure" and uses no host variables. Some examples follow:

```
'DELETE FROM table_name WHERE column_name = constant'
'CREATE TABLE table_name ...'
'DROP INDEX index_name'
'UPDATE table_name SET column_name = constant'
'GRANT SELECT ON table_name TO username'
```

The EXECUTE IMMEDIATE Statement

Method 1 parses, then immediately executes the SQL statement using the EXECUTE IMMEDIATE command. The command is followed by a character string (host variable or literal) containing the SQL statement to be executed, which cannot be a query.

The syntax of the EXECUTE IMMEDIATE statement follows:

```
EXEC SQL EXECUTE IMMEDIATE { :HOST-STRING | STRING-LITERAL }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.
     ...
```

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.

```
*****************************************************************
* 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 ORACLE MAKES ADDITIONAL RUNTIME STATUS
*    INFORMATION 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.

*    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.

     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 AS USER:  " WITH NO ADVANCING.
           DISPLAY USERNAME.
           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))".
           DISPLAY " ".
           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.
           DISPLAY " ".

       *   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.
           DISPLAY " ".
           EXEC SQL EXECUTE IMMEDIATE :DYNSTMT END-EXEC.

       *   COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
           EXEC SQL COMMIT RELEASE END-EXEC.
```

```
           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, if you have not specified MODE=ANSI, you need *not* re-prepare the SQL statement after a COMMIT or ROLLBACK (unless you log off and reconnect).

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 COMP-1, because Oracle9*i* supports all these datatype conversions to the NUMBER internal datatype.

The USING Clause

When the SQL statement EXECUTE is completed, input host variables in the USING clause replace corresponding place-holders in the prepared dynamic SQL statement.

Every place-holder in the dynamic SQL statement after PREPARE must correspond to a host variable in the USING clause. So, if the same place-holder appears two or more times in the statement after PREPARE, 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. Otherwise, only one record is then processed.

The names of the place-holders need not match the names of the host variables. However, the order of the place-holders in the dynamic SQL statement after PREPARE 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 and then delete them.

```
*****************************************************************
* 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.

     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 10.
 01  DEPTNO2   PIC S9(4) COMPUTATIONAL VALUE 20.
     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.

*    ROLLBACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
     EXEC SQL ROLLBACK 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

The PREPARE statement 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.

The following statement is correct also:

```
EXEC SQL
    PREPARE SQLSTMT FROM 'SELECT MGR, JOB FROM EMP WHERE SAL < :SALARY'
END-EXEC.
```

The following PREPARE statement, which uses the '%' wildcard, is also correct:

```
MOVE "SELECT ENAME FROM TEST WHERE ENAME LIKE 'SMIT%'" TO MY-STMT.
```

```
EXEC SQL
    PREPARE S FROM MY-STMT
END-EXEC.
```

DECLARE

The DECLARE statement 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 *EMPCURSOR* and associates it with *SQLSTMT*, 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

The OPEN statement allocates a 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

The FETCH statement 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" 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.
```

Host tables can be used with Method 3.

CLOSE

The CLOSE statement 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.

```
*****************************************************************
* 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.

     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 S9999 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 WITH A PREPARED
*      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.
```

Using Oracle Method 4

This section gives only an overview. For details, see Chapter 11, "Oracle Dynamic SQL: Method 4".

LOBs are not supported in Oracle Method 4. Use ANSI dynamic SQL for LOB applications and all other new applications.

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 Oracle9*i* 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 used OPEN for two or more cursors, for example), each statement must have its own SQLDAs statement. 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.

SQLDA Contents

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 variables 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 :sqlstring END-EXEC.
EXEC SQL EXECUTE sqlstmt END-EXEC.
```

In the example, *remotedb* tells Oracle9*i* 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 :sqlstring END-EXEC.
```

The usual sequence of statements is

```
EXEC SQL PREPARE sqlstmt FROM :sqlstring 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 Oracle9*i* 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:

- All PL/SQL host variables should be treated in the same way as input host variables regardless of whether they are input or output host variables (or both).
- You cannot FETCH from a PL/SQL block because it might contain any number of SQL statements. However, you can implement similar functionality by using cursor variables.

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. Once the PL/SQL string EXECUTE is completed, host variables in the USING clause replace corresponding place-holders in the string after PREPARE. 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 PL/SQL string after PREPARE 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 completion of a FETCH. 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 you refer to all PL/SQL host variables with the methods associated with 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.

Note that 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 (/* ... */).

10

ANSI Dynamic SQL

This chapter describes Oracle's implementation of ANSI dynamic SQL (also known as SQL92 dynamic SQL) which should be used for new Method 4 applications. It has enhancements over the older Oracle dynamic SQL Method 4, which is described in Chapter 11, "Oracle Dynamic SQL: Method 4". The ANSI Method 4 supports all Oracle types, while the older Oracle Method 4 does *not* support cursor variables, tables of group items, the DML returning clause, and LOBs.

In ANSI dynamic SQL, descriptors are internally maintained by Oracle, while in the older Oracle dynamic SQL Method 4, descriptors are defined in the user's Pro*COBOL program. In both cases, with Method 4 your Pro*COBOL program accepts or builds SQL statements that contain a varying number of host variables.

The main sections in this chapter are:

- Basics of ANSI Dynamic SQL
- Overview of ANSI SQL Statements
- Oracle Extensions
- ANSI Dynamic SQL Precompiler Options
- Full Syntax of the Dynamic SQL Statements
- Sample Programs: SAMPLE12.PCO

Basics of ANSI Dynamic SQL

Consider the SQL statement:

```
SELECT ename, empno FROM emp WHERE deptno = :deptno_data
```

The steps you follow to use ANSI dynamic SQL are:

- Declare variables, including a string to hold the statement to be executed.
- Allocate descriptors for input and output variables.
- Prepare the statement.
- Describe input for the input descriptor.
- Set the input descriptor (in our example the one input host bind variable, `deptno_data`).
- Declare and open a dynamic cursor.
- Set the output descriptors (in our example, the output host variables `ename` and `empno`).
- Repeatedly fetch data, using GET DESCRIPTOR to retrieve the `ename` and `empno` data fields from each row.
- Do something with the data retrieved (output it, for instance).
- Close the dynamic cursor and deallocate the input and output descriptors.

Precompiler Options

Normally, if you are using ANSI dynamic SQL you will be writing to the ANSI standard for precompilers and will therefore be using the macro command line option MODE=ANSI. If you wish to use this method and do not wish to use MODE=ANSI, then the functionality is controlled by the micro command line option: DYNAMIC=ANSI.

You can either set the micro precompiler option DYNAMIC to ANSI, or set the macro option MODE to ANSI. This causes the default value of DYNAMIC to be ANSI. The other setting of DYNAMIC is ORACLE. For more about micro options, see "Macro and Micro Options" on page 14-5 and "DYNAMIC" on page 14-18.

In order to use ANSI type codes, set the precompiler micro option TYPE_CODE to ANSI, or set the macro option MODE to ANSI. This changes the default setting of TYPE_CODE to ANSI. To set TYPE_CODE to ANSI, DYNAMIC must also be ANSI.

For a list of the ANSI SQL types see Table 10–1 on page 10-4. Use the ANSI types with precompiler option TYPE_CODE set to ANSI if you want your application to be portable across database platforms and be as compliant to ANSI as possible.

For more details, see "MODE" on page 14-30 and "TYPE_CODE" on page 14-40.

Overview of ANSI SQL Statements

Allocate a descriptor area before using it in a dynamic SQL statement.

The ALLOCATE DESCRIPTOR statement syntax is:

```
EXEC SQL ALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   [WITH MAX {:occurrences | numeric_literal}]
END-EXEC.
```

A global descriptor can be used in any module in the program. A local descriptor can be accessed only in the file in which it is allocated. Local is the default.

The descriptor name, `desc_nam`, is a host variable. A string literal can be used instead.

`occurrences` is the maximum number of bind variables or columns that the descriptor can hold, with a default of 100.

When a descriptor is no longer needed, deallocate it to conserve memory. Deallocation is done automatically when there are no more database connections.

The deallocate statement is:

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
END-EXEC.
```

Use the DESCRIBE statement to obtain information on a prepared SQL statement. DESCRIBE INPUT describes bind variables for the dynamic statement that has been prepared. DESCRIBE OUTPUT (the default) can give the number, type, and length of the output columns. The simplified syntax is:

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] sql_statement
   USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

If your SQL statement has input and output values, you must allocate two descriptors: one for input and one for output values. If there are no input values, for example:

```
SELECT ename, empno FROM emp
```

then the input descriptor is not needed.

Use the SET DESCRIPTOR statement to specify input values for INSERTS, UPDATES, DELETES and the WHERE clauses of SELECT statements. Use SET DESCRIPTOR to set the number of input bind variables (stored in *COUNT*) when you have not done a DESCRIBE into your input descriptor:

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   COUNT = {:kount | numeric_literal}
END-EXEC.
```

`kount` can be a host variable or a numeric literal, such as 5. Use a SET DESCRIPTOR statement for each host variable, giving at least the data value of the variable:

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number DATA = :hv3
END-EXEC.
```

You can also set the type and length of the input host variable:

> **Note:** If you do not set the type and length, either explicitly, through the SET DESCRIPTOR statement, or implicitly by doing a DESCRIBE OUTPUT, when TYPE_CODE=ORACLE, the precompiler will use values for these derived from the host variable itself. When TYPE_CODE=ANSI, you must set the type using the values in Table 10–1, "ANSI SQL Datatypes". You should also set the length because the ANSI default lengths may not match those of your host variable.

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number TYPE = :hv1, LENGTH = :hv2, DATA = :hv3
END-EXEC.
```

We use the identifiers `hv1`, `hv2`, and `hv3` to remind us that the values must be supplied by host variables. *item_number* is the position of the input variable in the SQL statement. It can be a host variable or an integer number.

TYPE is the Type Code selected from the following table, if TYPE_CODE is set to ANSI:

Table 10–1 ANSI SQL Datatypes

Datatype	Type Code
CHARACTER	1

Table 10–1 ANSI SQL Datatypes

Datatype	Type Code
CHARACTER VARYING	12
DATE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2
REAL	7
SMALLINT	5

See Table 11–2, "Oracle External and Related COBOL Datatypes" on page 11-16 for the Oracle type codes. Use the negative value of the Oracle code if the ANSI datatype is not in the table, and TYPE_CODE = ANSI.

DATA is the host variable value which is input.

You can also set other input values such as indicator, precision and scale. See the more complete discussion of "SET DESCRIPTOR" on page 10-18 for a list of all the possible descriptor item names.

The numeric values in the SET DESCRIPTOR statement must be declared as either `PIC S9(9) COMP` or `PIC S9(4) COMP` except for indicator and returned length values which you must declare as `PIC S9(4)COMP`.

In the following example, when you want to retrieve empno, set these values: VALUE=2, because `empno` is the second output host variable in the dynamic SQL statement. The host variable `EMPNO-TYP` is set to 3 (Oracle Type for integer). The length of a host integer, `EMPNO-LEN`, is set to 4, which is the size of the host variable. The DATA is equated to the host variable EMPNO-DATA which will receive the value from the database table. The code fragment is as follows:

```
...
01  DYN-STATEMENT PIC X(58)
    VALUE "SELECT ename, empno FROM emp WHERE deptno =:deptno_number".
01  EMPNO-DATA PIC S9(9) COMP.
01  EMPNO-TYP  PIC S9(9) COMP   VALUE 3.
01  EMPNO-LEN  PIC S9(9) COMP   VALUE 4.
```

```
...
    EXEC SQL SET DESCRIPTOR 'out' VALUE 2  TYPE=:EMPNO-TYP, LENGTH=:EMPNO-LEN,
       DATA=:EMPNO-DATA END-EXEC.
```

After setting the input values, execute or open your statement using the input descriptor. If there are output values in your statement, set them before doing a FETCH. If you have done a DESCRIBE OUTPUT, you may have to reset the actual types and lengths of your host variables because the DESCRIBE execution will produce internal types and lengths which differ from your host variable external types and length.

After the FETCH of the output descriptor, use GET DESCRIPTOR to access the returned data. Again we show a simplified syntax with details later in this chapter:

```
EXEC SQL GET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number
   :hv1 = DATA, :hv2 = INDICATOR, :hv3 = RETURNED_LENGTH
END-EXEC.
```

`desc_nam` and `item_number` can be literals or host variables. A descriptor name can be a literal such as 'out'. An item number can be a numeric literal such as 2.

`hv1, hv2, and hv3` are host variables. They must be host variables, not literals. Only three are shown in the example. See Table 10–4, "Definitions of Descriptor Item Names" on page 10-16 for a list of all possible items of returned data that you can get.

Use either `PIC S9(n) COMP` where n is the platform-dependent upper limit, `PIC S9(9)COMP` or `PIC S9(4)COMP` for all numeric values, except for indicator and returned length variables, which must be `PIC S9(4) COMP`.

Sample Code

The following example demonstrates the use of ANSI Dynamic SQL. It allocates an input descriptor `in` and an output descriptor `out` to execute a SELECT statement. Input values are set through the SET DESCRIPTOR statement. The cursor is opened and fetched from and the resulting output values are retrieved through a GET DESCRIPTOR statement.

```
...
01  DYN-STATEMENT PIC X(58)
    VALUE "SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO =:DEPTNO-DAT".
01  EMPNO-DAT  PIC S9(9) COMP.
01  EMPNO-TYP  PIC S9(9) COMP   VALUE 3.
01  EMPNO-LEN  PIC S9(9) COMP   VALUE 4.
```

```
 01  DEPTNO-TYP PIC S9(9) COMP   VALUE 3.
 01  DEPTNO-LEN PIC S9(9) COMP   VALUE 2.
 01  DEPTNO-DAT PIC S9(9) COMP   VALUE 10.
 01  ENAME-TYP  PIC S9(9) COMP   VALUE 3.
 01  ENAME-LEN  PIC S9(9) COMP   VALUE 30.
 01  ENAME-DAT  PIC X(30).
 01  SQLCODE PIC S9(9)    COMP   VALUE 0.
 ...
* Place preliminary code, including connection, here
...
     EXEC SQL ALLOCATE DESCRIPTOR 'in' END-EXEC.
     EXEC SQL ALLOCATE DESCRIPTOR 'out' END-EXEC.
     EXEC SQL PREPARE s FROM :DYN-STATEMENT END-EXEC.
     EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' END-EXEC.
     EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE=:DEPTNO-TYP,
        LENGTH=:DEPTNO-LEN, DATA=:DEPTNO-DAT END-EXEC.
     EXEC SQL DECLARE c CURSOR FOR s END-EXEC.
     EXEC SQL OPEN c USING DESCRIPTOR 'in' END-EXEC.
     EXEC SQL DESCRIBE OUTPUT s USING DESCRIPTOR 'out' END-EXEC.
     EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:ENAME-TYP,
        LENGTH=:ENAME-LEN, DATA=:ENAME-DAT END-EXEC.
     EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE=:EMPNO-TYP,
        LENGTH=:EMPNO-LEN, DATA=:EMPNO-DAT END-EXEC.

     EXEC SQL WHENEVER NOT FOUND GOTO BREAK END-EXEC.
 LOOP.
     IF SQLCODE NOT = 0
        GOTO BREAK.
     EXEC SQL FETCH c INTO DESCRIPTOR 'out' END-EXEC.
     EXEC SQL GET DESCRIPTOR 'OUT' VALUE 1 :ENAME-DAT = DATA END-EXEC.
     EXEC SQL GET DESCRIPTOR 'OUT' VALUE 2 :EMPNO-DAT = DATA END-EXEC.
     DISPLAY "ENAME = " WITH NO ADVANCING
     DISPLAY ENAME-DAT  WITH NO ADVANCING
     DISPLAY "EMPNO = " WITH NO ADVANCING
     DISPLAY EMPNO-DAT.
     GOTO LOOP.
 BREAK:
     EXEC SQL CLOSE c END-EXEC.
     EXEC SQL DEALLOCATE DESCRIPTOR 'in' END-EXEC.
     EXEC SQL DEALLOCATE DESCRIPTOR 'out' END-EXEC.
```

Oracle Extensions

These extensions are described next:

- Reference semantics for data items in SET statements.
- Arrays for bulk operations.
- Support for object types, NCHAR columns, and LOBs.

Reference Semantics

The ANSI standard specifies *value* semantics. To improve performance, Oracle has extended this standard to include *reference* semantics.

Value semantics makes a copy of your host variables data. Reference semantics uses the addresses of your host variables, avoiding a copy. Thus, reference semantics can provide performance improvements for large amounts of data.

To help speed up fetches, use the REF keyword before the data clauses:

```
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:ENAME-TYP,
   LENGTH=:ENAME-LEN, REF DATA=:ENAME-DAT END-EXEC.
EXEC SQL DESCRIPTOR 'out' VALUE 2 TYPE=:EMPNO-TYP,
   LENGTH=:EMPNO-LEN, REF DATA=:EMPNO-DAT END-EXEC.
```

Then the host variables receive the results of the retrieves. The GET statement is not needed. The retrieved data is written directly into `ename_data` and `empno_data` after each FETCH.

Use of the REF keyword is allowed *only* before DATA, INDICATOR and RETURNED_LENGTH items (which can vary with each row fetched) as in this fragment of code:

```
01   INDI       PIC S9(4) COMP.
01   RETRN-LEN  PIC S9(9) COMP.
...
     EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:ENAME-TYP,
        LENGTH=:ENAME-LEN, REF DATA=:ENAME-DAT,
        REF INDICATOR=:INDI, REF RETURNED_LENGTH =:RETRN-LEN END-EXEC.
```

After each fetch, `RETRN-LEN` holds the actual retrieved length of the `ename` field, which is useful for `CHAR` or `VARCHAR2` data.

`ENAME-LEN` will not receive the returned length. It will not be changed by the FETCH statement. Use a DESCRIBE statement, followed by a GET statement to find out the maximum column width before fetching rows of data.

REF keyword is also used for other types of SQL statements than SELECT, to speed them up. Note that with reference semantics, the host variable is used rather than a

value copied into the descriptor area. The host variable data at the time of execution of the SQL statement is used, not its data at the time of the SET. Here is an example:

```
 ...
     MOVE 1 to VAL.
 ...
     EXEC SQL SET DESCRIPTOR 'value' VALUE 1 DATA = :VAL END-EXEC.
     EXEC SQL SET DESCRIPTOR 'reference' VALUE 1 REF DATA = :VAL END-EXEC.
     MOVE 2 to VAL.
* Will use  VAL = 1
     EXEC SQL EXECUTE s USING  DESCRIPTOR 'value' END-EXEC.
*Will use VAL = 2
     EXEC SQL EXECUTE s USING DESCRIPTOR 'reference' END-EXEC.
```

See "SET DESCRIPTOR" on page 10-18 for many more details on the differences.

Using Tables for Bulk Operations

Oracle extends the SQL92 ANSI dynamic standard by providing bulk operations. To use bulk operations, use the FOR clause with an array size to specify the amount of input data or the number of rows you want to process.

The FOR clause is used in the ALLOCATE statement to give the maximum amount of data or number of rows. For example, to use a maximum array size of 100:

```
EXEC SQL FOR 100 ALLOCATE DESCRIPTOR 'out' END-EXEC.
```

or:

```
MOVE 100 TO INT-ARR-SIZE.
EXEC SQL FOR :INT-ARR-SIZE ALLOCATE DESCRIPTOR 'out' END-EXEC.
```

The FOR clause is then used in subsequent statements that access the descriptor. In an output descriptor the FETCH statement must have an array size equal to or less than the array size already used in the ALLOCATE statement:

```
EXEC SQL FOR 20 FETCH c1 USING DESCRIPTOR 'out' END-EXEC.
```

Subsequent GET statements for the same descriptor, that get DATA, INDICATOR, or RETURNED_LENGTH values, must use the same array size as the FETCH statement.

```
 01  VAL-DATA   OCCURS 20 TIMES  PIC S9(9) COMP.
 01  VAL-INDI   OCCURS 20 TIMES  PIC S9(4) COMP.
...
```

```
    EXEC SQL FOR 20 GET DESCRIPTOR 'out' VALUE 1 :VAL-DATA = DATA,
      :VAL-INDI = INDICATOR
    END-EXEC.
```

However, GET statements that reference other items which do not vary from row to row, such as LENGTH, TYPE and COUNT, must *not* use the FOR clause:

```
 01  CNT  PIC S9(9) COMP.
 01  LEN  PIC S9(9) COMP.
...
    EXEC SQL GET DESCRIPTOR 'out'   :CNT = COUNT END-EXEC.
    EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :LEN = LENGTH END-EXEC.
```

The same holds true for SET statements with reference semantics. SET statements which precede the FETCH and employ reference semantics for DATA, INDICATOR, or RETURNED_LENGTH must have the same array size as the FETCH:

```
 ...
 01  REF-DATA   OCCURS 20 TIMES  PIC S9(9) COMP.
 01  REF-INDI   OCCURS 20 TIMES  PIC S9(4) COMP.
...
    EXEC SQL FOR 20 SET DESCRIPTOR 'out' VALUE 1 REF DATA = :REF-DATA,
       REF INDICATOR = :REF-INDI END-EXEC.
```

Similarly, for a descriptor that is used for input, to insert a batch of rows, for instance, the EXECUTE or OPEN statement must use an array size equal to or less than the size used in the ALLOCATE statement. The SET statement, for both value and reference semantics, that accesses DATA, INDICATOR, or RETURNED_ LENGTH must use the same array size as in the EXECUTE statement.

The FOR clause is never used on the DEALLOCATE or PREPARE statements.

The following code sample illustrates a bulk operation with no output descriptor (there is no output, only input to be inserted into the table `emp`). The value of `CNT` is 2 (there are two host variables, `ENAME and EMPNO`, in the INSERT statement). The data table `ENAME-TABLE` holds three character strings: `Tom`, `Dick` and `Harry`, in that order. Their employee numbers are in the table `EMPNO-TABLE`. The indicator table `ENAME-IND` has a value of -1 for the second element; so a `NULL` will be inserted instead of `Dick`.

```
 01  DYN-STATEMENT  PIC X(240) value
        "INSERT INTO EMP(ENAME, EMPNO) VALUES (:ENAME,:EMPNO)".
 01  ARRAY-SIZE PIC S9(9) COMP   VALUE 3.
 01  ENAME-VALUES.
      05 FILLER PIC X(6)   VALUE "Tom   ".
      05 FILLER PIC X(6)   VALUE "Dick  ".
```

```
     05 FILLER PIC X(6)   VALUE "Harry ".
 01  ENAME-TABLE REDEFINES ENAME-VALUES.
     05 ENAME  PIC X(6)OCCURS 3 TIMES.
 01  ENAME-IND  PIC S9(4) COMPOCCURS 3 TIMES.
 01  ENAME-LEN  PIC S9(9) COMP   VALUE   6.
 01  ENAME-TYP  PIC S9(9) COMP   VALUE  96.
 01  EMPNO-VALUES.
     05 FILLER PIC S9(9) COMP   VALUE 8001.
     05 FILLER PIC S9(9) COMP   VALUE 8002.
     05 FILLER PIC S9(9) COMP   VALUE 8003.
 01  EMPNO-TABLE REDEFINES EMPNO-VALUES.
     05 EMPNO  PIC S9(9) DISPLAY SIGN LEADING  OCCURS 3 TIMES.
 01  EMPNO-LEN  PIC S9(9) COMP   VALUE   4.
 01  EMPNO-TYP  PIC S9(9) COMP   VALUE   3.
 01  CNT        PIC S9(9) COMP   VALUE   2.
........
     EXEC SQL FOR :ARRAY-SIZE ALLOCATE DESCRIPTOR 'in' END-EXEC.
     EXEC SQL PREPARE S FROM :DYN-STATEMENT  END-EXEC.
     MOVE 0 TO ENAME-IND(1).
     MOVE -1 TO ENAME-IND(2).
     MOVE 0 TO ENAME-IND(3).
     EXEC SQL SET DESCRIPTOR 'in' COUNT = :CNT END-EXEC.
     EXEC SQL SET DESCRIPTOR 'in' VALUE 1
        TYPE = :ENAME-TYP, LENGTH =:ENAME-LEN
     END-EXEC.
     EXEC SQL FOR :ARRAY-SIZE  SET DESCRIPTOR 'in' VALUE 1
        DATA = :ENAME, INDICATOR = :ENAME-IND
     END-EXEC.
     EXEC SQL SET DESCRIPTOR 'in' VALUE 2
        TYPE = :EMPNO-TYP, LENGTH =:EMPNO-LEN
     END-EXEC.
     EXEC SQL FOR :ARRAY-SIZE  SET DESCRIPTOR 'in' VALUE 2
        DATA = :EMPNO
     END-EXEC.
     EXEC SQL FOR :ARRAY-SIZE EXECUTE S
        USING DESCRIPTOR 'in' END-EXEC.
  ...
```

The preceding code inserts these values into the table:

```
EMPNO   ENAME
8001    Tom
8002
8003    Harry
```

See the discussion in "The FOR Clause" on page 7-16 for restrictions and cautions.

ANSI Dynamic SQL Precompiler Options

The macro option MODE (See "MODE" on page 14-30) sets ANSI compatibility characteristics and controls a number of functions. It can have the values ANSI or ORACLE. For individual functions there are micro options that override the MODE setting.

The precompiler micro option *DYNAMIC* specifies the descriptor behavior in dynamic SQL. The precompiler micro option *TYPE_CODE* specifies whether ANSI or Oracle datatype codes are to be used.

When the macro option MODE is set to ANSI, the micro option DYNAMIC becomes ANSI automatically. When MODE is set to ORACLE, DYNAMIC becomes ORACLE.

DYNAMIC and TYPE_CODE cannot be used inline.

The following table describes how the DYNAMIC setting affects various functionality:

Table 10–2 DYNAMIC Option Settings

Function	DYNAMIC=ANSI	DYNAMIC=ORACLE
Descriptor creation.	Must use ALLOCATE statement.	Must use an Oracle format descriptor.
Descriptor destruction.	May use DEALLOCATE statement.	N / A
Retrieving data.	May use both FETCH and GET statements.	Must use only FETCH statement.
Setting input data.	May use DESCRIBE INPUT statement. Must use SET statement.	Must set descriptor values in code. Must use DESCRIBE BIND VARIABLES statement.
Descriptor representation.	Single quoted literal or host identifier which contains the descriptor name.	Host variable, a pointer to SQLDA.
Data types available.	All ANSI types except BIT and all Oracle types.	Oracle types except objects, LOBs, and cursor variables.

The micro option TYPE_CODE is set by the precompiler to the same setting as the macro option MODE. TYPE_CODE can only equal ANSI if DYNAMIC equals ANSI.

The following table shows the functionality corresponding to the TYPE_CODE settings.

Table 10–3 TYPE_CODE Option Settings

Function	TYPE_CODE=ANSI	TYPE_CODE=ORACLE
Data type code numbers input and returned in dynamic SQL.	Use ANSI code numbers when ANSI type exists. Otherwise, use the negative of the Oracle code number. Only valid when DYNAMIC=ANSI.	Use Oracle code numbers. May be used regardless of the setting of DYNAMIC.

Full Syntax of the Dynamic SQL Statements

For more details on all these statements, see the alphabetical listing in the appendix Appendix F, "Embedded SQL Statements and Precompiler Directives".

ALLOCATE DESCRIPTOR

This statement is used only for ANSI dynamic SQL.

Purpose

Use this statement to allocate a SQL descriptor area. Supply a descriptor, the maximum number of occurrences of host bind items, and an array size.

Syntax

```
EXEC SQL [FOR [:]array_size] ALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal} [WITH MAX occurrences]
END-EXEC.
```

Variables

A number of variables can be used with the ALLOCATE descriptor. These include: array_size, desc_nam, and occurrences.

array_size

The optional array_size clause (an Oracle extension) supports table processing. It tells the precompiler that the descriptor is usable for table processing.

```
GLOBAL | LOCAL
```

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

desc_nam

The desc_nam variable defines the local descriptors that must be unique in the module. A runtime error is generated if the descriptor has been previously allocated, but not deallocated. A global descriptor must be unique for the application or a runtime error results.

occurrences

The optional occurrences clause is the maximum number of host variables possible in the descriptor. The occurrences variable must be an integer constant between 0 and 64K, or an error is returned. The default is 100. A precompiler error results if it does not conform to these rules.

Examples

```
EXEC SQL ALLOCATE DESCRIPTOR 'SELDES' WITH MAX 50 END-EXEC.

EXEC SQL FOR :BATCH ALLOCATE DESCRIPTOR GLOBAL :BINDDES WITH MAX 25
   END-EXEC.
```

DEALLOCATE DESCRIPTOR

Purpose

To free memory, use the deallocate statement. This statement deallocates a previously allocated SQL descriptor area.

Syntax

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
END-EXEC.
```

desc_nam

The only variable available with the deallocate descriptor is desc_nam (for descriptor name.) A runtime error results when a descriptor with the same name and scope has not been allocated, or has already been allocated and deallocated.

Examples

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' END-EXEC.

EXEC SQL DEALLOCATE DESCRIPTOR :BINDDES END-EXEC.
```

GET DESCRIPTOR

Purpose

Use to obtain information from a SQL descriptor area.

Syntax

```
EXEC SQL [FOR [:]array_size] GET DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
   {:hv0  = COUNT | VALUE item_number :hv1 = item_name1
   [{, :hvN = item_nameN }]}
END-EXEC.
```

Variables

array_size

The `FOR array_size` variable is an optional Oracle extension. `FOR array_size` has to be equal to the `array_size` field in the FETCH statement.

desc_nam

The descriptor name.

GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

COUNT

The total number of bind variables.

VALUE item_number

The position of the item in the SQL statement. `item_number` can be a variable or a constant. If `item_number` is greater than `COUNT`, the "no data found" condition is returned. `item_number` must be greater than 0.

hv1 .. hvN

These are host variables to which values are transferred.

item_name1 .. item_nameN

The descriptor item names corresponding to the host variables. The possible ANSI descriptor item names are listed in the following table.

Table 10–4 Definitions of Descriptor Item Names

Descriptor Item Name	Meaning
TYPE	See Table 10–1 on page 10-4 for the ANSI type codes. See Table 11–2 on page 11-16 for the Oracle type codes. Use the negative value of the Oracle code if the ANSI datatype is not in the table, and TYPE_CODE = ANSI.
LENGTH	Length of data in the column. In characters for NCHAR, and otherwise in bytes. Set by the DESCRIBE OUTPUT.
OCTET_LENGTH	Length of data in bytes.
RETURNED_LENGTH	The actual data length after a FETCH. It is undefined for fixed-length character types.
RETURNED_OCTET_LENGTH	Length of the returned data in bytes.
PRECISION	The number of digits.
SCALE	For exact numeric types, the number of digits to the right of the decimal point.
NULLABLE	If 1, the column can have NULL values. If 0, the column cannot have NULL values.
INDICATOR	The associated indicator value.
DATA	The data value.
NAME	Column name.
CHARACTER_SET_NAME	Column's character set.

The following table lists the Oracle extensions to the descriptor item names.

Table 10–5 Oracle Extensions to Definitions of Descriptor Item Names

Descriptor Item Name	Meaning
NATIONAL_CHARACTER	If 2, NCHAR or NVARCHAR2. If 1, character. If 0, non-character.
INTERNAL_LENGTH	The internal length, in bytes.

Usage Notes

Use the FOR clause in GET DESCRIPTOR statements which contain DATA, INDICATOR, and RETURNED_LENGTH items only.

The internal type is provided by the DESCRIBE OUTPUT statement. For both input and output, you must set the type to be the external type of your host variable. TYPE is the Oracle or ANSI code in Table 10–1 on page 10-4. You will receive the negative value of the Oracle type code if the ANSI type is not in the table.

LENGTH contains the column length in characters for fields that have fixed-width National Language character sets. It is in bytes for other character columns. It is set in DESCRIBE OUTPUT.

RETURNED_LENGTH is the actual data length set by the FETCH statement. It is in bytes or characters as described for LENGTH. The fields OCTET_LENGTH and RETURNED_OCTET_LENGTH are the lengths in bytes.

NULLABLE = 1 means that the column can have NULLS; NULLABLE = 0 means it cannot.

CHARACTER_SET_NAME only has meaning for character columns. For other types, it is undefined. The DESCRIBE OUTPUT statement obtains the value.

DATA and INDICATOR are the data value and the indicator status for that column. If data = NULL, but the indicator was not requested, an error is generated at runtime ("DATA EXCEPTION, NULL VALUE, NO INDICATOR PARAMETER").

Oracle-Specific Descriptor Item Names

NATIONAL_CHARACTER = 2 if the column is an NCHAR or NVARCHAR2 column. If the column is a character (but not National Character) column, this item is set to 1. If a non-character column, this item becomes 0 after DESCRIBE OUTPUT is executed.

INTERNAL_LENGTH is for compatibility with Oracle dynamic Method 4. It has the same value as the length member of the Oracle descriptor area. See "Oracle Dynamic SQL: Method 4" on page 11-1.

Examples

```
EXEC SQL GET DESCRIPTOR :BINDDES :COUNT = COUNT END-EXEC.

EXEC SQL GET DESCRIPTOR 'SELDES' VALUE 1 :TYP = TYPE, :LEN = LENGTH
   END-EXEC.

EXEC SQL FOR :BATCH GET DESCRIPTOR LOCAL 'SELDES'
   VALUE :SEL-ITEM-NO :IND = INDICATOR, :DAT = DATA END-EXEC.
```

SET DESCRIPTOR

Purpose

Use this statement to set information in the descriptor area from host variables. The SET DESCRIPTOR statement supports *only* host variables for the item names.

Syntax

```
EXEC SQL [FOR [:]array_size] SET DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
   {COUNT = :hv0 | VALUE item_number
   [REF] item_name1 = :hv1
   [{, [REF] item_nameN = :hvN}]}
END-EXEC.
```

Variables

array_size

This optional Oracle clause permits using arrays when setting the descriptor items DATA, INDICATOR, and RETURNED_LENGTH only. You cannot use other items in a SET DESCRIPTOR that contains the FOR clause. All host variable table sizes must match. Use the same array size for the SET statement that you use for the FETCH statement.

desc_nam

The descriptor name. It follows the rules in ALLOCATE DESCRIPTOR.

COUNT

The number of bind (input) or define (output) variables.

VALUE item_number

Position in the dynamic SQL statement of a host variable.

hv1 .. hvN

The host variables (not constants) that you set.

item_name1 .. item_nameN

In a similar way to the GET DESCRIPTOR syntax (see "GET DESCRIPTOR" on page 10-15), `item_name` can take on these values:

Table 10–6 Descriptor Item Names for SET DESCRIPTOR

Descriptor Item Name	Meaning
TYPE	See Table 10–1 on page 10-4 for the ANSI type codes. See Table 11–2 on page 11-16 for the Oracle type codes. Use the negative value of the Oracle type code if the Oracle type is not in the table, and TYPE_CODE = ANSI.
LENGTH	Maximum length of data in the column.
PRECISION	The number of digits.
SCALE	For exact numeric types, the number of bytes to the right of the decimal point.
Descriptor Item Name	**Meaning**
INDICATOR	The associated indicator value. Set for reference semantics.
DATA	Value of the data to be set. Set for reference semantics.
CHARACTER_SET_NAME	Column's character set.

The Oracle extensions to the descriptor item names are listed in the following table.

Table 10–7 Oracle Extensions to Descriptor Item Names for SET DESCRIPTOR

Descriptor Item Name	Meaning
RETURNED_LENGTH	Length returned after a FETCH. Set if reference semantics is being used.

Table 10–7 Oracle Extensions to Descriptor Item Names for SET DESCRIPTOR

NATIONAL_CHARACTER	Set to 2 when the input host variable is an NCHAR or NVARCHAR2 type.

Usage Notes

Reference semantics is another optional Oracle extension that speeds performance. Use the keyword REF before these descriptor items names only: DATA, INDICATOR, RETURNED_LENGTH. When you use the REF keyword you do not need to use a GET statement. Complex data types and DML returning clauses require the REF form of SET DESCRIPTOR. See "DML Returning Clause" on page 5-9.

When REF is used the associated host variable itself is used in the SET. The GET is not needed in this case. The RETURNED_LENGTH can only be set when you use the REF semantics, not the value semantics.

Use the same array size for the SET or GET statements that you use in the FETCH.

Set the NATIONAL_CHAR field to 2 for NCHAR host input values.

When setting an object type's characteristics, you must set USER_DEFINED_TYPE_NAME and USER_DEFINED_TYPE_NAME_LENGTH.

If omitted, USER_DEFINED_TYPE_SCHEMA and USER_DEFINED_TYPE_SCHEMA_LENGTH default to the current connection.

Example

Bulk table examples are found in "Using Tables for Bulk Operations" on page 10-9.

```
 ...
 01  BINDNO  PIC S9(9) COMP  VALUE 2.
 01  INDI    PIC S9(4) COMP  VALUE -1.
 01  DATA    PIC X(6)  COMP  VALUE "ignore".
 01  BATCH   PIC S9(9) COMP  VALUE 1.
 ...
     EXEC SQL FOR :batch ALLOCATE DESCRIPTOR :BINDDES END-EXEC.
     EXEC SQL SET DESCRIPTOR GLOBAL :BINDDES COUNT = 3 END-EXEC.
     EXEC SQL FOR :batch SET DESCRIPTOR :BINDDES
        VALUE :BINDNO INDICATOR = :INDI, DATA = :DATA END-EXEC.
...
```

Use of PREPARE

Purpose

The PREPARE statement used in this method is the same as the PREPARE statement used in the Oracle dynamic SQL methods. An Oracle extension allows a quoted string for the SQL statement, as well as a variable.

Syntax

```
EXEC SQL PREPARE statement_id FROM :sql_statement END-EXEC.
```

Variables

statement_id

This must not be declared; it is an undeclared SQL identifier associated with the prepared SQL statement.

sql_statement

A character string (a constant or a variable) holding the embedded SQL statement.

Example

```
...
01  STATEMENT    PIC X(255)
       VALUE "SELECT ENAME FROM emp WHERE deptno = :d".
...
    EXEC SQL PREPARE S1 FROM :STATEMENT END-EXEC.
```

DESCRIBE INPUT

Purpose

This statement returns information about the input bind variables.

Syntax

```
EXEC SQL DESCRIBE INPUT statement_id USING [SQL] DESCRIPTOR
   [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

Variables

statement_id

The same as used in PREPARE and DESCRIBE OUTPUT. This must not be declared; it is a SQL identifier.

GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

desc_nam

The descriptor name.

Usage Notes

Only COUNT and NAME are implemented for bind variables in this version.

Examples

```
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR GLOBAL :BINDDES END-EXEC.
EXEC SQL DESCRIBE INPUT S2 USING DESCRIPTOR 'input' END-EXEC.
```

DESCRIBE OUTPUT

Purpose

The DESCRIBE INPUT statement is used to obtain information about the columns in a PREPARE statement. The ANSI syntax differs from the older syntax. The information which is stored in the SQL descriptor area is the number of values returned and associated information such as type, length, and name.

Syntax

```
EXEC SQL DESCRIBE [OUTPUT] statement_id USING [SQL] DESCRIPTOR
   [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

Variables

statement_id

The statement_id is a SQL identifier. It must not be declared.

GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

desc_nam

The descriptor name. Either a host variable preceded by a ':', or a single-quoted string. OUTPUT is the default and can be omitted.

Examples

```
 ...
 01  DESNAME   PIC X(10) VALUE "SELDES".
 ...
     EXEC SQL DESCRIBE S1 USING SQL DESCRIPTOR 'SELDES' END-EXEC.
* Or:
     EXEC SQL DESCRIBE OUTPUT S1 USING DESCRIPTOR :DESNAME END-EXEC.
```

EXECUTE

Purpose

EXECUTE matches input and output variables in a prepared SQL statement and then executes the statement. This ANSI version of EXECUTE differs from the older EXECUTE statement by allowing two descriptors in one statement to support DML RETURNING.

Syntax

```
EXEC SQL [FOR [:]array_size] EXECUTE statement_id
   [USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}]
   [INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}]
END-EXEC.
```

Variables

array_size

The number of rows the statement will process.

statement_id

The same as used in PREPARE. This must not be declared; it is a SQL identifier. It can be a literal.

GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

desc_nam

The descriptor name. Either a host variable preceded by a ':', or a single-quoted string.

Usage Notes

The INTO clause implements the RETURNING clause for INSERT, UPDATE and DELETE (See "Inserting Rows" on page 5-9 and succeeding pages).

Examples

```
EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR GLOBAL :BINDDES END-EXEC.

EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES'
   END-EXEC.
```

Use of EXECUTE IMMEDIATE

Purpose

The EXECUTE IMMEDIATE statement executes a literal or host variable character string containing the SQL statement.The ANSI SQL form of this statement is the same as in the older dynamic SQL:

Syntax

```
EXEC SQL EXECUTE IMMEDIATE [:]sql_statement END-EXEC.
```

Variable

Only one variable is available with the EXECUTE IMMEDIATE statement.

sql_statement

The sql_statement variable is the SQL statement or PL/SQL block in a character string. It can be a host variable or a literal.

Example

```
EXEC SQL EXECUTE IMMEDIATE :statement END-EXEC.
```

Use of DYNAMIC DECLARE CURSOR

Purpose

The DYNAMIC DECLARE CURSOR statement declares a cursor that is associated with a statement which is a query. This is a form of the generic Declare Cursor statement.

Syntax

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_id END-EXEC.
```

Variables

cursor_name

A cursor variable (a SQL identifier, not a host variable).

statement_id

An undeclared SQL identifier (the same as the one used in the PREPARE statement).

Example

```
EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.
```

OPEN Cursor

Purpose

The OPEN statement associates input parameters with a cursor and then opens the cursor.

Syntax

```
EXEC SQL [FOR [:]array_size] OPEN dyn_cursor
   [[USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] desc_nam1]
   [INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] desc_nam2] ]
END-EXEC.
```

Variables

array_size

This limit is less than or equal to number specified when the descriptor was allocated.

GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

dyn_cursor

The cursor variable.

desc_nam1, desc_nam2

The optional descriptor names.

Usage notes

If the prepared statement associated with the cursor contains colons or question marks, a USING clause must be specified, or an error results at runtime. The INTO clause supports DML RETURNING (See "Inserting Rows" on page 5-9 and succeeding sections on DELETE and UPDATE).

Examples

```
EXEC SQL OPEN C1 USING SQL DESCRIPTOR :BINDDES END-EXEC.

EXEC SQL FOR :LIMIT OPEN C2 USING DESCRIPTOR :B1, :B2
  INTO SQL DESCRIPTOR :SELDES
END-EXEC.
```

FETCH

Purpose

The FETCH statement fetches a row for a cursor declared with a dynamic DECLARE statement.

Syntax

```
EXEC SQL [FOR [:]array_size] FETCH cursor INTO [SQL] DESCRIPTOR
  [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

Variables

array_size

The number of rows the statement will process.

cursor

The dynamic cursor that was previously declared.

GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

desc_nam

Descriptor name.

Usage Notes

The optional `array_size` in the FOR clause must be less than or equal to the number specified in the ALLOCATE DESCRIPTOR statement.

RETURNED_LENGTH is undefined for fixed-length character types.

Examples

```
EXEC SQL FETCH FROM C1 INTO DESCRIPTOR 'SELDES' END-EXEC.

EXEC SQL FOR :ARSZ FETCH C2 INTO DESCRIPTOR :DESC END-EXEC.
```

CLOSE a Dynamic Cursor

Purpose

The CLOSE statement closes a dynamic cursor. Its syntax is identical to the Oracle Method 4.

Syntax

```
EXEC SQL CLOSE cursor END-EXEC.
```

Variable

Only one variable is available with the CLOSE statement.

cursor

The cursor variable describes the previously declared dynamic cursor.

Example

```
EXEC SQL CLOSE C1 END-EXEC.
```

Differences From Oracle Dynamic Method 4

The ANSI dynamic SQL interface supports all the features supported by the Oracle dynamic Method 4, with these additions:

- All datatypes, including cursor variables, and LOB types are supported by ANSI Dynamic SQL.
- The ANSI mode uses an internal *SQL descriptor area* which is an expansion of the external SQLDA used in Oracle older dynamic Method 4 to store its input and output information.
- New embedded SQL statements are introduced: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, GET DESCRIPTOR, and SET DESCRIPTOR.
- The DESCRIBE statement does not return the names of indicator variables in ANSI Dynamic SQL.
- ANSI Dynamic SQL does not allow you to specify the maximum size of the returned column name or expression. The default size is set at 128.

- The descriptor name must be either an identifier in single-quotes or a host variable preceded by a colon.
- For output, the optional SELECT LIST FOR clause in the DESCRIBE is replaced by the optional keyword OUTPUT. The INTO clause is replaced by the USING DESCRIPTOR clause, which can contain the optional keyword SQL.
- For input, the optional BIND VARIABLES FOR clause of the DESCRIBE can be replaced by the keyword INPUT. The INTO clause is replaced by the USING DESCRIPTOR clause, which can contain the optional keyword SQL.
- The optional keyword SQL can come before the keyword DESCRIPTOR in the USING clause of the EXECUTE, FETCH and OPEN statements.

Restrictions

Restrictions in effect on ANSI dynamic SQL are:

- You cannot mix the two dynamic methods in the same module.
- The precompiler option DYNAMIC must be set to ANSI. The precompiler option TYPE_CODE can be set to ANSI only if DYNAMIC is set to ANSI.
- The SET statement supports only host variables as item names.

Sample Programs: SAMPLE12.PCO

The following ANSI SQL dynamic Method 4 program, SAMPLE12.PCO, is found in the demo directory. SAMPLE12 mimics SQL*Plus by prompting for SQL statements to be input by the user. Read the comments at the beginning for details of the program flow.

```
*****************************************************************
* Sample Program 12: Dynamic SQL Method 4 using ANSI Dynamic SQL *
*                                                               *
* This program shows the basic steps required to use dynamic    *
* SQL Method 4 with ANSI Dynamic SQL.  After logging on to      *
* ORACLE, the program prompts the user for a SQL statement,     *
* PREPAREs the statement, DECLAREs a cursor, checks for any     *
* bind variables using DESCRIBE INPUT, 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.                                       *
* use option dynamic=ansi when precompiling this sample.        *
*****************************************************************
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  ANSIDYNSQL4.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

01  USERNAME       PIC X(20).
01  PASSWD         PIC X(20).
01  BDSC           PIC X(6) VALUE "BNDDSC".
01  SDSC           PIC X(6) VALUE "SELDSC".
01  BNDCNT         PIC S9(9) COMP.
01  SELCNT         PIC S9(9) COMP.
01  BNDNAME        PIC X(80).
01  BNDVAL         PIC X(80).
01  SELNAME        PIC X(80) VARYING.
01  SELDATA        PIC X(80).
01  SELTYP         PIC S9(4) COMP.
01  SELPREC        PIC S9(4) COMP.
01  SELLEN         PIC S9(4) COMP.
01  SELIND         PIC S9(4) COMP.
01  DYN-STATEMENT  PIC X(80).
01  BND-INDEX      PIC S9(9) COMP.
01  SEL-INDEX      PIC S9(9) COMP.
01  VARCHAR2-TYP   PIC S9(4) COMP VALUE 1.
01  VAR-COUNT      PIC 9(2).
01  ROW-COUNT      PIC 9(4).
01  NO-MORE-DATA   PIC X(1) VALUE "N".
01  TMPLEN         PIC S9(9) COMP.
01  MAX-LENGTH     PIC S9(9) COMP VALUE 80.

    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.
```

```
*    ALLOCATE THE BIND AND SELECT DESCRIPTORS.

     EXEC SQL ALLOCATE DESCRIPTOR :BDSC WITH MAX 20 END-EXEC.
     EXEC SQL ALLOCATE DESCRIPTOR :SDSC WITH MAX 20 END-EXEC.

*    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 BIND VARIABLES.

     EXEC SQL DESCRIBE INPUT S1 USING DESCRIPTOR :BDSC END-EXEC.

     EXEC SQL GET DESCRIPTOR :BDSC :BNDCNT = COUNT END-EXEC.

     IF BNDCNT < 0
         DISPLAY "TOO MANY BIND VARIABLES."
         GO TO END-SQL
     ELSE
         DISPLAY "NUMBER OF BIND VARIABLES: " WITH NO ADVANCING
         MOVE BNDCNT TO VAR-COUNT
         DISPLAY VAR-COUNT
*        EXEC SQL SET DESCRIPTOR :BDSC COUNT = :BNDCNT END-EXEC
     END-IF.

     IF BNDCNT = 0
         GO TO DESCRIBE-ITEMS.
     PERFORM SET-BND-DSC
             VARYING BND-INDEX FROM 1 BY 1
             UNTIL BND-INDEX > BNDCNT.

*    OPEN THE CURSOR AND DESCRIBE THE SELECT-LIST ITEMS.

 DESCRIBE-ITEMS.
     EXEC SQL  OPEN C1 USING DESCRIPTOR :BDSC END-EXEC.

     EXEC SQL  DESCRIBE OUTPUT S1 USING DESCRIPTOR :SDSC  END-EXEC.
```

```
           EXEC SQL GET DESCRIPTOR :SDSC :SELCNT = COUNT END-EXEC.

           IF SELCNT < 0
               DISPLAY "TOO MANY SELECT-LIST ITEMS."
               GO TO END-SQL
           ELSE
               DISPLAY "NUMBER OF SELECT-LIST ITEMS: "
                   WITH NO ADVANCING
               MOVE SELCNT TO VAR-COUNT
               DISPLAY VAR-COUNT
               DISPLAY " "
       *       EXEC SQL SET DESCRIPTOR :SDSC COUNT = :SELCNT END-EXEC
           END-IF.

       *   SET THE INPUT DESCRIPTOR

           IF SELCNT > 0
               PERFORM SET-SEL-DSC
                   VARYING SEL-INDEX FROM 1 BY 1
                   UNTIL SEL-INDEX > SELCNT
               DISPLAY " ".

       *   FETCH EACH ROW AND PRINT EACH SELECT-LIST VALUE.

           IF SELCNT > 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 DEALLOCATE DESCRIPTOR :BDSC END-EXEC.
           EXEC SQL DEALLOCATE DESCRIPTOR :SDSC END-EXEC.
           EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
           DISPLAY " ".
           DISPLAY "HAVE A GOOD DAY!".
           DISPLAY " ".
           STOP RUN.

       *   DISPLAY ORACLE ERROR MESSAGE AND CODE.
```

```
 SQL-ERROR.
     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:

*    SET A BIND-LIST ELEMENT'S ATTRIBUTE
*    LET THE USER FILL IN THE BIND VARIABLES AND
*    REPLACE THE 0S DESCRIBED INTO THE DATATYPE FIELDS OF THE
*    BIND DESCRIPTOR WITH 1S TO AVOID AN "INVALID DATATYPE"
*    ORACLE ERROR
 SET-BND-DSC.
         EXEC SQL GET DESCRIPTOR :BDSC VALUE
                :BND-INDEX :BNDNAME = NAME END-EXEC.
         DISPLAY "ENTER VALUE FOR ", BNDNAME.
         ACCEPT BNDVAL.
         EXEC SQL SET DESCRIPTOR :BDSC VALUE :BND-INDEX
           TYPE = :VARCHAR2-TYP, LENGTH = :MAX-LENGTH,
           DATA = :BNDVAL END-EXEC.

* SET A SELECT-LIST ELEMENT'S ATTRIBUTES
 SET-SEL-DSC.
     MOVE SPACES TO SELNAME-ARR.
     EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
           :SELNAME = NAME, :SELTYP = TYPE,
           :SELPREC = PRECISION, :SELLEN = LENGTH END-EXEC.

*    IF DATATYPE IS DATE, LENGTHEN TO 9 CHARACTERS.
     IF SELTYP = 12
         MOVE 9 TO SELLEN.

*    IF DATATYPE IS NUMBER, SET LENGTH TO PRECISION.
     MOVE 0 TO TMPLEN.
     IF SELTYP = 2 AND SELPREC = 0
         MOVE 40 TO TMPLEN.
     IF SELTYP = 2 AND SELPREC > 0
         ADD 2 TO SELPREC
         MOVE SELPREC TO TMPLEN.

     IF SELTYP = 2
```

```
            IF TMPLEN > MAX-LENGTH
                DISPLAY "COLUMN VALUE TOO LARGE FOR DATA BUFFER."
                GO TO END-SQL
            ELSE
                MOVE TMPLEN TO SELLEN.


      * COERCE DATATYPES TO VARCHAR2.
           MOVE 1 TO SELTYP.

      *    DISPLAY COLUMN HEADING.
           DISPLAY "  ", SELNAME-ARR(1:SELLEN) WITH NO ADVANCING.

           EXEC SQL SET DESCRIPTOR :SDSC VALUE :SEL-INDEX
              TYPE = :SELTYP, LENGTH = :SELLEN END-EXEC.

      * FETCH A ROW AND PRINT THE SELECT-LIST VALUE.

       FETCH-ROWS.
           EXEC SQL  FETCH C1 INTO DESCRIPTOR :SDSC END-EXEC.
           IF SQLCODE NOT = 0
               MOVE "Y" TO NO-MORE-DATA.
           IF SQLCODE = 0
               PERFORM PRINT-COLUMN-VALUES
                   VARYING SEL-INDEX FROM 1 BY 1
                   UNTIL SEL-INDEX > SELCNT
               DISPLAY " ".

      * PRINT A SELECT-LIST VALUE.

       PRINT-COLUMN-VALUES.
           MOVE SPACES TO SELDATA.
      *    returned length is not set for blank padded types
           IF SELTYP EQUALS 1
              EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
                :SELDATA = DATA, :SELIND = INDICATOR,
                :SELLEN = LENGTH END-EXEC
           ELSE
              EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
                 :SELDATA = DATA, :SELIND = INDICATOR,
                 :SELLEN = RETURNED_LENGTH END-EXEC.
           IF (SELIND = -1)
              move "  NULL" to SELDATA.

           DISPLAY SELDATA(1:SELLEN), "  "
```

```
                WITH NO ADVANCING.
```

11

Oracle Dynamic SQL: Method 4

This chapter shows you how to implement Oracle dynamic SQL Method 4, which lets your program accept or build dynamic SQL statements that contain a varying number of host variables.

New applications should be developed using the newer ANSI SQL Method 4 described in Chapter 10, "ANSI Dynamic SQL".The ANSI Method 4 supports all Oracle types, while the older Oracle Method 4 does *not* support cursor variables, tables of group items, the DML returning clause, and LOBs.

Subjects discussed include the following:

- Meeting the Special Requirements of Method 4
- Understanding the SQL Descriptor Area (SQLDA)
- The SQLDA Variables
- Prerequisite Knowledge
- 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 Oracle Method 4, see Chapter 9, "Oracle Dynamic SQL"

Meeting the Special Requirements of Method 4

Before looking into the requirements of Method 4, you should be familiar 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.

Advantages of Method 4

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 Oracle9*i* and COBOL types

To add this flexibility to your program, you must give the runtime library additional information.

Information the Database Needs

Pro*COBOL generates calls to Oracle9 for all executable dynamic SQL statements. If a dynamic SQL statement contains no select-list items or place-holders, the database 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, the database needs information about the program variables that will hold output or input values. Specifically, the database 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, the database needs the address of the corresponding output variable.

Where the Information is Stored

All the information the database 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 the database knows where to write output values and read input values.

How do values get stored in these data variables? A FETCH generates output values using a cursor, and input values are filled in by your program, typically from information entered interactively by the user.

How Information is Obtained

You use the DESCRIBE statement to help get the information the database 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. Therefore, 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 11-14.

Multiple SQLDAs

If your program has more than one active dynamic SQL statement, each statement must have its own SQLDA. 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 code them into your program using the sample select and bind SQLDAs shown in Figure 11–1. 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 11–1 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).
```

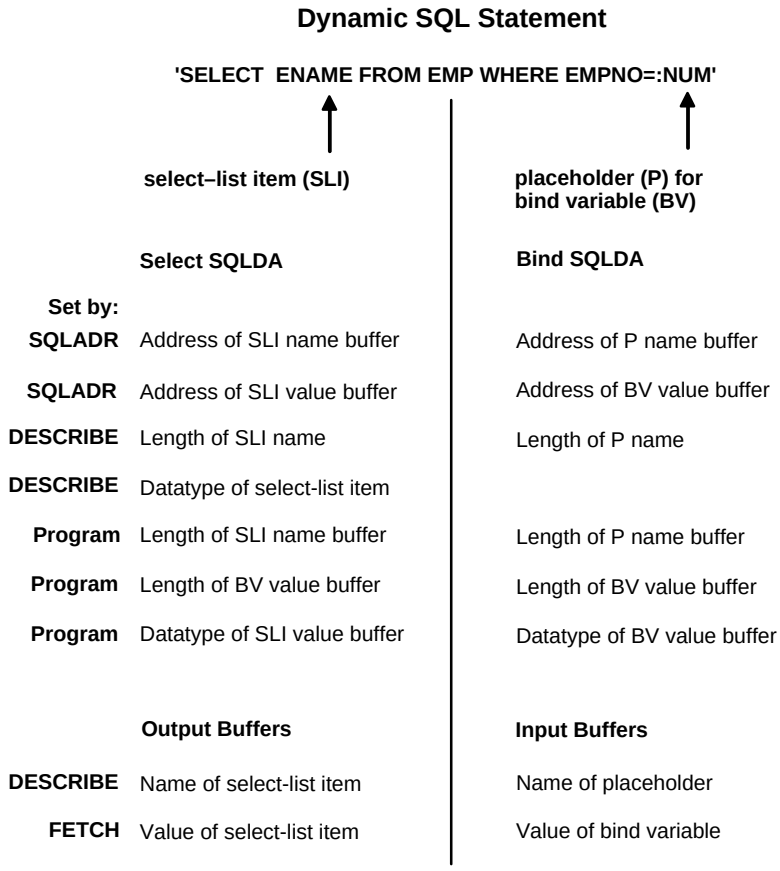
You can store the SQLDAs in files (named SELDSC and BNDDSC, for example),

and 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 11–2 shows whether variables are set by SQLADR calls, DESCRIBE commands, FETCH commands, or program assignments.

Figure 11–2 How Variables Are Set



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 included in DESCRIBE. 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 in the DESCRIBE, which is stored in SQLDFND.

SQLDFND

The SQLDFND variable 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

The **SELDV | BNDDV** table contains 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 the database to store FETCHed select-list values in the data buffers addressed by SELDV(1) through SELDV(SQLDNUM). Thus, the database stores the Jth select-list value in SEL-DV(J).

Bind Descriptors

You must set the bind descriptors before issuing the OPEN command. The following statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

directs Oracle9 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.) The database finds the Jth bind-variable value in BND-DV(J).

SELDFMT | BNDDFMT

The **SELDFMT | BNDDFMT** table contains the addresses of data buffers that store select-list or bind-variable conversion format strings. You can currently 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 11-20.

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

The **SELDVLN | BNDDV**LN table contains the lengths of select-list variables 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 SELDVLN(J) before the FETCH command.

The format of the length differs among 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 11-20.

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 SELDVLN(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 11-15.

Bind Descriptors

You must set the Bind Descriptor 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 Oracle9*i* 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 Oracle9*i* 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) are not used.

SELDVTYP | BNDDVTYP

The **SELDVTYP | BNDDVTYP** table contains the datatype codes of select-list or bind-variable values. These codes determine how Oracle9*i* data is converted when stored in the data buffers addressed by elements of SELDV. The datatype descriptor table is further described in "Converting Data" on page 11-15.

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 a FETCH is executed, you might want to reset some datatypes because the internal format of 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 Oracle9*i* to COBOL format. See "Coercing Datatypes" on page 11-18.

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. 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 11-21.

It is best to change the 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

The **SELDI | BNDDI** table contains 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 Oracle9*i* 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 Oracle9*i* 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

The **SELDH-VNAME | BNDDH-VNAME** table contains 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 Oracle9*i* 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, Oracle9*i* stores the Jth select-list or place-holder name in SEL-DH-VNAME(J) or BND-DH-VNAME(J).

Note: The **SELDH-VNAME | BNDDH-VNAME** table contains only the name of the column, and *not* the table-qualifier.column name, even if you provide it in your SQL statement. If, for example, you were to do a describe of select-list in the SQL statement `select a.owner from all_tables` the software will return `not a.owner`, but instead, `owner`. If necessary, you should use column aliases to correctly identify a column in the select list.

SELDH-MAX-VNAMEL | BNDDH-MAX-VNAMEL

The **SELDH-MAX-VNAMEL | BNDDH-MAX-VNAMEL** table contains 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

The **SELDH-CUR-VNAMEL | BNDDH-CUR-VNAMEL** table contains 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

The **SELDI-VNAME | BNDDI-VNAME** table contains 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. 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 Oracle9*i* to store any indicator-variable names in the data buffers addressed by BNDDI-VNAME(1) through BNDDI-VNAME(SQLDNUM). Thus, Oracle9*i* stores the Jth indicator-variable name in BND-DI-VNAME(J).

SELDI-MAX-VNAMEL | BNDDI-MAX-VNAMEL

The **SELDI-MAX-VNAMEL | BNDDI-MAX-VNAMEL** table contains 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. 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

The **SELDI-CUR-VNAMEL | BNDDI-CUR-VNAMEL** table contains the actual lengths of the names of the indicator variables. You can associate indicator-variable names only with bind variables. 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

The **SELDFCLP | BNDDFCLP** table is reserved for future use. It must be present because Oracle9*i* expects the group item SELDSC or BNDDSC to be a certain size. You must currently set the elements of SELDFCLP and BNDDFCLP to zero.

SELDFCRCP | BNDDFCRCP

The **SELDFCRCP | BNDDFCRCP** table is reserved for future use. It must be present because Oracle9*i* expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCRCP and BNDDFCRCP to zero.

Prerequisite Knowledge

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 Oracle9*i* 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. The next example uses 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 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 Oracle9*i* to store column values in database tables to represent pseudocolumn values.

When you issue a DESCRIBE SELECT LIST command, Oracle9*i* 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 11 shows the internal datatypes and their codes:

Table 11–1 Internal Datatypes and Related Codes

Internal Datatype	Code
VARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHAR	96

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. Therefore, you must reset the codes *before* issuing the OPEN command. The codes tell Oracle9*i* 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 external datatypes and their codes, as well as the corresponding COBOL datatypes:

Table 11–2 Oracle 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*)

Table 11–2 Oracle External and Related COBOL Datatypes

Name	Code	COBOL Datatype
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*)
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

Notes:

1. For use in an EXEC SQL VAR statement only.
2. Include the *n*-byte length field.

For more information about the datatypes and their formats, see "The Oracle9i 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 11–3 shows the predefined PL/SQL scalar datatypes and their internal datatype equivalence

Table 11–3 PL/SQL Datatype Equivalences with Internal Datatypes

PL/SQL Datatype	Oracle Internal Datatype
VARCHAR	VARCHAR2
VARCHAR2	
BINARY_INTEGER	NUMBER
DEC	
DECIMAL	
DOUBLE PRECISION	
FLOAT	
INT	
INTEGER	
NATURAL	
NUMBER	
NUMERIC	
POSITIVE	
REAL	
SMALLINT	
LONG	LONG
ROWID	ROWID
DATE	DATE
RAW	RAW
LONG RAW	LONG RAW
CHAR	CHAR
CHARACTER	
STRING	

Coercing Datatypes

For a select descriptor, DESCRIBE SELECT LIST can return any of the internal datatypes. Often, as in the case of character data, the internal datatype corresponds exactly to the external datatype you want to use. However, a few internal datatypes

map to external datatypes that can be difficult to handle. Thus, 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. Oracle9*i* does any necessary conversion between internal and external datatypes at FETCH time. 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 BNDDVTYP table of datatype codes to tell Oracle9*i* the external datatype of each bind variable. Oracle9*i* 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 Oracle9*i* the buffer length to use. To specify the length of the Jth select-list value, set SELDVLN(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, Oracle9*i* 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, Oracle9*i* 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 desired. Therefore, 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 a 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 this table:

Table 11–4 Datatype Exceptions to the SQLPR2 Procedure

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 NULL datatypes 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 integer 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 11-40, 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. A full-length example program illustrating Method 4 is seen at the end of this chapter. 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.
```

Alternatively, 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.
```

The next section illustrates 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, respectively.

Declare a Host String

Your program needs a host variable to store the text of the dynamic SQL statement. The host variable (SELECTSTMT in our example) must be declared as a character string:

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    ...
01  SELECTSTMT  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 hard-coding 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

You next 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 11–3 and Figure 11–4 represent the resulting descriptors.

Figure 11–3 Initialized Select Descriptor

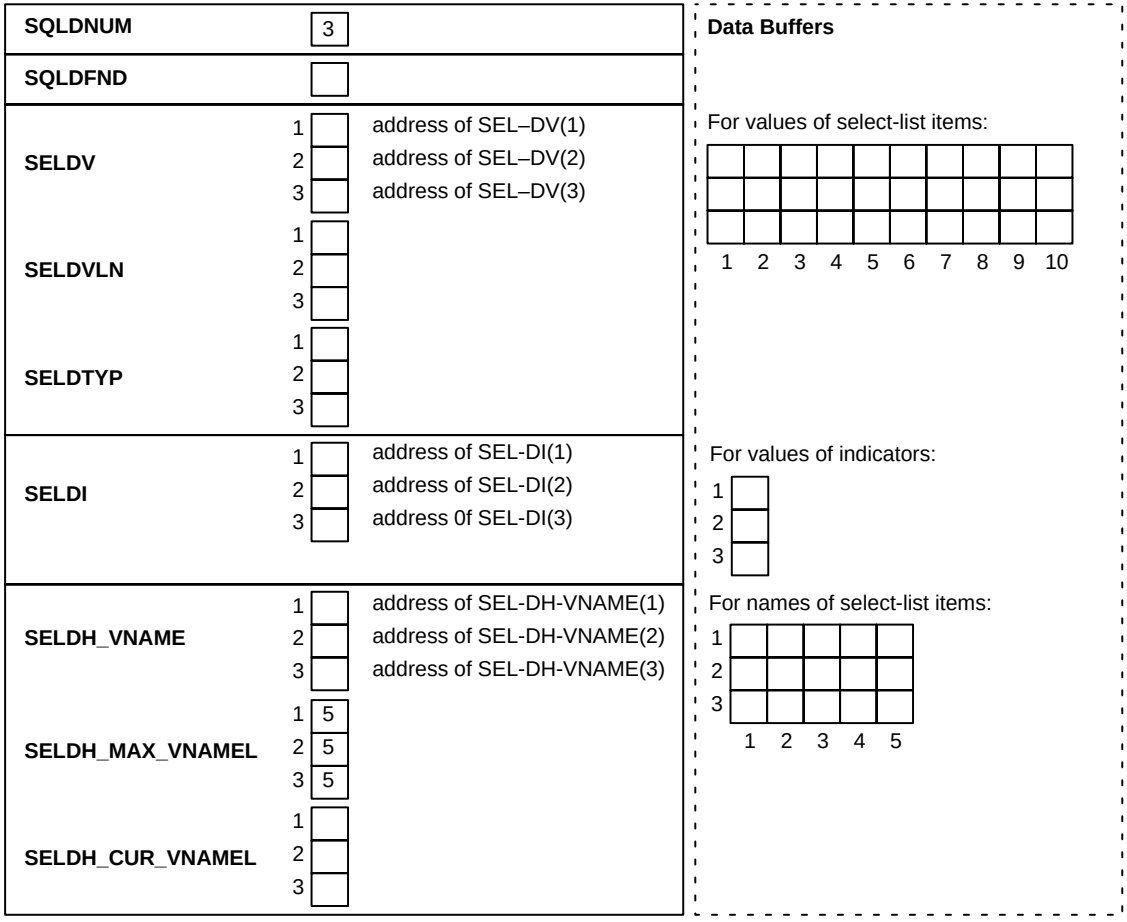
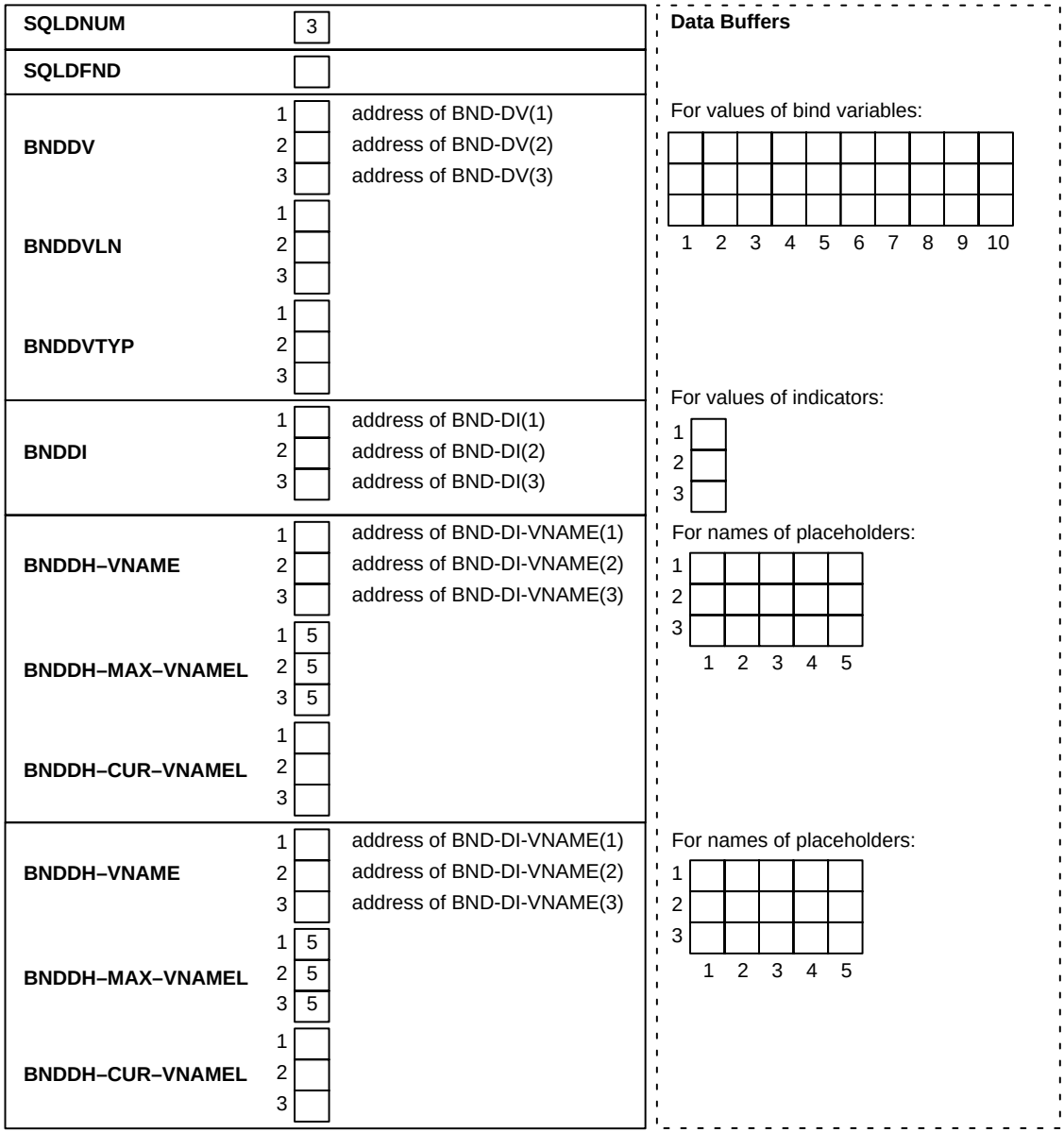


Figure 11–4 Initialized Bind Descriptor


SQLDNUM
3
SQLDFND
BNDDV
1 address of BND-DV(1)
2 address of BND-DV(2)
3 address of BND-DV(3)
BNDDVLN
BNDDVTYP
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-DI-VNAME(1)
2 address of BND-DI-VNAME(2)
3 address of BND-DI-VNAME(3)
BNDDH–MAX–VNAMEL
BNDDH–CUR–VNAMEL
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
BNDDH–CUR–VNAMEL
Data Buffers
For values of bind variables:
1 2 3 4 5 6 7 8 9 10
For values of indicators:
For names of placeholders:
1 2 3 4 5
For names of placeholders:
1 2 3 4 5

Store the Query Text in the Host String

Next, you prompt the user for a SQL statement, then store the input string in SELECTSTMT as follows:

```
DISPLAY "Enter a SELECT statement: " WITH NO ADVANCING.
ACCEPT SELECTSTMT.
```

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 SELECTSTMT and gives it the name SQLSTMT, as follows:

```
EXEC SQL PREPARE SQLSTMT FROM :SELECTSTMT 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, 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 replaces the static query. In our example, DECLARE CURSOR defines a cursor named EMPCURSOR and associates it with SQLSTMT, as follows:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR SQLSTMT 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 SQLSTMT
    INTO BNDDSC
```

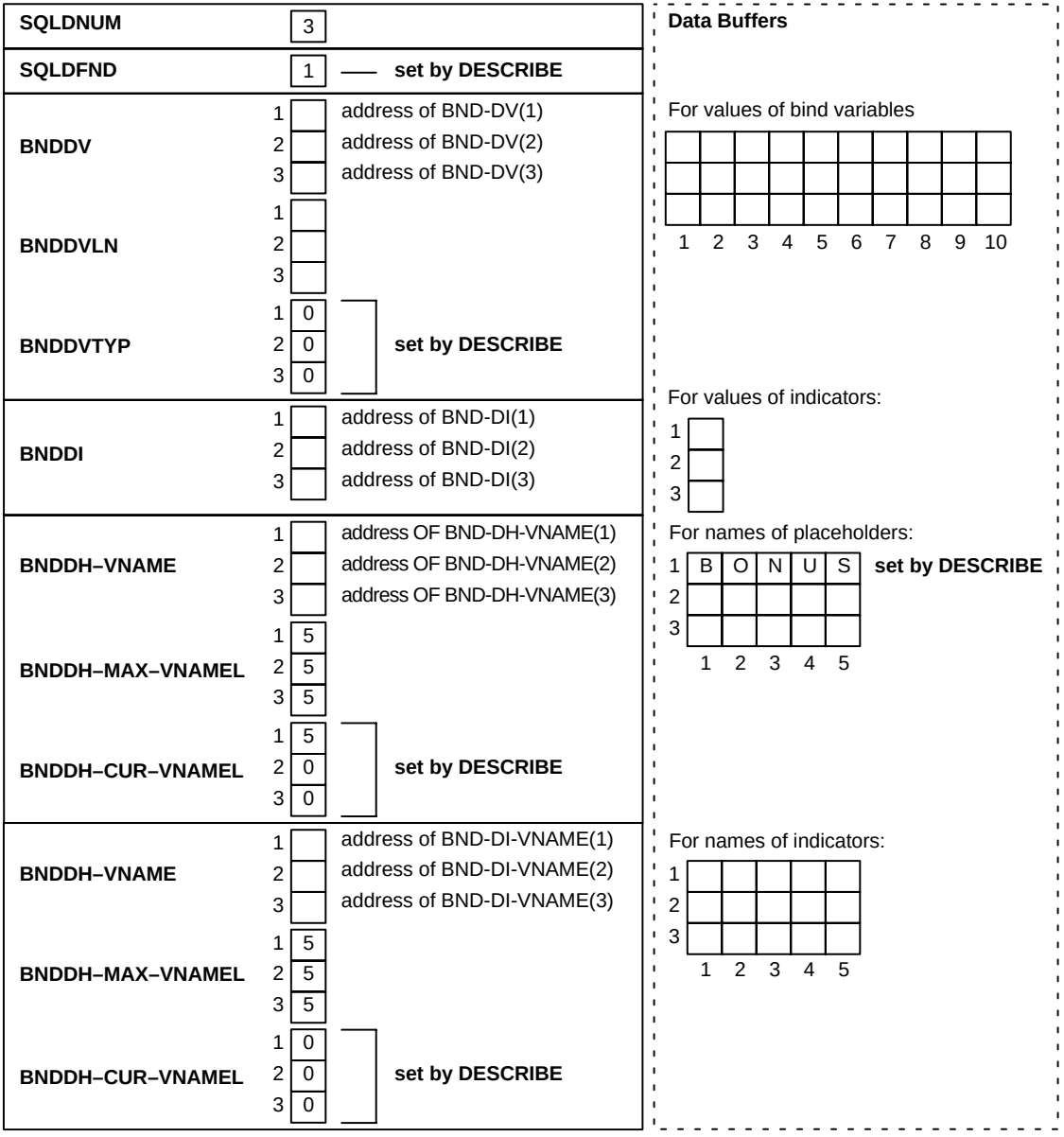
```
    END-EXEC.
```

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 11–5 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 11–5 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

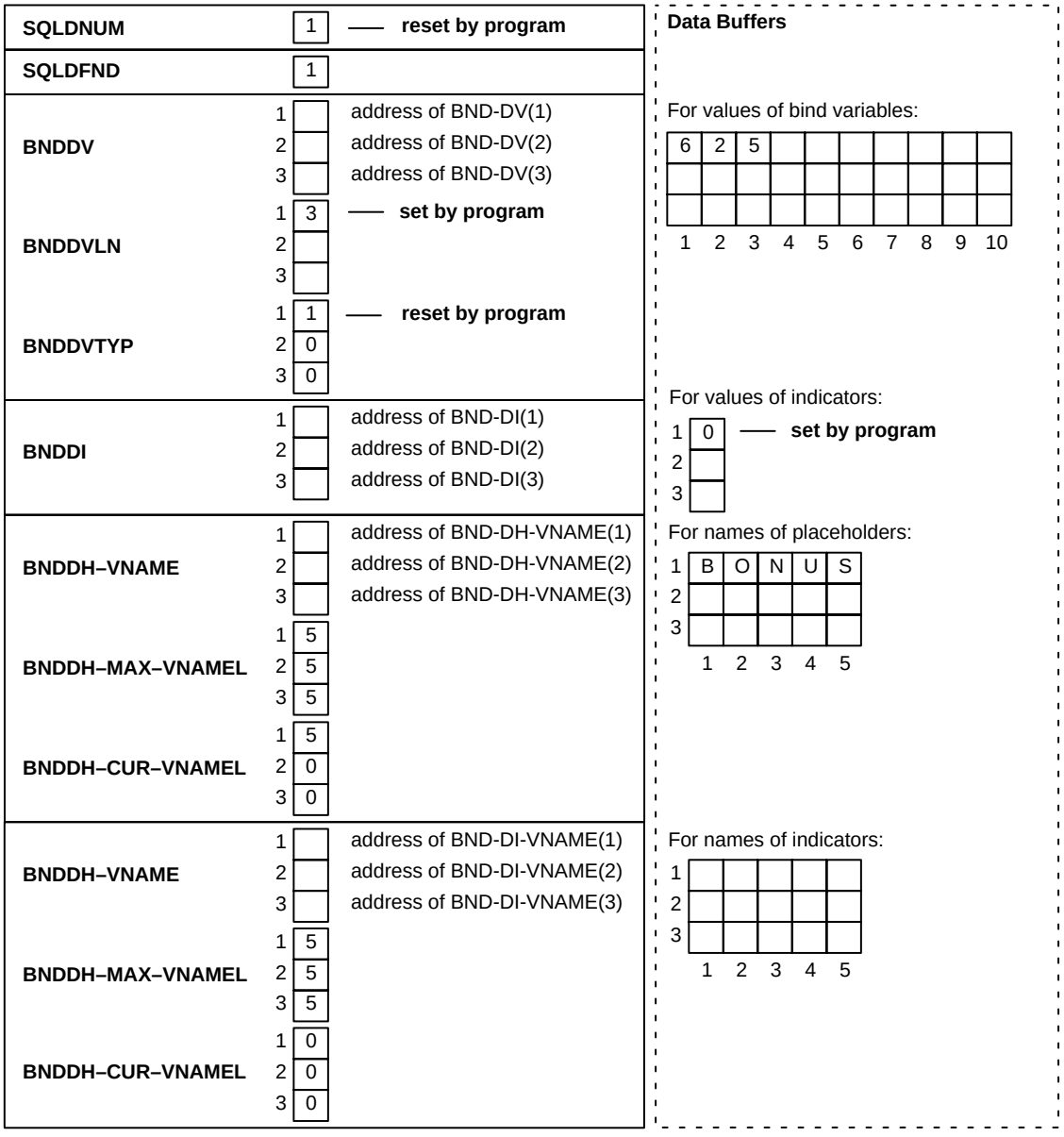
Your program must get values for the bind variables in the SQL statement. How the program gets the values is up to you. For example, they can be hard-coded, read from a file, or entered interactively.

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 11–6 Bind Descriptor after Assigning Values



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 EMPCURSOR with BNDDSC as follows:

```
EXEC SQL
    OPEN EMPCUR 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.

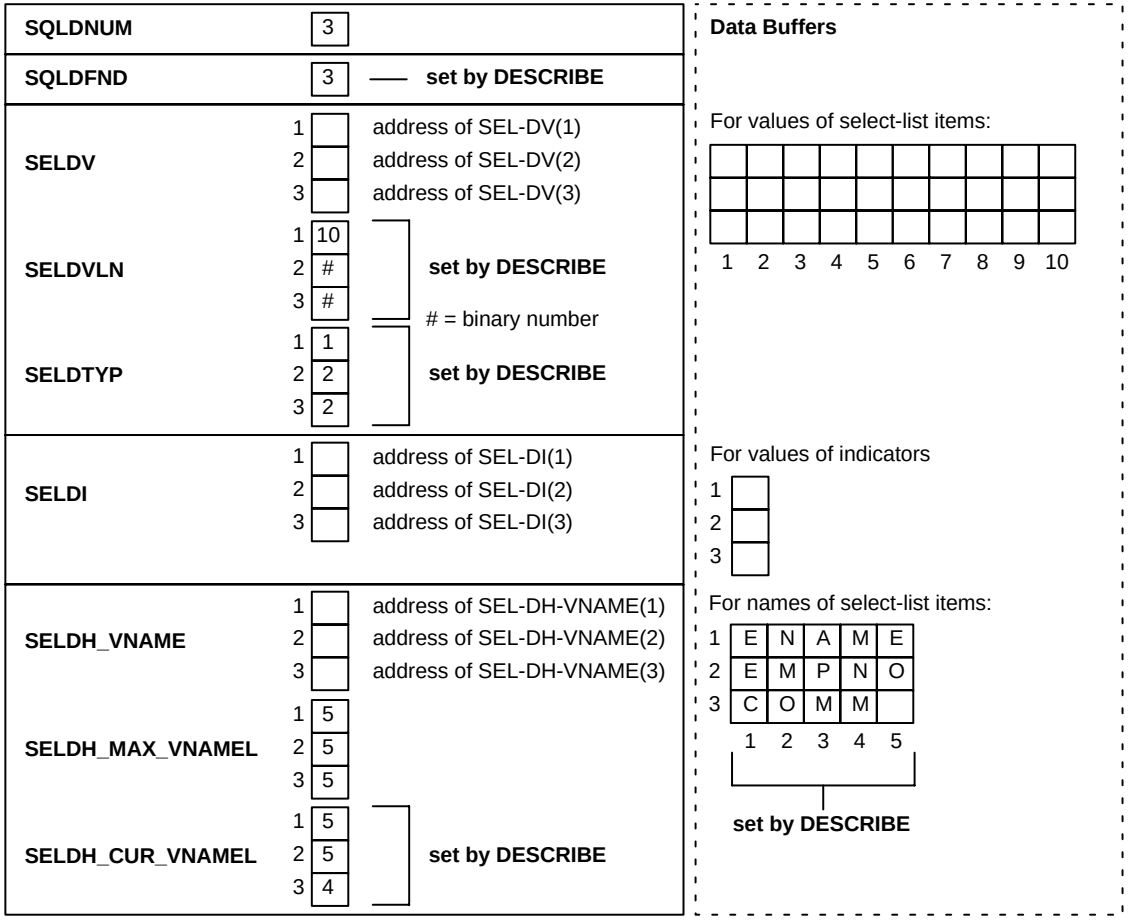
DESCRIBE SELECT LIST puts descriptions of select-list items into a select descriptor. In our example, DESCRIBE readies SELDSC as follows:

```
EXEC SQL
    DESCRIBE SELECT LIST FOR SQLSTMT INTO SELDSC
END-EXEC.
```

Accessing the data dictionary, DESCRIBE sets the length and datatype of each select-list value.

Figure 11–7 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 11-18.

Figure 11–7 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.
```

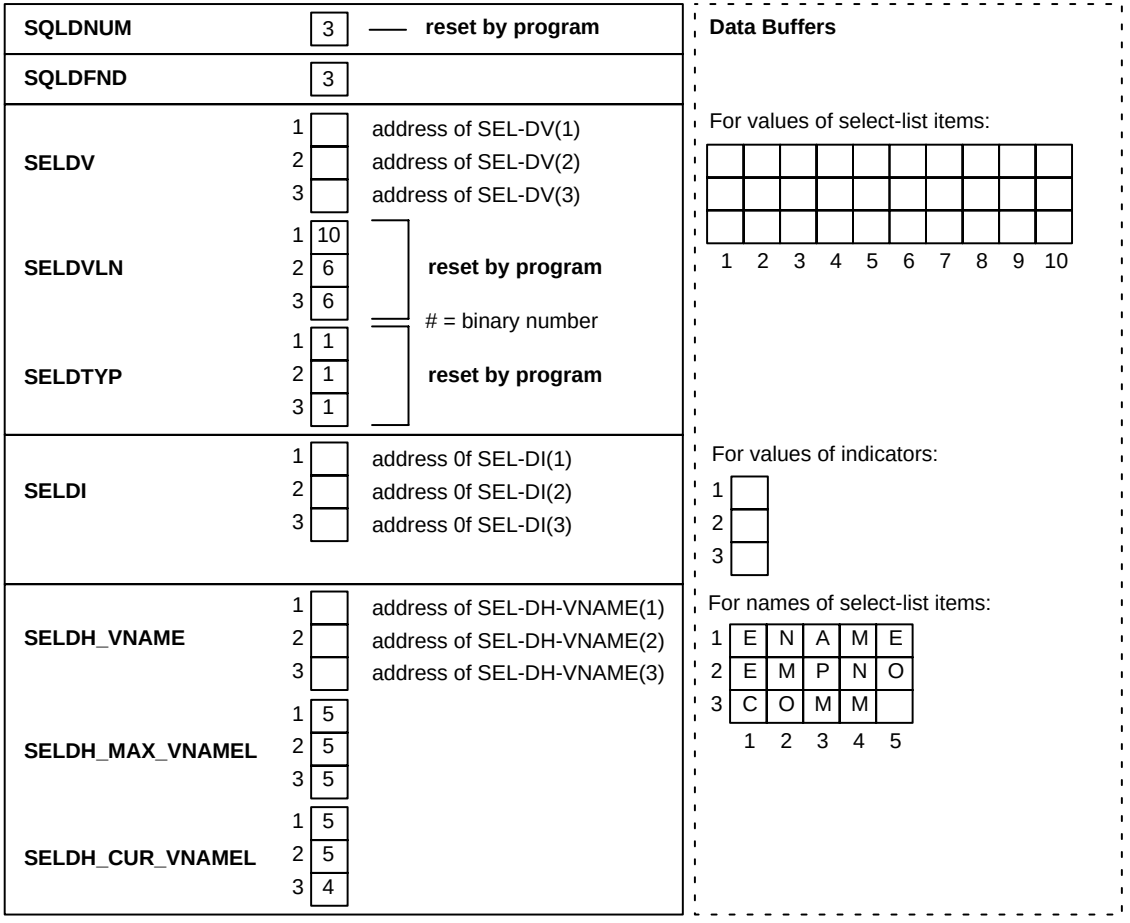
Reset Length/Datatype of Each Select-List Item

Before fetching the select-list values, the example resets 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 11–8 shows the resulting select descriptor. Notice that the NUMBER lengths are now usable and that all the datatypes are VARCHAR2. The lengths in SELDVLN(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 11–8 Select Descriptor before the FETCH



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 SQLSTATE variable to the "no data found" error code. In the following example, FETCH returns the values of columns ENAME, EMPNO, and COMM to SELDSC:

```
EXEC SQL
```

```
    FETCH EMPCURSOR USING DESCRIPTOR SELDSC
END-EXEC.
```

Figure 11–9 shows the select descriptor in our example after the FETCH. Notice that Oracle9*i* 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, Oracle9*i*, 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), Oracle9*i* 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 Oracle9*i* 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 Oracle9*i* right-justifies the value in a 9-byte field.

Get and Process Select-List Values

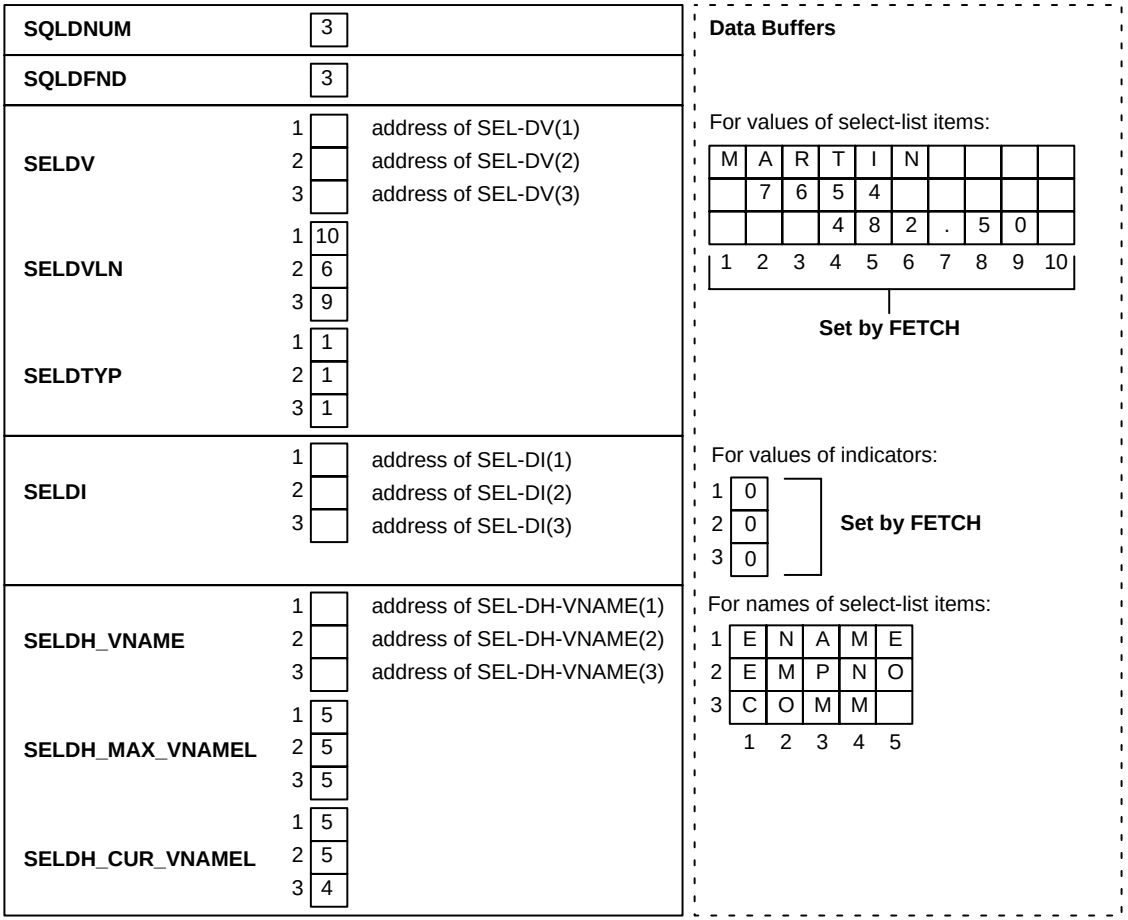
After the FETCH, your program can process the select-list values returned by FETCH. In our example, values for columns ENAME, EMPNO, and COMM are processed.

CLOSE the Cursor

CLOSE disables the cursor. In our example, CLOSE disables EMPCURSOR as follows:

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

Figure 11–9 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 Oracle9 the size of your host table. For more information about the FOR clause, see Chapter 7, "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 Oracle9*i* the number of table elements you want to process.

This procedure is necessary because Oracle9*i* 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, the program prompts the user for a SQL statement, prepares 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.

```
*****************************************************************
* Sample Program 10: Dynamic SQL Method 4                       *
*                                                               *
* This program shows the basic steps required to use dynamic    *
* SQL Method 4.  After logging on to ORACLE, the program        *
* prompts the user for a SQL statement, PREPAREs 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).
01  NULL-VAL                 PIC X(80) VALUE SPACES.
    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 "INVALID DATATYPE"
*    ORACLE 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.

*    OPEN THE CURSOR AND DESCRIBE THE SELECT-LIST ITEMS.

 DESCRIBE-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.

*    DISPLAY ORACLE ERROR MESSAGE AND CODE.

 SQL-ERROR.
     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.

 INIT-BNDDSC.

     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).

*       INIT-SELDSC: INITIALIZE THE SELECT DESCRIPTOR.

 INIT-SELDSC.

        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).

*       COERCE SELECT-LIST DATATYPES TO VARCHAR2.

 COERCE-COLUMN-TYPE.
        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.

*FETCH A ROW AND PRINT THE SELECT-LIST VALUE.

 FETCH-ROWS.
        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 A SELECT-LIST VALUE.

 PRINT-COLUMN-VALUES.
     IF SEL-DI(TABLE-INDEX) = -1
       DISPLAY NULL-VAL(1:SELDVLN(TABLE-INDEX)), " "
           WITH NO ADVANCING
     ELSE
       DISPLAY SEL-DV(TABLE-INDEX)(1:SELDVLN(TABLE-INDEX)), " "
           WITH NO ADVANCING
     END-IF.
```

12

Multithreaded Applications

If your development platform does not support threads, you may ignore this chapter.

The sections of this chapter are:

Introduction to Threads

Multithreaded applications have multiple *threads* executing in a shared address space. Threads are "lightweight" subprocesses that execute within a process. They share code and data segments, but have their own program counters, machine registers and stack. Variables declared without the thread-local attribute in working storage (as opposed to local-storage or thread-local storage) are common to all threads, and a mutual exclusivity mechanism is often required to manage access to these variables from multiple threads within an application. *Mutexes* are the synchronization mechanism to insure that data integrity is preserved.

For further discussion of mutexes, see texts on multithreading. For more detailed information about multithreaded applications, see the documentation of your threads functions.

Pro*COBOL supports development of multithreaded Oracle9*i* Server applications (on platforms that support multithreaded applications) using the following:

- A command-line option to generate thread-safe code.
- Embedded SQL statements and directives to support multithreading.
- Thread-safe SQLLIB and other client-side Oracle9*i* libraries.

Note: While your platform may support a particular thread package, see your platform-specific Oracle documentation to determine whether Oracle9*i* supports it.

The chapter's topics discuss how to use the preceding features to develop multithreaded Pro*COBOL applications:

- Runtime contexts for multithreaded applications.
- Two models for using runtime contexts.
- User-interface features for multithreaded applications.
- Programming considerations for writing multithreaded applications with Pro*COBOL.
- Sample multithreaded Pro*COBOL applications.

Runtime Contexts in Pro*COBOL

To loosely couple a thread and a connection, in Pro*COBOL we introduce the concept of a *runtime context*. The runtime context includes the following resources and their current states:

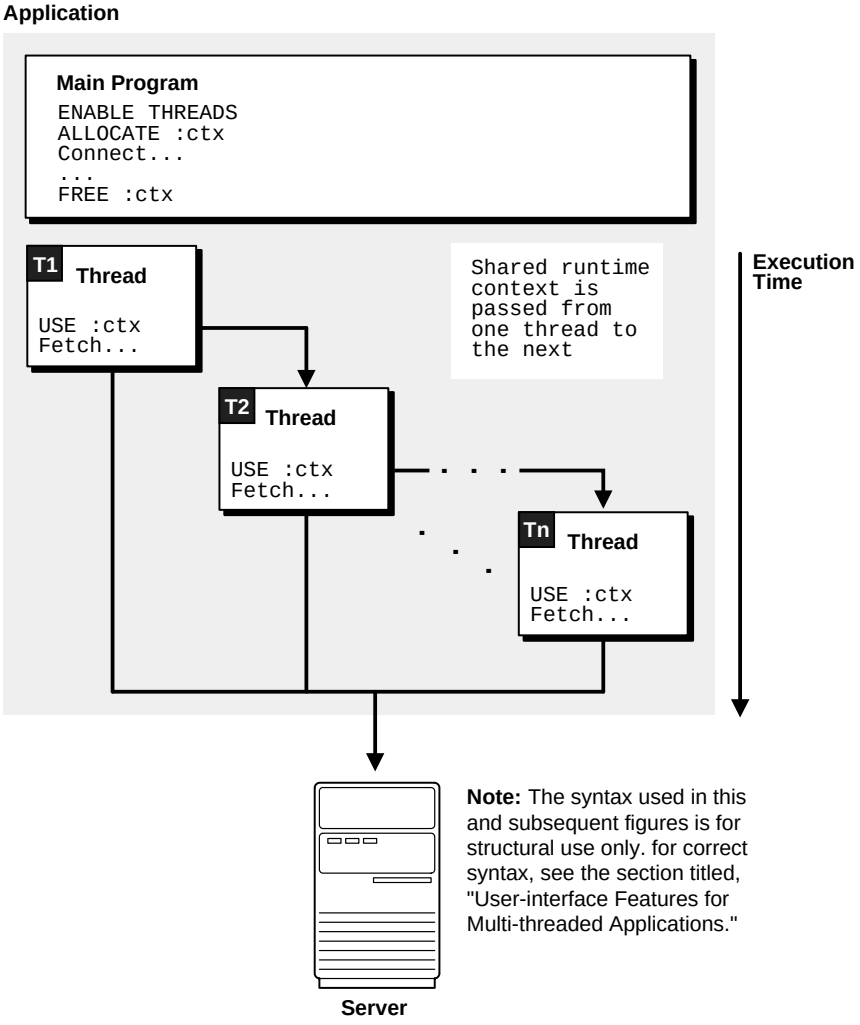
- Zero or more connections to one or more Oracle servers.

- Zero or more cursors used for the server connections.
- Inline options, such as MODE, HOLD_CURSOR, RELEASE_CURSOR, and SELECT_ERROR.

Rather than simply supporting a loose coupling between threads and connections, Pro*COBOL enables you to loosely couple threads with runtime contexts. Pro*COBOL enables your application to define a handle to a runtime context, and pass that handle from one thread to another.

For example, an interactive application spawns a thread, T1, to execute a query and return the first 10 rows to the application. T1 then terminates. After obtaining the necessary user input, another thread, T2, is spawned (or an existing thread is used) and the runtime context for T1 is passed to T2 so it can fetch the next 10 rows by processing the same cursor.This is shown in Figure 12–1:

Figure 12–1 Loosely Coupling Connections and Threads



Runtime Context Usage Models

Two possible models for using runtime contexts in multithreaded applications are shown here:

- Multiple threads sharing a single runtime context.

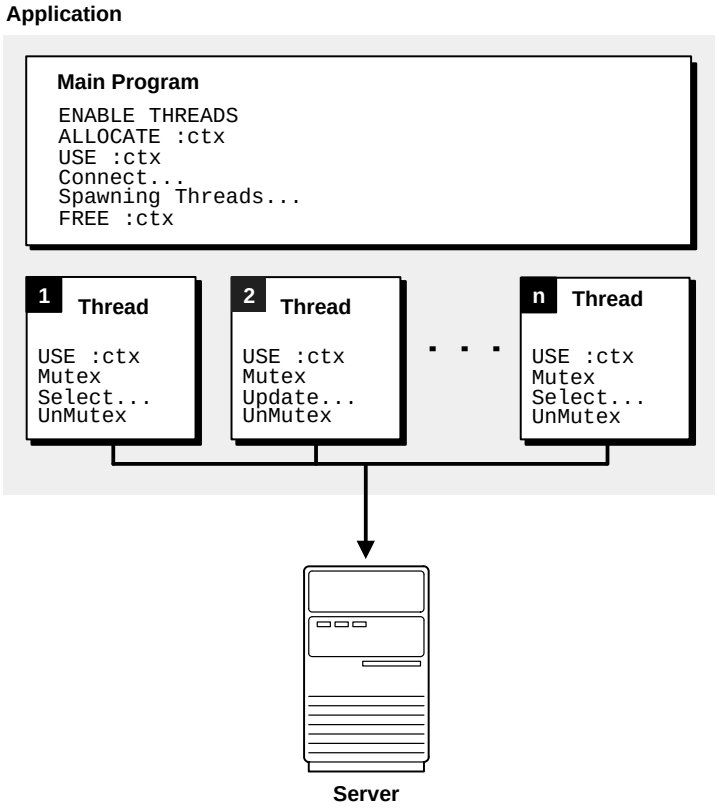
- Multiple threads using separate runtime contexts.

Regardless of the model you use for runtime contexts, you *cannot* share a runtime context between multiple threads *at the same time*. If two or more threads attempt to use the same runtime context simultaneously, a runtime error occurs

Multiple Threads Sharing a Single Runtime Context

Figure 12–2 shows an application running in a multithreaded environment. The various threads share a single runtime context to process one or more SQL statements. Again, runtime contexts cannot be shared by multiple threads at the same time. The mutexes in Figure 12–2 show how to prevent concurrent usage.

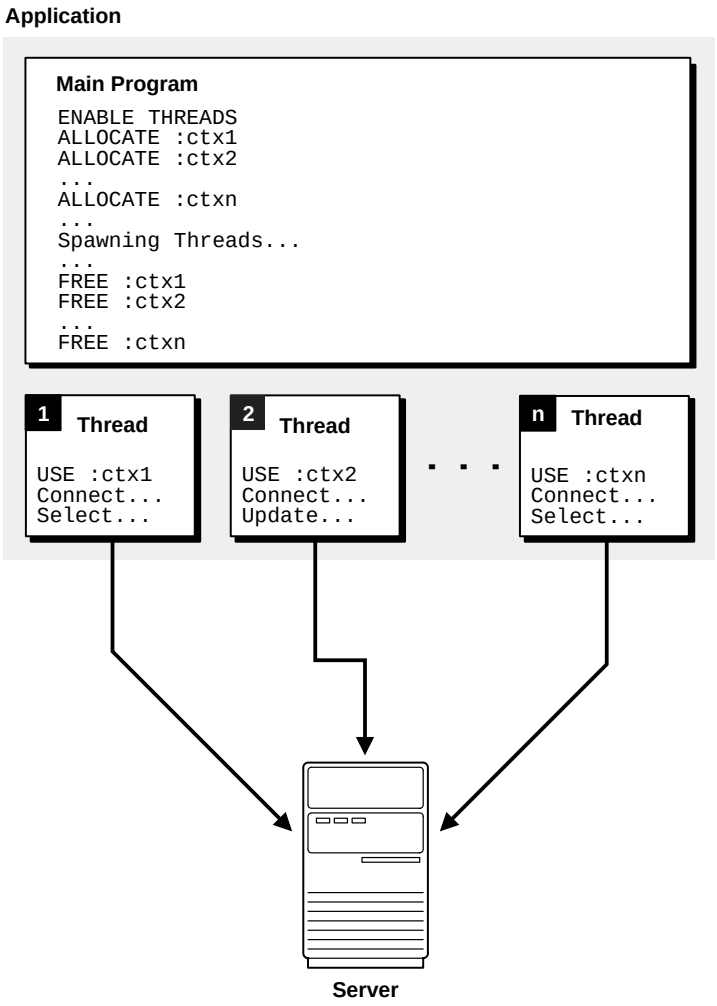
Figure 12–2 Context Sharing Among Threads



Multiple Threads Sharing Multiple Runtime Contexts

Figure 12–3 shows an application that executes multiple threads using multiple runtime contexts. In this situation, the application does not require mutexes, because each thread has a dedicated runtime context.

Figure 12–3 No Context Sharing Among Threads



User Interface Features for Multithreaded Applications

Pro*COBOL provides the following user-interface features to support multithreaded applications:

- Host variables can be declared in the LOCAL-STORAGE and the THREAD-LOCAL-STORAGE sections.

- The command-line option THREADS=YES | NO.
- Embedded SQL statements and directives.
- Thread-safe SQLLIB public functions.

THREADS Option

With THREADS=YES specified on the command line, Pro*COBOL ensures that the generated code is thread-safe, given that you follow the guidelines described in "Multithreading Programming Considerations" on page 12-10. With THREADS=YES specified, Pro*COBOL verifies that all SQL statements execute within the scope of a user-defined runtime context. If your program does not meet this requirement, a precompiler error is returned. See "THREADS" on page 14-40.

Embedded SQL Statements and Directives for Runtime Contexts

The following embedded SQL statements and directives support the definition and usage of runtime contexts and threads:

- EXEC SQL ENABLE THREADS END-EXEC.
- EXEC SQL CONTEXT ALLOCATE :*context_var* END-EXEC.
- EXEC SQL CONTEXT USE { :*context_var* | DEFAULT} END-EXEC.
- EXEC SQL CONTEXT FREE :*context_var* END-EXEC.

For these EXEC SQL statements, *context_var* is the handle to the runtime context and must be declared of type SQL-CONTEXT as follows:

```
01  SQL-CONTEXT context_var END-EXEC.
```

Using DEFAULT means that the default (global) runtime context will be used in all embedded SQL statements that lexically follow until another CONTEXT USE statement overrides it.

Examples illustrating the various uses of context statements are shown on page 12-10.

Host Tables of SQL-CONTEXT Are Not Allowed

You cannot declare host tables of SQL-CONTEXT. Instead, declare a host table of S9(9) COMP variables and then pass them to the subprogram one at a time after redeclaring them in the subprogram as SQL-CONTEXT.

EXEC SQL ENABLE THREADS

This executable SQL statement initializes a process that supports multiple threads. This must be the first executable SQL statement in a program that contains a multithreaded application. There can only be one ENABLE THREADS statement in all files of an application, or an error results. For more detailed information, see "ENABLE THREADS (Executable Embedded SQL Extension)" on page F-38.

EXEC SQL CONTEXT ALLOCATE

This executable SQL statement allocates and initializes memory for the specified runtime context; the runtime-context variable must be declared of type SQL_CONTEXT. For more detailed information, see "CONTEXT ALLOCATE (Executable Embedded SQL Extension)" on page F-19.

EXEC SQL CONTEXT USE

The EXEC SQL CONTEXT USE directive instructs the precompiler to use the specified runtime context for subsequent executable SQL statements. The runtime context specified must be previously allocated using an EXEC SQL CONTEXT ALLOCATE statement.

The EXEC SQL CONTEXT USE directive works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given source file without regard to standard COBOL scope rules.

For more detailed information, see "CONTEXT USE (Oracle Embedded SQL Directive)" on page F-21, and "CONTEXT ALLOCATE (Executable Embedded SQL Extension)" on page F-19.

EXEC SQL CONTEXT FREE

The EXEC SQL CONTEXT FREE executable SQL statement frees the memory associated with the specified runtime context and places a null pointer in the host program variable. For more detailed information, see "CONTEXT FREE (Executable Embedded SQL Extension)" on page F-20.

Communication with Pro*C/C++ Programs

Runtime contexts can be passed using arguments defined in the Linkage Section. Multithreaded Pro*C/C++ programs can call Pro*COBOL subprograms and Pro*COBOL programs can call subprograms written in Pro*C/C++.

Multithreading Programming Considerations

While Oracle9*i* ensures that the SQLLIB code is thread-safe, you are responsible for ensuring that your source code is designed to work properly with threads. For example, carefully consider the scope of the variables you use.

In addition, multithreading requires design decisions regarding the following:

- Including one SQLCA for each runtime context.
- Declaring the SQLDA as a thread-safe group item, like the SQLCA, typically an auto variable, one for each runtime context.
- Declaring host variables in a thread-safe fashion, in other words, carefully consider your use of static and global host variables.
- Avoiding simultaneous use of a runtime context in multiple threads.
- Whether or not to use default database connections or to explicitly define them using the AT clause.

No more than one executable embedded SQL statement, for example, EXEC SQL UPDATE, may be outstanding on a runtime context at a given time.

Existing requirements for precompiled applications also apply. For example, all references to a given cursor must appear in the same source file.

Restrictions on Multithreading

The following restrictions be in effect when using threads:

- You cannot use an array of datatype SQL-CONTEXT.
- Concurrent threads should each have its own SQLCA.
- Concurrent threads should each have its own context areas.

Multiple Context Examples

The code fragments in this section show how to use multiple contexts, and demonstrate the scope of the context use statement.

Example 1

In the first example, the precompiler option setting THREADS=YES is not needed, because we are not generating threads:

```
IDENTIFICATION DIVISION.
```

```
 PROGRAM-ID. MAIN.
 ...
* declare a context area
 01  CTX1  SQL-CONTEXT.
 01  UID1  PIC X(11) VALUE "SCOTT/TIGER".
 01  UID2  PIC X(10) VALUE "MARY/LION"

 PROCEDURE DIVISION.
...
* allocate context area
     EXEC SQL CONTEXT ALLOCATE :CTX1 END-EXEC.
     EXEC SQL CONTEXT USE :CTX1 END-EXEC.
* all statements until the next context use will use CTX1
     EXEC SQL CONNECT :UID1 END-EXEC.
     EXEC SQL SELECT ....
     EXEC SQL CONTEXT USE DEFAULT END-EXEC.
* all statements physically following the above will use the default context
     EXEC SQL CONNECT :UID2 END-EXEC.
     EXEC SQL INSERT ...
 ...
```

Example 2

This next example shows multiple contexts. One context is used by the generated thread while the other is used by the main program. The started thread, SUBPRGM1, will use context CTX1, which is passed to it through the LINKAGE SECTION. This example also demonstrates the scope of the CONTEXT USE statement.

> **Note:** You must precompile the main program file, and the main program of every subsequent example in this section, with the option THREADS=YES.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MAIN.
 ...
* declare two context areas
 01  CTX1  SQL-CONTEXT.
 01  CTX2  SQL-CONTEXT.

 PROCEDURE DIVISION.

* enable threading
     EXEC SQL ENABLE THREADS END-EXEC.

* allocate context areas
```

```
     EXEC SQL CONTEXT ALLOCATE :CTX1 END-EXEC.
     EXEC SQL CONTEXT ALLOCATE :CTX2 END-EXEC.

* include your code to start thread "SUBPGM1" using CTX1 here.

     EXEC SQL CONTEXT USE :CTX2 END-EXEC.
* all statement physically following the above will use CTX2

     EXEC SQL CONNECT :USERID END-EXEC.
     EXEC SQL INSERT .....
 ...
```

The thread SUBPRGM1 is in a another file:

```
 PROGRAM-ID. SUBPRGM1.
 ...
 01  USERID PIC X(11) VALUE "SCOTT/TIGER".
 LINKAGE SECTION.
 01  CTX1 SQL-CONTEXT.
 PROCEDURE DIVISION USING CTX1.

     EXEC SQL CONTEXT USE :CTX1 END-EXEC.
     EXEC SQL CONNECT :USERID END-EXEC.
     EXEC SQL SELECT ...
 ...
```

Example 3

The following example uses multiple threads. Each thread has its own context. If the threads are to be executed concurrently, each thread *must* have its own context. Contexts are passed to the thread with the USING CLAUSE of the START statement and are declared in the LINKAGE SECTION of the threaded subprogram.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MAIN.
 ...
 DATA DIVISION.

 01  CTX1 SQL-CONTEXT.
 01  CTX2 SQL-CONTEXT.

 PROCEDURE DIVISION.
 ...
     EXEC SQL ENABLE THREADS END-EXEC.
     EXEC SQL CONTEXT ALLOCATE :CTX1 END-EXEC.
     EXEC SQL CONTEXT ALLOCATE :CTX2 END-EXEC.
```

```
* include your code to start thread "SUBPGM" using CTX1 here.
* include your code to start thread "SUBPGM" using CTX2 here.
 ...
```

The thread SUBPGM is placed in another file:

```
PROGRAM-ID. SUBPGM.
 ...
DATA DIVISION.
 ...
01  USERID PIC X(11) VALUE "SCOTT/TIGER".
 ...
LINKAGE SECTION.
01  CTX SQL-CONTEXT.
PROCEDURE DIVISION USING CTX.
    EXEC SQL CONTEXT USE :CTX END-EXEC.
    EXEC SQL CONNECT :USERID END-EXEC.
    EXEC SQL SELECT ....
 ...
```

Example 4

The next example is based on the previous example, but does the connection in the top level program and passes the connection with the context to the threaded subprogram.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
 ...
DATA DIVISION.

01  CTX1 SQL-CONTEXT.
01  CTX2 SQL-CONTEXT.
01  USERID PIC X(11) VALUE "SCOTT/TIGER".

ROCEDURE DIVISION.

    EXEC SQL ENABLE THREADS END-EXEC.
    EXEC SQL CONTEXT ALLOCATE :CTX1 END-EXEC.
    EXEC SQL CONTEXT ALLOCATE :CTX2 END-EXEC.
    EXEC SQL CONTEXT USE :CTX1 END-EXEC.
    EXEC SQL CONNECT :USERID END-EXEC.
    EXEC SQL CONTEXT USE :CTX2 END-EXEC.
    EXEC SQL CONNECT :USERID END-EXEC.
```

```
* include your code to start thread "SUBPGM" using CTX1 here.
* include your code to start thread "SUBPGM" using CTX2 here.
 ...
```

The thread SUBPRGM is in another file:

```
 PROGRAM-ID. SUBPGM.
 ...
 LINKAGE SECTION.
 01  CTX SQL-CONTEXT.
 PROCEDURE DIVISION USING CTX.
     EXEC SQL CONTEXT USE :CTX END-EXEC.
     EXEC SQL SELECT ....
 ...
```

Example 5

The following example shows multiple threads which share a context. Note that in this case, the threads *must* be serialized.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MAIN.
 ...
 DATA DIVISION.

 01  CTX1 SQL-CONTEXT.

 PROCEDURE DIVISION.

     EXEC SQL ENABLE THREADS END-EXEC.
     EXEC SQL CONTEXT ALLOCATE :CTX1 END-EXEC.

* include your code to start thread1 "SUBPGM1" using CTX1 here.
* include your code to wait for thread1 here.
* include your code to start thread2 "SUBPGM2" using CTX1 here.
 ...
```

There are two separate files for the two threads. First there is:

```
 PROGRAM-ID. SUBPGM1.
 ...
 DATA DIVISION.
 ..
 01  USERID PIC X(11) VALUE "SCOTT/TIGER".
 ...
 LINKAGE SECTION.
```

```
01  CTX SQL-CONTEXT.
PROCEDURE DIVISION USING CTX.
    EXEC SQL CONTEXT USE :CTX END-EXEC.
...
    EXEC SQL CONNECT :USERID END-EXEC.
```

Another file contains SUBPGM2:

```
PROGRAM-ID. SUBPGM2.
...
DATA DIVISION.
...
LINKAGE SECTION.
01  CTX SQL-CONTEXT.
PROCEDURE DIVISION USING CTX.
    EXEC SQL CONTEXT USE :CTX END-EXEC.
    EXEC SELECT ....
...
```

Multithreaded Example

This multi-file application demonstrates one way to use the SQLLIB runtime context area (SQL-CONTEXT) to support multiple threads. Precompile with THREADS=YES.

The main program, orathrd2, declares an array of S9(9) COMP variables to be used to hold the sqllib contexts. It enables threading through the

```
EXEC SQL ENABLE THREADS END-EXEC.
```

statement and then calls the subprogram oracon (in file `oracon.pco`) to allocate the threads. oracon also establishes a connection for each allocated context.

Next, ORTHRD2 passes the context to one of the threaded entry points, THREAD-1 or THREAD-2. THREAD-1 simply selects and displays the salary for an employee. THREAD-2 selects and updates the salary for that employee. Since THREAD-2 issues a commit, the update is visible to threads that do the SELECT after it has committed. (But not those which run concurrently with the update.) Note that the output will vary from run to run because the timing of the update and commit is non-determinant.

It is important to remember that concurrent threads must each have their own contexts. Contexts may be passed to and used by subsequent threads, but threads may not use the same context concurrently. This model could be used for

connection pooling, where the maximum number of connections are created initially and passed to threads as available, to execute user's requests.

An array of S9(9) COMP variables is used because you cannot currently declare an array of SQL-CONTEXT.

Note: This program was developed specifically for a Sun workstation running Solaris and Merant MicroFocus ServerExpress compiler and uses vendor-specific directives and functionality.

See your platform-specific documentation for the specific COBOL statements that support multithreading.

The main program is in file `orathrd2.pco`:

```
$SET REENTRANT MF
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ORATHRD2.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 78 MAX-LOOPS            VALUE 10.
 01 THREAD-ID            USAGE POINTER.
 01 TP-1                 USAGE THREAD-POINTER OCCURS MAX-LOOPS.
 01 IDEN-4               PIC 9(4).
 01 LOOP-COUNTER         PIC 9(2)  COMP-X EXTERNAL.
 01 PEMPNO               PIC S9(4) COMP EXTERNAL.
 01 ISAL                 PIC S9(4) COMP   VALUE ZERO.
     EXEC SQL
          INCLUDE SQLCA
     END-EXEC.
 THREAD-LOCAL-STORAGE SECTION.
 01  CONTEXT-AREA          PIC S9(9) COMP OCCURS MAX-LOOPS.
 PROCEDURE DIVISION.
 MAIN SECTION.

     PERFORM INITIALISATION
     PERFORM ORACLE-CONNECTIONS VARYING LOOP-COUNTER
             FROM 1 BY 1 UNTIL LOOP-COUNTER > MAX-LOOPS
     PERFORM  VARYING LOOP-COUNTER FROM 1 BY 1
        UNTIL LOOP-COUNTER > MAX-LOOPS
         PERFORM START-THREAD
     END-PERFORM
     STOP RUN.

*-----------------------------------------------------------------
```

```
* CHECK THAT WE ARE RUNNING UNDER A MULTI THREADED RTS.
*-----------------------------------------------------------------
 INITIALISATION SECTION.

     CALL "CBL_THREAD_SELF" USING THREAD-ID ON EXCEPTION
          DISPLAY "NO THREAD SUPPORT IN THIS RTS"
          STOP RUN
     END-CALL
     IF RETURN-CODE = 1008
          DISPLAY "CANNOT RUN THIS TEST ON SINGLE THREADED RTS"
          STOP RUN
     END-IF
     DISPLAY "MULTI-THREAD RTS"

* ENABLING THREADS MUST BE DONE ONCE BEFORE ANY CONTEXT USEAGE
     EXEC SQL ENABLE THREADS END-EXEC.
     IF SQLCODE NOT = ZERO
        DISPLAY 'ERROR ENABLING ORACLE THREAD SUPPORT '
                ' - ABORTING : ' SQLERRMC
        STOP RUN
     END-IF

*  SET A VALUE FOR THE EMPLOYEE NUMBER.  BECAUSE THIS IS AN
*  EXTERNAL VARIABLE, A COPY OF ITS VALUE IS VISIBLE TO THE
*  OTHER MODULES IN THIS APPLICATION
     MOVE 7566 TO PEMPNO
     EXIT SECTION.

*-----------------------------------------------------------------
* CREATE THREADS AND START WITH EITHER THREAD-1 OR THREAD-2
*-----------------------------------------------------------------
 START-THREAD SECTION.

     IF LOOP-COUNTER = 2 OR LOOP-COUNTER = 5
        START "THREAD-2 "
           USING CONTEXT-AREA(LOOP-COUNTER)
           IDENTIFIED BY TP-1(LOOP-COUNTER)
           STATUS IS IDEN-4
           ON EXCEPTION DISPLAY "THREAD CREATE FAILED"
        END-START
        IF IDEN-4 NOT = ZERO
          DISPLAY "THREAD CREATE FAILED RETURNED " IDEN-4
        END-IF
     ELSE
        START "THREAD-1 "
```

```
               USING CONTEXT-AREA(LOOP-COUNTER)
               IDENTIFIED BY TP-1(LOOP-COUNTER)
               STATUS IS IDEN-4
               ON EXCEPTION DISPLAY "THREAD CREATE FAILED"
            END-START
            IF IDEN-4 NOT = ZERO
              DISPLAY "THREAD CREATE FAILED RETURNED " IDEN-4
            END-IF
        END-IF.

    START-THREAD-END.
        EXIT SECTION.


   *----------------------------------------------------------------
   * ALLOCATE CONTEXT AREAS ESTABLISH CONNECTION WITH EACH AREA.
   *----------------------------------------------------------------
    ORACLE-CONNECTIONS SECTION.

        CALL "ORACON" USING CONTEXT-AREA(LOOP-COUNTER).
    ORACLE-CONNECTIONS-END.
        EXIT SECTION.
```

Here is the file `thread-1.pco`:

```
* This is Thread 1.  It selects and displays the data for
* the employee. The context area upon which a connection
* has been established is passed to the thread via the
* linkage section. In a multi-file application, you
* can pass the context via the linkage section.
* Precompile with THREADS=YES.
*
$SET REENTRANT MF
 IDENTIFICATION DIVISION.
 PROGRAM-ID. THREAD-1.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 PEMPNO              PIC S9(4) COMP EXTERNAL.

 LOCAL-STORAGE SECTION.
 01 DEMPNO              PIC Z(4) VALUE ZERO.
 01 PEMP-NAME1          PIC X(15) VARYING  VALUE SPACES.
 01 PSAL-VALUE1         PIC S9(7)V99 COMP-3 VALUE ZERO.
 01 ISAL1               PIC S9(4)   COMP   VALUE ZERO.
```

```
 01 DSAL-VALUE           PIC +(7).99 VALUE ZERO.
     EXEC SQL
          INCLUDE SQLCA
     END-EXEC.

 LINKAGE SECTION.
 01 CONTEXT-AREA1         SQL-CONTEXT.

*----------------------------------------------------------------
* USING THE PASSED IN CONTEXT, SELECT AND DISPLAY THE
* DATA FOR EMPLOYEE.
*----------------------------------------------------------------
 PROCEDURE DIVISION USING CONTEXT-AREA1.
 MAIN SECTION.

     EXEC SQL WHENEVER SQLERROR GOTO SELECT-ERROR END-EXEC
     EXEC SQL CONTEXT USE :CONTEXT-AREA1 END-EXEC
     EXEC SQL
          SELECT  ENAME, SAL
            INTO  :PEMP-NAME1, :PSAL-VALUE1:ISAL1
            FROM  EMP
           WHERE  EMPNO = :PEMPNO
     END-EXEC
     IF ISAL1 < ZERO
        MOVE ZERO     TO PSAL-VALUE1
     END-IF
     MOVE PEMPNO      TO DEMPNO
     MOVE PSAL-VALUE1 TO DSAL-VALUE
     DISPLAY "FOR EMP ", DEMPNO, " NAME ",
             PEMP-NAME1-ARR(1:PEMP-NAME1-LEN),
             " THE CURRENT SALARY IS ", DSAL-VALUE
     EXIT PROGRAM.


*----------------------------------------------------------------
* THERE HAS BEEN AN ERROR WHEN SELECTING FROM THE EMP TABLE
*----------------------------------------------------------------
 SELECT-ERROR SECTION.

     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC
     DISPLAY "HIT AN ORACLE ERROR SELECTING EMPNO 7566"
     DISPLAY "SQLCODE = ", SQLCODE
     DISPLAY "ERROR TEXT ", SQLERRMC(1:SQLERRML)
     GOBACK
     EXIT SECTION.
```

Here is the file `thread-2.pco`:

```
* This is Thread 2.  The program will select, then update,
* increment, and then commit the salary.  It uses the passed-in
* context upon which a connection has previously been established.
* Precompile with THREADS=YES.
*
$SET REENTRANT MF
 IDENTIFICATION DIVISION.
 PROGRAM-ID. THREAD-2.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 PEMPNO              PIC S9(4)   COMP EXTERNAL.

 LOCAL-STORAGE SECTION.
 01 DEMPNO              PIC Z(4) VALUE ZERO.
 01 PEMP-NAME2          PIC X(15) VARYING  VALUE SPACES.
 01 PSAL-VALUE2         PIC S9(7)V99 COMP-3 VALUE 100.
 01 ISAL2               PIC S9(4)   COMP   VALUE ZERO.
 01 DSAL-VALUE          PIC +(7).99 VALUE ZERO.
     EXEC SQL
          INCLUDE SQLCA
     END-EXEC.

 LINKAGE SECTION.
 01 CONTEXT-AREA2         SQL-CONTEXT.

*-------------------------------------------------------------
* USING THE PASSED IN CONTEXT AREA, FIRST SELECT TO GET INITIAL
* VALUES, INCREMENT THE SALARY, UPDATE AND COMMIT.
*-------------------------------------------------------------
 PROCEDURE DIVISION USING CONTEXT-AREA2.
 MAIN SECTION.

     EXEC SQL WHENEVER SQLERROR GOTO UPDATE-ERROR END-EXEC
     EXEC SQL CONTEXT USE     :CONTEXT-AREA2 END-EXEC
     EXEC SQL
          SELECT  ENAME, SAL
            INTO  :PEMP-NAME2, :PSAL-VALUE2:ISAL2
            FROM  EMP
           WHERE  EMPNO = :PEMPNO
     END-EXEC
     ADD  10  TO PSAL-VALUE2
```

```
           EXEC SQL
                   UPDATE  EMP
                      SET  SAL   = :PSAL-VALUE2
                     WHERE  EMPNO = :PEMPNO
           END-EXEC
           MOVE PEMPNO      TO DEMPNO
           MOVE PSAL-VALUE2 TO DSAL-VALUE
           DISPLAY "FOR EMP ", DEMPNO, " NAME ",
                   PEMP-NAME2-ARR(1:PEMP-NAME2-LEN),
                   " THE UPDATED SALARY IS ", DSAL-VALUE
      *    THIS COMMIT IS REQUIRED, OTHERWISE THE DATABASE
      *    WILL BLOCK SINCE THE UPDATES ARE TO THE SAME ROW
           EXEC SQL COMMIT WORK END-EXEC
           EXIT PROGRAM.

      *---------------------------------------------------------------
      * THERE HAS BEEN AN ERROR WHEN UPDATING THE SAL IN THE EMP TABLE
      *---------------------------------------------------------------
       UPDATE-ERROR SECTION.

           EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC
           DISPLAY "HIT AN ORACLE ERROR UPDATING EMPNO 7566"
           DISPLAY "SQLCODE = ", SQLCODE
           DISPLAY "ERROR TEXT ", SQLERRMC(1:SQLERRML)
           GOBACK
           EXIT SECTION.
```

The file `oracon.pco` follows:

```
* This program allocates SQLLIB runtime contexts, stores
* a pointer to the context in the variable which was
* passed in from the main program via the linkage section,
* and establishes a connection on the allocated context.
*
* This program is written for Merant MicroFocus COBOL and uses
* vendor-specific directives and functionality. Precompile
* with THREADS=YES.
*
$SET REENTRANT MF
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ORACON.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
```

```
 01 LOGON-STRING         PIC X(40)          VALUE SPACES.
     EXEC SQL
          INCLUDE SQLCA
     END-EXEC.
 LINKAGE SECTION.
 01  CONTEXT          SQL-CONTEXT.

 PROCEDURE DIVISION USING CONTEXT.
 MAIN SECTION.

*----------------------------------------------------------------
* ALLOCATE CONTEXT AREAS ESTABLISH CONNECTION WITH EACH AREA.
*----------------------------------------------------------------
 ORACLE-CONNECTION SECTION.

     MOVE "SCOTT/TIGER"  TO LOGON-STRING
     EXEC SQL CONTEXT ALLOCATE :CONTEXT END-EXEC
     IF SQLCODE NOT = ZERO
        DISPLAY 'ERROR ALLOCATING CONTEXT '
                '- ABORTING : ' SQLERRMC
        GOBACK
     ELSE
        DISPLAY 'CONTEXT ALLOCATED'
     END-IF

     EXEC SQL CONTEXT USE :CONTEXT END-EXEC
     EXEC SQL  CONNECT    :LOGON-STRING  END-EXEC
     IF SQLCODE NOT = ZERO
        DISPLAY 'ERROR CONNECTING SECOND THREAD TO THE DATABASE '
                '- ABORT TEST : ' SQLERRMC
        GOBACK
     ELSE
        DISPLAY 'CONNECTION ESTABLISHED'
     END-IF
     EXIT SECTION.
```

13

Large Objects (LOBs)

This chapter describes the support provided by embedded SQL statements for the *LOB* (Large Object) datatypes. The four types of LOBs are introduced and compared to the older LONG and LONG RAW datatypes.

The embedded SQL interface in Pro*COBOL is shown to provide similar functionality to that of the PL/SQL language.

The LOB statements and their options and host variables are presented.

Last, an example of Pro*COBOL programming using the LOB interface.

The main sections are:

- Using LOBs
- How to Use LOBs
- Rules for LOB Statements
- LOB Statements
- LOB Sample Program: LOBDEMO1.PCO

For additional details on Large Objects, see:

Using LOBs

LOBs (large objects) are database types that are used to store large amounts of data (maximum size is 4 Gigabytes) such as ASCII text, text in National Characters, files in various graphics formats, and sound wave forms.

Internal LOBs

Internal LOBs (BLOBs, CLOBs, NCLOBs) are stored in database table spaces and have transactional support of the database server. (COMMITT, ROLLBACK, and so forth work with them.)

BLOBs (Binary LOBs) store unstructured binary (also called "raw") data, such as video clips.

CLOBs (Character LOBs) store large blocks of character data from the database character set.

NCLOBs (National Character LOBs) store large blocks of character data from the national character set.

External LOBs

External LOBs are operating system files outside the database tablespaces, that have no transactional support from the database server.

BFILEs (Binary Files) store data in external binary files. A BFILE can be in GIF, JPEG, MPEG, MPEG2, text, or other formats.

Security for BFILEs

The *DIRECTORY* object is used to access and use BFILEs. The DIRECTORY is a logical alias name (stored in the server) for the actual physical directory in the server file system containing the file. Users are permitted to access the file only if granted access privilege on the DIRECTORY object.

Two kinds of SQL statements can be used with BFILEs:

- The DDL (Data Definition Language) SQL statements CREATE, REPLACE, ALTER, and DROP.
- The DML (Data Management Language) SQL statements are used to GRANT and REVOKE the READ system and object privileges on DIRECTORY objects.

A sample CREATE DIRECTORY directive is:

```
EXEC SQL CREATE OR REPLACE DIRECTORY "Mydir" AS '/usr/home/mydir' END-EXEC.
```

Other users or roles can read the directory only if you grant them permission with a DML (Data Manipulation Language) statement, such as GRANT. For example, to allow user `scott` to read BFILES in directory `/usr/home/mydir`:

```
EXEC SQL GRANT READ ON DIRECTORY "Mydir" TO scott END-EXEC.
```

Up to 10 BFILES can be opened simultaneously in one session. This default value can be changed by setting the SESSION_MAX_OPEN_FILES parameter.

See *Oracle9i Application Developer's Guide - Fundamentals* for more details on DIRECTORY objects and BFILE security, and the the GRANT command.

LOBs Compared with LONG and LONG RAW

LOBs are different from the older LONG and LONG RAW datatypes in many ways.

- The maximum size of a LOB is 4 Gigabytes versus 2 Gigabytes for LONG and LONG RAW.
- You can use random as well as sequential access methods on LOBs; you can only use sequential access methods on LONG and LONG RAW.
- LOBs (except NCLOBs) can be attributes of an object type that you define.
- Tables can have multiple LOB columns, but can have only one LONG or LONG RAW column.

Migration of existing LONG and LONG Raw attributes to LOBs is recommended by Oracle. Oracle plans to end support of LONG and LONG RAW in future releases. See Also: *Oracle9i Database Migration* for more information on migration, and *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for more information on LOBs.

LOB Locators

A *LOB locator* points to the actual LOB contents. The locator is returned when you retrieve the LOB, not the LOB's contents. LOB locators cannot be saved in one transaction or session and used again in a later transaction or session.

Temporary LOBs

You can create *temporary* LOBs to assist your use of database LOBs. Temporary LOBs are like local variables and are not associated with any table. They are only accessible by their creator using their locators and are deleted when a session ends.

There is no support for temporary BFILES. Temporary LOBs are only permitted to be input variables (IN values) in the WHERE clause of an INSERT statement, in the SET clause of an UPDATE, or in the WHERE clause of a DELETE statement. Temporary LOBs have no transactional support from the database server, which means that you cannot do a COMMIT or ROLLBACK on them.

Temporary LOB locators can span transactions. They are deleted when the server abnormally terminates, and when an error is returned from a database SQL operation.

LOB Buffering Subsystem

The *LBS* (LOB Buffering Subsystem) is an area of user memory provided for use as a buffer for one or more LOBs in the client's address space.

Buffering has these advantages, especially for applications on a client that does many small reads and writes to specific regions of the LOB:

- The LBS reduces round-trips to the server because you fill the buffer with multiple reads / writes to the LOBs, and then write to the server when a FLUSH directive is executed.
- Buffering also reduces the total number of LOB updates on the server. This creates better LOB performance and saves disk space.

Oracle provides a simple buffer subsystem, not a cache. Oracle does not guarantee that the contents of a buffer are always synchronized with the server LOB value. Use the FLUSH statement to actually write updates in the server LOB.

Buffered reads / writes of a LOB are performed through its locator. A locator enabled for buffering provides a consistent read version of the LOB until you perform a write through that locator.

After being used for a buffered WRITE, a locator becomes an updated locator and provides access to the latest LOB version *as seen through the buffering subsystem*. All further buffered WRITEs to the LOB can be done only through this updated locator. Transactions involving buffered LOB operations cannot migrate across user sessions.

The LBS is managed by the user, who is responsible for updating server LOB values by using FLUSH statements to update them. The LBS is single-user and single-threaded. Use ROLLBACK and SAVEPOINT actions to guarantee correctness in the server LOBs. Transactional support for buffered LOB operations is not guaranteed by Oracle. To ensure transactional semantics for buffered LOB updates, you must maintain logical savepoints to perform a rollback in the event of an error.

For more information on the LBS, see *Oracle9i Application Developer's Guide - Fundamentals*.

How to Use LOBs

There are two methods available to access LOBs in Pro*COBOL:

- The DBMS_LOB package inside PL/SQL blocks.
- Embedded SQL statements.

The imbedded SQL statements are designed to give users a functional equivalent to the PL/SQL interface.

The following table compares LOB access in PL/SQL and embedded SQL statements in Pro*COBOL. Empty boxes indicate missing functionality.

Table 13–1 LOB Access Methods

PL/SQL[1]	Pro*COBOL Embedded SQL
COMPARE()	
INSTR()	
SUBSTR()	
APPEND()	APPEND
:=	ASSIGN
CLOSE()	CLOSE
COPY()	COPY
CREATETEMPORARY()	CREATE TEMPORARY
	DISABLE BUFFERING
	ENABLE BUFFERING

Table 13–1 LOB Access Methods

PL/SQL[1]	Pro*COBOL Embedded SQL
ERASE()	ERASE
GETCHUNKSIZE()	DESCRIBE
ISOPEN()	DESCRIBE
FILECLOSE()	CLOSE
FILECLOSEALL()	FILE CLOSE ALL
FILEEXISTS()	DESCRIBE
FILEGETNAME()	DESCRIBE
FILEISOPEN()	DESCRIBE
FILEOPEN()	OPEN
BFILENAME()	FILE SET[2]
	FLUSH BUFFER
FREETEMPORARY()	FREE TEMPORARY
GETLENGTH()	DESCRIBE
=	
ISTEMPORARY()	DESCRIBE
LOADFROMFILE()	LOAD FROM FILE
OPEN()	OPEN
READ()	READ
TRIM()	TRIM
WRITE()	WRITE
WRITEAPPEND()	WRITE

[1] From dbmslob.sql. All routines are prefixed with 'DBMS_LOB.' except BFILENAME.

[2] The BFILENAME() built in SQL function may also be used.

Note: You must explicitly lock the row before using any of the new statements that modify or change a LOB in any way. Operations that can modify a LOB value are APPEND, COPY, ERASE, LOAD FROM FILE, TRIM, and WRITE.

LOB Locators in Your Application

To use LOB locators in your Pro*COBOL application use these pseudo-types:

- SQL-BLOB
- SQL-CLOB
- SQL-NCLOB
- SQL-BFILE

For example, to declare an NCLOB variable called MY-NCLOB:

```
01  MY-NCLOB    SQL-NCLOB.
```

Initializing a LOB

Internal LOBs

To initialize a BLOB to empty, use the *EMPTY_BLOB()* function or use the ALLOCATE SQL statement. For CLOBs and NCLOBs, use the *EMPTY_CLOB()* function. See *Oracle9i SQL Reference* for more about EMPTY_BLOB() and EMPTY_CLOB(). These functions are permitted only in the VALUES clause of an INSERT statement or as the source of the SET clause in an UPDATE statement.

For example:

```
    EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
       VALUES (EMPTY_BLOB(), EMPTY_CLOB()) END-EXEC.
```

The ALLOCATE statement allocates a LOB locator and initializes it to empty, so, the following code is equivalent to the previous example:

```
...
 01  A-BLOB      SQL-BLOB.
 01  A-CLOB      SQL-CLOB.
...
    EXEC SQL ALLOCATE :A-BLOB END-EXEC.
    EXEC SQL ALLOCATE :A-CLOB END-EXEC.
    EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
       VALUES (:A-BLOB, :A-CLOB) END-EXEC.
```

External LOBs

Use the LOB FILE SET statement to initialize the DIRECTORY alias of the BFILE and FILENAME this way:

```
...
 01  ALIAS        PIC X(14) VARYING.
 01  FILENAME     PIC X(14) VARYING.
 01  A-BFILE      SQL-BFILE.
 ...
     MOVE "lob_dir" TO ALIAS-ARR.
     MOVE 7 TO ALIAS-LEN.
     MOVE "image.gif" TO FILENAME-ARR
     MOVE 9 TO FILENAME-LEN..
     EXEC SQL ALLOCATE :A-BFILE END-EXEC.
     EXEC SQL LOB FILE SET :A-BFILE
        DIRECTORY = :ALIAS, FILENAME = :FILENAME END-EXEC.
     EXEC SQL INSERT INTO file_table (a_bfile) VALUES (:A-BFILE) END-EXEC.
```

Refer to *Oracle9i Application Developer's Guide - Fundamentals* for a complete description of DIRECTORY object naming conventions and DIRECTORY object privileges.

Alternatively, you can use the BFILENAME('directory', 'filename') function in an INSERT or UPDATE statement to initialize a BFILE column or attribute for a particular row, and give the name of the actual physical directory and filename:

```
EXEC SQL INSERT INTO file_table (a_bfile)
   VALUES (BFILENAME('lob_dir', 'image.gif'))
      RETURNING a_bfile INTO :A-BFILE END-EXEC.
```

Note: BFILENAME() does not check permissions on the directory or filename, or whether the physical directory actually exists. Subsequent file accesses that use the BFILE locator will do those checks and return an error if the file is inaccessible.

Temporary LOBs

A temporary LOB is initialized to empty when it is first created using the embedded SQL LOB CREATE TEMPORARY statement. The EMPTY_BLOB() and EMPTY_CLOB() functions cannot be used with temporary LOBs.

Freeing LOBs

The FREE statement is used to free the memory used by an ALLOCATE statement:

```
EXEC SQL FREE :A-BLOB END-EXEC.
```

Rules for LOB Statements

Here are the rules for using LOB statements:

For All LOB Statements

These general restrictions and limitations apply when manipulating LOBs with the SQL LOB statements:

- The FOR clause is not allowed in embedded SQL LOB statements. Only one LOB locator can be used in those statement. However, the ALLOCATE and FREE statements do allow FOR clauses.
- Distributed LOBs are not supported. Although you may use the AT database clause in any of the new embedded SQL LOB statements, you cannot mix LOB locators that were created or allocated using different database connections in the same SQL LOB statement.

For the LOB Buffering Subsystem

For the LBS, these rules must be followed:

- Errors in read or write accesses are reported at the next access to the server. Therefore, error recovery has to be coded by you, the user.
- When updating a LOB with buffered writes, do not update the same LOB with a method that bypasses the LOB Buffering Subsystem.
- An updated LOB locator enabled for buffering can be passed as an IN parameter to a PL/SQL procedure, but not as an IN OUT or OUT parameter. An error is returned. An error is also returned when there is an attempt to return an updated locator.
- An ASSIGN of an *updated* locator enabled for buffering to another locator is not allowed.
- You can append to the LOB value with buffered writes, but the starting offset must be one character after the end of the LOB. The LBS does not allow APPEND statements resulting in zero-byte fillers or spaces in LOBs in the database server.
- The character sets of the host locator bind variable and the database server CLOB must be the same.
- Only ASSIGN, READ and WRITE statements work with a locator enabled for buffering.
- The following statements result in errors when used with a locator enabled for buffering: APPEND, COPY, ERASE, DESCRIBE (LENGTH only), SELECT, and TRIM. Errors are also returned when you use these statements with a locator

that is not enabled for buffering, if the LOB pointed to by the locator is being accessed in buffered mode by another locator.

Note: The FLUSH statement must be used on a LOB enabled by the LOB Buffering Subsystem before

- Committing the transaction.
- Migrating from the current transaction to another.
- Disabling buffer operations on a LOB.

For Host Variables

Use the following rules and notes for the LOB statements:

- *src* and *dst* can refer to either internal or external LOB locators, but *file* refers only to external locators.
- Numeric host values (`amt, src_offset, dst_offset`, and so forth.) are declared `as` a 4-byte integer variable, PIC S9(9) COMP. The values are restricted between 0 and 4 Gigabytes.
- The concept of NULL is part of a LOB locator. There is no need for indicator variables in the LOB statements. NULL cannot be used with numeric value variables such as `amt, src_offset`, and so forth and result in an error.
- The offset values `src_offset` and `dst_offset` have default values 1.

  Note: BLOB, CLOB, and NCLOB variables need to respect the alignment requirements of your platform. Refer to you platform documentation on alignment restrictions of your particular platform.

LOB Statements

The statements are presented alphabetically. In all the statements where it appears, *database* refers to a database connection

APPEND

Purpose

The APPEND statement appends a LOB value at the end of another LOB.

Syntax

```
EXEC SQL [AT [:]database] LOB APPEND :src TO :dst END-EXEC.
```

Host Variables

src (IN)

An internal LOB locator uniquely referencing the source LOB.

dst (IN OUT)

An internal LOB locator uniquely referencing the destination LOB.

Usage Notes

The data is copied from the source LOB to the end of the destination LOB, extending the destination LOB up to a maximum of 4 Gigabytes. If the LOB is extended beyond 4 Gigabytes, an error will occur.

The source and destination LOBs must already exist and the destination LOB must be initialized.

Both the source and destination LOBs must be of the same internal LOB type. It is an error to have enabled LOB buffering for either type of locator.

ASSIGN

Purpose

Assigns a LOB or BFILE locator to another.

Syntax

```
EXEC SQL [AT [:]database] LOB ASSIGN :src to :dst END-EXEC.
```

Host Variables

src (IN)

LOB or BFILE locator source copied from.

dst (IN OUT)

LOB or BFILE locator copied to.

Usage Notes

After the assignment, both locators refer to the same LOB value. The destination LOB locator must be a valid initialized (allocated) locator.

For internal LOBs, the source locator's LOB value is copied to the destination locator's LOB value only when the destination locator is stored in the table. For Pro*COBOL, issuing a FLUSH of an object containing the destination locator will copy the LOB value.

An error is returned when a BFILE locator is assigned to an internal LOB locator and vice-versa. It is also an error if the *src* and *dst* LOBs are not of the same type.

If the source locator is for an internal LOB that was enabled for buffering, and the source locator has been used to modify the LOB value through the LOB Buffering Subsystem, and the buffers have not been flushed since the WRITE, then the source locator cannot be assigned to the destination locator. This is because only one locator for each LOB can modify the LOB value through the LOB Buffering Subsystem.

CLOSE

Purpose

Close an open LOB or BFILE.

Syntax

```
EXEC SQL [AT [:]database] LOB CLOSE :src END-EXEC.
```

Host Variables

src (IN OUT)

The locator of the LOB or BFILE to be closed.

Usage Notes

It is an error to close the same LOB twice either with different locators or with the same locator. For external LOBs, no error is produced if the BFILE exists but has not been opened.

It is an error to COMMIT a transaction before closing all previously opened LOBs. At transaction ROLLBACK time, all LOBs that are still open will be discarded without first being closed.

COPY

Purpose

Copy all or part of a LOB value into a second LOB.

Syntax

```
EXEC SQL [AT [:]database] LOB COPY :amt FROM :src [AT :src_offset]
   TO :dst [AT :dst_offset] END-EXEC.
```

Host Variables

amt (IN)

The maximum number of bytes for BLOBs, or characters for CLOBs and NCLOBs, to copy.

src (IN)

The locator of the source LOB.

src_offset (IN)

This is the number of characters for CLOB or NCLOB, and the number of bytes for a BLOB, starting from 1 at the beginning of the LOB.

dst (IN)

The locator of the destination LOB.

dst_offset (IN)

The destination offset. Same rules as for src_offset.

Usage Notes

If the data already exists at the destination's offset and beyond, it is overwritten with the source data. If the destination's offset is beyond the end of the current data, zero-byte fillers (BLOBs) or spaces (CLOBs) are written into the destination LOB from the end of the current data to the beginning of the newly written data from the source.

The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is a runtime error to extend this LOB beyond 4 Gigabytes.

It is also an error to try to copy from a LOB that is not initialized.

Both the source and destination LOBs must be of the same type. LOB buffering must not be enabled for either locator.

The `amt` variable indicates the maximum amount to copy. If the end of the source LOB is reached before the specified amount is copied, the operation terminates without an error.

To make a temporary LOB permanent, the COPY statement must be used to explicitly COPY the temporary LOB into a permanent one.

CREATE TEMPORARY

Purpose

Creates a temporary LOB.

Syntax

```
EXEC SQL [AT [:]database] LOB CREATE TEMPORARY :src END-EXEC.
```

Host Variables

src (IN OUT)

Before execution, when IN, `src` is a LOB locator previously allocated.

After execution, when OUT, `src` is a LOB locator that will point to a new empty temporary LOB.

Usage Notes

After successful execution, the locator points to a newly created temporary LOB that resides on the database server independent of a table. The temporary LOB is empty and has zero length.

At the end of a session, all temporary LOBs are freed. Reads and Writes to temporary LOBs never go through the buffer cache.

DISABLE BUFFERING

Purpose

Disables LOB buffering for the LOB locator.

Syntax

```
EXEC SQL [AT [:]database] LOB DISABLE BUFFERING :src END-EXEC.
```

Host Variable

src (IN OUT)

An internal LOB locator.

Usage Notes

This statement does not support BFILEs. Subsequent reads or writes will not be done through the LBS.

Note: Use a FLUSH BUFFER command to make changes permanent, since the DISABLE BUFFERING statement does not implicitly flush the changes made in the LOB Buffering Subsystem.

ENABLE BUFFERING

Purpose

Enables LOB buffering for the LOB locator.

Syntax

```
EXEC SQL [AT [:]database] LOB ENABLE BUFFERING :src END-EXEC.
```

Host Variable

src (IN OUT)

An internal LOB locator.

Usage Notes

This statement does not support BFILEs. Subsequent reads and writes are done through the LBS.

ERASE

Purpose

Erases a given amount of LOB data starting from a given offset.

Syntax

```
EXEC SQL [AT [:]database] LOB ERASE :amt
   FROM :src [AT :src_offset] END-EXEC.
```

Host Variables

amt (IN OUT)

The input is the number of bytes or characters to erase. The returned output is the actual number erased.

src (IN OUT)

An internal LOB locator.

src_offset (IN)

The offset from the beginning of the LOB, starting from 1.

Usage Notes

This statement does not support BFILEs.

After execution, *amt* returns the actual number of characters / bytes that were erased. The actual number and requested number will differ if the end of the LOB value is reached before erasing the requested number of characters / bytes. If the LOB is empty, *amt* will indicate that 0 characters / bytes were erased.

For BLOBs, erasing means zero-byte fillers overwrite the existing LOB value. For CLOBs, erasing means that spaces overwrite the existing LOB value.

FILE CLOSE ALL

Purpose

Closes all BFILES opened in the current session.

Syntax

```
EXEC SQL [AT [:]database] LOB FILE CLOSE ALL END-EXEC.
```

Usage Notes

If there are any open files in the session whose closure has not been handled properly, you can use the FILE CLOSE ALL statement to close all files opened in the session and resume file operations from the beginning.

FILE SET

Purpose

Sets DIRECTORY alias and FILENAME in a BFILE locator.

Syntax

```
EXEC SQL [AT [:]database] LOB FILE SET :file
   DIRECTORY = :alias, FILENAME = :filename END-EXEC.
```

Host Variables

file (IN OUT)

BFILE locator where the DIRECTORY alias and FILENAME is set.

alias (IN)

DIRECTORY alias name to set.

filename (IN)

The FILENAME to set.

Usage Notes

The given BFILE locator must be first ALLOCATEd prior to its use in this statement.

Both the DIRECTORY alias name and FILENAME must be provided.

The maximum length of the DIRECTORY alias is 30 bytes. The maximum length of the FILENAME is 255 bytes.

The only external datatypes supported for use with the DIRECTORY alias name and FILENAME attributes are VARCHAR, VARCHAR2 and CHARF.

It is an error to use this statement with anything but an external LOB locator.

FLUSH BUFFER

Purpose

Writes this LOB's buffers to the database server.

Syntax

```
EXEC SQL [AT [:]database] LOB FLUSH BUFFER :src [FREE] END-EXEC.
```

Host Variables

src (IN OUT)

Internal LOB locator.

Usage Notes

Writes the buffer data to the database LOB in the server from the LOB referenced by the input locator.

LOB buffering must have already been enabled for the input LOB locator.

The FLUSH operation, by default, does not free the buffer resources for reallocation to another buffered LOB operation. However, if you want to free the buffer explicitly, you can include the optional FREE keyword to so indicate.

FREE TEMPORARY

Purpose

Free the temporary space for the LOB locator.

Syntax

```
EXEC SQL [AT [:]database] LOB FREE TEMPORARY :src END-EXEC.
```

Host Variable

src (IN OUT)

The LOB locator pointing to the temporary LOB.

Usage Notes

The input locator must point to a temporary LOB. The output locator is marked not initialized and can be used in subsequent LOB statements.

LOAD FROM FILE

Purpose

Copy all or a part of a BFILE into an internal LOB.

Syntax

```
EXEC SQL [AT [:]database] LOB LOAD :amt
   FROM FILE :file [AT :src_offset] INTO :dst [AT :dst_offset] END-EXEC.
```

Host Variables

amt (IN)

Maximum number of bytes to be loaded.

file (IN OUT)

The source BFILE locator.

src_offset (IN)

The number of bytes offset from the beginning of the file, starting from 1.

dst (IN OUT)

The destination LOB locator which can be BLOB, CLOB, be NCLOB.

dst_offset (IN)

The number of bytes (for BLOBs) or characters (CLOBs and NCLOBs) from the beginning of the destination LOB where writing will begin. It starts at 1.

Usage Notes

The data is copied from the source BFILE to the destination internal LOB. No character set conversions are performed when copying the BFILE data to a CLOB or NCLOB. Therefore, the BFILE data must already be in the same character set as the CLOB or NCLOB in the database.

The source and destination LOBs must already exist. If the data already exists at the destination's start position, it is overwritten with the source data. If the destination's start position is beyond the end of the current data, zero-byte fillers (BLOBs) or spaces (CLOBs and NCLOBs) are written into the destination LOB from the end of the data to the beginning of the newly written data from the source.

The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend this LOB beyond 4 Gigabytes.

It is also an error to copy from a BFILE that is not initialized.

The amount parameter indicates the maximum amount to load. If the end of the source BFILE is reached before the specified amount is loaded, the operation terminates without error.

OPEN

Purpose

Open a LOB or BFILE for read or read / write access.

Syntax

```
EXEC SQL [AT [:]database] LOB OPEN :src
   [ READ ONLY | READ WRITE ] END-EXEC.
```

Host Variables

src (IN OUT)

LOB locator of the LOB or BFILE.

Usage Notes

The default mode in which a LOB or BFILE can be Opened is for READ ONLY access.

For internal LOBs, being OPEN is associated with the LOB, not with the locator. Assigning an already Opened locator to another locator does not count as OPENing a new LOB. Instead, both locators refer to the same LOB. For BFILEs, being OPEN is associated with the locator.

Only 32 LOBs can be OPEN at any one time. An error will be returned when the 33rd LOB is Opened.

There is no support for writable BFILEs. Therefore, when you OPEN a BFILE in READ WRITE mode, an error is returned.

It is also an error to open a LOB in READ ONLY mode and then attempt to WRITE to the LOB.

READ

Purpose

Reads all or part of a LOB or BFILE into a buffer.

Syntax

```
EXEC SQL [AT [:]database] LOB READ :amt FROM :src [AT :src_offset]
   INTO :buffer [WITH LENGTH :buflen] END-EXEC.
```

Host Variables

amt (IN OUT)

The input is the number of characters or bytes to be read. The output is the actual number of characters or bytes that were read.

If the amount of bytes to be read is larger than the buffer length it is assumed that the LOB is being READ in a polling mode. On input if this value is 0, then the data will be read in a polling mode from the input offset until the end of the LOB.

The number of bytes or characters actually read is returned in `amt`. If the data is read in pieces, `amt` will always contain the length of the last piece read.

When the end of a LOB is reached an *ORA-1403: no data found* error will be issued.

When reading in a polling mode, the application must invoke the LOB READ repeatedly to read more pieces of the LOB until no more data is left. Control the use of the polling mode with the NOT FOUND condition in a WHENEVER directive to catch the ORA-1403 error.

src (IN)

The LOB or BFILE locator.

src_offset (IN)

This is the absolute offset from the beginning of the LOB value from which to start reading. For character LOBs it is the number of characters from the beginning of the LOB. For binary LOBs or BFILEs it is the number of bytes. The first position is 1.

buffer (IN / OUT)

A buffer into which the LOB data will be read. The external datatype of the buffer is restricted to only a few types depending on the type of the source LOB. The maximum length of the buffer depends on the external datatype being used to store the LOB value. The following table summarizes the legal external datatypes and their corresponding maximum lengths categorized by source LOB type:

Table 13–2 Source LOB and Precompiler Datatypes

External LOB[1]	Internal LOB	Precompiler External Datatype	Precompiler Maximum Length [2]	PL/SQL Datatype	PL/SQL Maximum Length
	BLOB	RAW	65535	RAW	32767
		VARRAW	65533		
		LONG RAW	2147483647		
BFILE		LONG VARRAW	2147483643		
	CLOB	VARCHAR2	65535	VARCHAR2	32767
		VARCHAR	65533		
		LONG VARCHAR	2147483643		
	NCLOB	NVARCHAR2	4000	NVARCHAR2	4000

[1] Any of the external datatypes shown can be used with BFILES.

[2] Lengths are measured in bytes, not characters.

buflen (IN)

Specifies the length of the given buffer when it cannot be determined otherwise.

Usage Notes

A BFILE must already exist on the database server and must have been opened using the input locator. The database must have permission to read the file and the user must have read permission on the directory.

It is an error to try to read from an un-initialized LOB or BFILE.

The length of the buffer is determined this way:

- From `buflen`, when the WITH LENGTH clause is present.
- In the absence of the WITH LENGTH clause, the length is determined by treating the buffer host variable in OUT mode according to the rules in "Handling Character Data" on page 4-1.

TRIM

Purpose

Truncates the LOB value.

Syntax

```
EXEC SQL [AT [:]database] LOB TRIM :src TO :newlen END-EXEC.
```

Host Variables

src (IN OUT)

LOB locator for internal LOB.

newlen (IN)

The new length of the LOB value.

Usage Notes

This statement is not for BFILES. The new length cannot be greater than the current length, or an error is returned.

WRITE

Purpose

Writes the contents of a buffer to a LOB.

Syntax

```
EXEC SQL [AT [:]database] LOB WRITE [APPEND] [ FIRST | NEXT | LAST | ONE ]
   :amt FROM :buffer [WITH LENGTH :buflen]
      INTO :dst [AT :dst_offset] END-EXEC.
```

Host Variables

amt (IN OUT)

The input is the number of characters or bytes to be written.

The output is the actual number of characters or bytes that is written.

When writing using a polling method, `amt` will return the cumulative total length written for the execution of the WRITE statement after a WRITE LAST is executed. If the WRITE statement is interrupted, `amt` will be undefined.

buffer (IN)

A buffer from which the LOB data is written. See "READ" on page 13-20 for the lengths of datatypes.

dst (IN OUT)

The LOB locator.

dst_offset (IN)

The offset from the beginning of the LOB (counting from 1), in characters for CLOBs and NCLOBs, in bytes for BLOBs.

buflen (IN)

The buffer length when it cannot be calculated in any other way.

Usage Notes

If LOB data already exists, it is overwritten with the data stored in the buffer. If the offset specified is beyond the end of the data currently in the LOB, zero-byte fillers or spaces are inserted into the LOB.

Specifying the keyword APPEND in the WRITE statement causes the data to automatically be written to the end of the LOB. When APPEND is specified, the destination offset is assumed to be the end of the LOB. It is an error to specify the destination offset when using the APPEND option in the WRITE statement.

The buffer can be written to the LOB in one piece (using the ONE orientation which is the default) or it can be provided piece-wise using a standard polling method.

Polling is begun by using FIRST, then NEXT to write subsequent pieces. The LAST keyword is used to write the final piece that terminates the write.

Using this piece-wise write mode, the buffer and the length can be different in each call if the pieces are of different sizes and from different locations.

If the total amount of data passed to Oracle is less than the amount specified by the `amt` parameter after doing all the writes, an error results.

The same rules apply for determining the buffer length as in the READ statement. See "READ" on page 13-20.

DESCRIBE

Purpose

This is a statement that is equivalent to several OCI and PL/SQL statements (which is why it is saved for last). Use the LOB DESCRIBE SQL statement to retrieve attributes from a LOB. This capability is similar to OCI and PL/SQL procedures. The LOB DESCRIBE statement has this format:

Syntax

```
EXEC SQL [AT [:]database] LOB DESCRIBE :src GET attribute1 [{, attributeN}]
   INTO :hv1 [[INDICATOR] :hv_ind1] [{, :hvN [[INDICATOR] :hv_indN] }]
      END-EXEC.
```

where an attribute can be any of these choices:

```
CHUNKSIZE | DIRECTORY | FILEEXISTS | FILENAME | ISOPEN | ISTEMPORARY | LENGTH
```

Host variables

src (IN)

The LOB locator of an internal or external LOB.

hv1 ... hvN (OUT)

The host variables that receive the attribute values, in the order specified in the attribute name list.

hv_ind1 ... hv_indN (OUT)

Optional host variables that receive the indicator NULL status in the order of the attribute name list.

The following table describes the attributes, which LOB it is associated with, and the COBOL types into which they should be read:

Table 13–3 LOB Attributes

LOB Attribute	Attribute Description	Restrictions	COBOL Type
CHUNKSIZE	The amount (in bytes for BLOBs and characters for CLOBs / NCLOBs) of space used in the LOB chunk to store the LOB value. You speed up performance if you issue READ / WRITE requests using a multiple of this chunk size. If all writes are done on a chunk basis, no extra / excess versioning is done nor duplicated. Users could batch up the WRITE until they have enough for a chunk instead of issuing several WRITE calls for the same CHUNK.	BLOBs, CLOBs and NCLOBs only	PIC S9(9) COMP
DIRECTORY	The name of the DIRECTORY alias for the BFILE. The length, n, is between 1 and 30 bytes. Use that length in the picture.	FILE LOBs only	PIC X(n) [VARYING]
FILEEXISTS	Determines whether or not the BFILE exists on the server's operating system file system. FILEEXISTS is true when it is nonzero; false when it equals 0.	FILE LOBs only	PIC S9(9) COMP
FILENAME	The name of the BFILE. The length, n, is between 1 and 255 bytes. Use that length in the picture.	FILE LOBs only	PIC X(n) [VARYING]
ISOPEN	For BFILEs, if the input BFILE locator was never used in an OPEN statement, the BFILE is considered not to be OPENed by this locator. However, a different BFILE locator may have OPENed the BFILE. More than one OPEN can be performed on the same BFILE using different locators. For LOBs, if a different locator opened the LOB, the LOB is still considered to be OPEN by the input locator. ISOPEN is true when it is nonzero; false when it equals 0.		PIC S9(9) COMP
ISTEMPORARY	Determines whether or not the input LOB locator refers to a temporary LOB or not. ISTEMPORARY is true when it is nonzero; false when it equals 0.	BLOBs, CLOBs and NCLOBs only	PIC S9(9) COMP
LENGTH	Length of BLOBs and BFILEs in bytes, CLOBs and NCLOBs in characters. For BFILEs, the length includes EOF if it exists. Empty internal LOBs have zero length. LOBs / BFILEs that are not initialized have undefined length.		PIC 9(9) COMP

Usage Notes

Indicator variables should be declared as PIC S9(4) COMP. After execution has completed, SQLERRD(3) contains the number of attributes retrieved without error.

If there was an execution error, the attribute at which it occurred is one more than the contents of SQLERRD(3).

DESCRIBE Example

Here is a simple Pro*COBOL example that extracts the DIRECTORY and FILENAME attributes of a given BFILE:

```
...
 01  A-BFILE      SQL-BFILE.
 01  DIRECTORY    PIC X(30) VARYING.
 01  FILENAME     PIC X(30) VARYING.
 01  D-IND        PIC S9(4) COMP.
 01  F-IND        PIC S9(4) COMP.
 01  FEXISTS      PIC S9(9) COMP.
 01  ISOPN        PIC S9(9) COMP.
...
```

Finally, select a BFILE locator from some LOB table and do the DESCRIBE:

```
EXEC SQL ALLOCATE :A-BFILE END-EXEC.
EXEC SQL INSERT INTO lob_table (a_bfile) VALUES (BFILENAME ('lob.dir',
   'image.gif')) END-EXEC.
EXEC SQL SELECT a_bfile INTO :A-BFILE FROM lob_table WHERE ... END-EXEC.
EXEC SQL DESCRIBE :A-BFILE GET DIRECTORY, FILENAME, FILEEXISTS, ISOPEN
   INTO :DIRECTORY:D-IND, :FILENAME:F-IND, FEXISTS, ISOPN ND-EXEC.
```

Indicator variables are valid for use only with the DIRECTORY and FILENAME attributes. These attributes are character strings whose values may be truncated if the host variable buffers used to hold their values are not large enough. When truncation occurs, the value of the indicator will be set to the original length of the attribute.

READ and WRITE Using the Polling Method

Here is an outline of using READ with the polling method:

Start the read polling by setting the amount to zero in the first LOB READ (or set the amount to the size of the total data to be read). The amount is first set to zero in this case outline which omits details:

```
EXEC SQL ALLOCATE :CLOB1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND GOTO END-OF-CLOB END-EXEC.
```

```
     EXEC SQL SELECT A_CLOB INTO :CLOB1 FROM LOB_TABLE WHERE ... END-EXEC.

     MOVE 0 TO AMT.
     EXEC SQL LOB READ :AMT FROM :VLOB1 AT :OFFSET INTO :BUFFER END-EXEC.

READ-LOOP.
     EXEC SQL LOB READ :AMT FROM :CLOB1 INTO BUFFER $END-EXEC.
     GO TO READ-LOOP.

END-OF-CLOB.
     EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

     EXEC SQL FREE :CLOB1 END-EXEC.
```

The following code outline writes data from a buffer into an internal CLOB. The value of AMT (16 characters) in the initial write statement should be the length of the entire data you will write. The buffer is 5 characters long.

If EOF is read in the initial read, then do the LOB WRITE ONE. If not, start polling with a LOB WRITE FIRST of the buffer. Read the data, and do a LOB WRITE NEXT of the output. No offset is needed in the LOB WRITE NEXT because data is written at the end of the last write. After EOF is read, break out of the read loop and do a LOB WRITE LAST. The amount returned must equal the initial amount value (16).

```
    MOVE 16 TO AMT.
    PERFORM READ-NEXT-RECORD.
    MOVE INREC TO BUFFER-ARR.
    MOVE 5 TO BUFFER-LEN.
    IF (END-OF-FILE = "Y")
        EXEC SQL LOB WRITE ONE :AMT FROM :BUFFER INTO CLOB1
           AT :OFFSET END-EXEC.
        PERFORM DISPLAY-CLOB
    ELSE
        EXEC SQL LOB WRITE FIRST :AMT FROM :BUFFER INTO :CLOB1
           AT :OFFSET END-EXEC.
    PERFORM READ-NEXT-RECORD.
    PERFORM WRITE-TO-CLOB
       UNTIL END-OF-FILE = "Y".
    MOVE INREC TO BUFFER-ARR.
    MOVE 1 TO BUFFER-LEN.
    EXEC SQL LOB WRITE LAST :AMT FROM :BUFFER INTO :CLOB1 END-EXEC.
    PERFORM DISPLAY-CLOB.
    ...
WRITE-TO-CLOB.
    MOVE INREC TO BUFFER-ARR.
```

```
    MOVE 5 TO BUFFER-LEN.
    EXEC SQL LOB WRITE NEXT :AMT FROM :BUFFER INTO :CLOB1 END-EXEC.
    PERFORM READ-NEXT RECORD.

READ-NEXT-RECORD.
    MOVE SPACES TO INREC.
    READ INFILE NEXT RECORD
       AT END
       MOVE "Y" TO END-OF-FILE.
...
```

LOB Sample Program: LOBDEMO1.PCO

The LOBDEMO1.PCO program illustrates several LOB embedded SQL statements. The source code is in your `demo` directory. The application uses a table named `license_table` whose columns are social security number, name, and a CLOB containing text summarizing driving offenses. Several simplified SQL operations of a typical motor vehicle department are modeled.

The possible actions are:

- Add new records.
- List records by social security number.
- List information in a record, given a social security number.
- Append a new traffic violation to an existing CLOB's contents.

Here is the listing of LOBDEMO1.PCO:

```
*************************************************************************
* LOB Demo 1: DMV Database                                              *
*                                                                       *
* SCENARIO:                                                             *
*                                                                       *
* We consider the example of a database used to store driver's          *
* licenses. The licenses are stored as rows of a table containing       *
* three columns: the sss number of a person, his/her name and the       *
* text summary of the info found in his license.                        *
*                                                                       *
* The sss number and the name are the unique social security number     *
* and name of an individual.  The text summary is a summary of the      *
* information on the individual, including his driving record,          *
* which can be arbitrarily long and may contain comments and data       *
* regarding the person's driving ability.                               *
```

```
*                                                                *
* APPLICATION OVERVIEW:                                          *
*                                                                *
* This example demonstrate how a Pro*COBOL client can handle the *
* new LOB datatypes.  Demonstrated are the mechanisms for accessing *
* and storing lobs to/from tables.                               *
*                                                                *
*  To run the demo:                                              *
*                                                                *
*  1. Execute the script, lobdemo1.sql in Server Manager         *
*  2. Precompile using Pro*COBOL                                 *
*       procob lobdemo1                                          *
*  3. Compile/Link (This step is platform specific)              *
*                                                                *
* lobdemo1.sql contains the following SQL statements:            *
*                                                                *
* connect scott/tiger;                                           *
*                                                                *
* drop table license_table;                                      *
*                                                                *
* create table license_table(                                    *
* sss char(9),                                                   *
* name varchar2(50),                                             *
* txt_summary clob);                                             *
*                                                                *
* insert into license_table                                      *
*  values('971517006', 'Dennis Kernighan',                       *
*  'Wearing a Bright Orange Shirt');                             *
*                                                                *
* insert into license_table                                      *
*  values('555001212', 'Eight H. Number',                        *
*  'Driving Under the Influence');                               *
*                                                                *
* insert into license_table                                      *
*  values('010101010', 'P. Doughboy',                            *
*  'Impersonating An Oracle Employee');                          *
*                                                                *
* insert into license_table                                      *
*  values('555377012', 'Calvin N. Hobbes',                       *
*  'Driving Under the Influence');                               *
*                                                                *
* The main program provides the menu of actions that can be      *
* performed.  The program stops when the number 5 (Quit) option  *
* is entered.  Depending on the input, this main program calls   *
* the appropriate nested program to execute the chosen action.  *
```

```
*                                                                       *
************************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  LOBDEMO1.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  USERNAME       PIC X(5).
 01  PASSWD         PIC X(5).
 01  CHOICE         PIC 9 VALUE 0.
 01  SSS            PIC X(9).
 01  SSSEXISTS      PIC 9 VALUE ZERO.
 01  LICENSE-TXT    SQL-CLOB .
 01  NEWCRIME       PIC X(35) VARYING.
 01  SSSCOUNT       PIC S9(4) COMP.
 01  THE-STRING    PIC X(200) VARYING.
 01  TXT-LENGTH    PIC S9(9) COMP.
 01  CRIMES.
     05 FILLER PIC X(35) VALUE "Driving Under the Influence".
     05 FILLER PIC X(35) VALUE "Grand Theft Auto".
     05 FILLER PIC X(35) VALUE "Driving Without a License".
     05 FILLER PIC X(35) VALUE
            "Impersonating an Oracle Employee".
     05 FILLER PIC X(35) VALUE "Wearing a Bright Orange Shirt".
 01  CRIMELIST REDEFINES CRIMES.
     05 CRIME  PIC X(35) OCCURS 5 TIMES.
 01  CRIME-INDEX   PIC 9.
 01  TXT-LEN       PIC S9(9) COMP.
 01  CRIME-LEN     PIC S9(9) COMP.
 01  NAME1         PIC X(50) VARYING.
 01  NEWNAME         PIC X(50).
************************************************************************

     EXEC SQL INCLUDE SQLCA END-EXEC.

 PROCEDURE DIVISION.
 A000-CONTROL SECTION.
************************************************************************
*  A000-CONTROL
*     Overall control section
************************************************************************
 A000-CNTRL.
        EXEC SQL
WHENEVER SQLERROR DO PERFORM Z900-SQLERROR
      END-EXEC.
        PERFORM B000-LOGON.
```

```
PERFORM C000-MAIN UNTIL CHOICE = 5.
PERFORM D000-LOGOFF.
 A000-EXIT.
STOP RUN.

 B000-LOGON SECTION.
*************************************************************************
*  B000-LOGON
*    Log on to database.
*************************************************************************
 B000-LGN.
     DISPLAY '************************************************'.
     DISPLAY '*            Welcome to the DMV Database         *'.
     DISPLAY '************************************************'.
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSWD.
     EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
     END-EXEC.
     DISPLAY " ".
     DISPLAY "Connecting to license database account:  ",
        USERNAME, "/", PASSWD.
     DISPLAY " ".
 B000-EXIT.
     EXIT.
 C000-MAIN SECTION.
*************************************************************************
*  C000-MAIN
*    Display the main menu and action requests
*************************************************************************
 C000-MN.

     DISPLAY " ".
     DISPLAY "License Options:".
     DISPLAY "1. List available records by SSS number".
     DISPLAY "2. Get information on a particular record".
     DISPLAY "3. Add crime to a record".
     DISPLAY "4. Insert new record to database".
     DISPLAY "5. Quit".
     DISPLAY " ".

     MOVE ZERO TO CHOICE.
     PERFORM Z300-ACCEPT-CHOICE UNTIL CHOICE < 6
                             AND CHOICE > 0.
     IF (CHOICE = 1)
```

```
               PERFORM C100-LIST-RECORDS.
           IF (CHOICE = 2)
               PERFORM C200-GET-RECORD.
           IF (CHOICE = 3)
               PERFORM C300-ADD-CRIME.
           IF (CHOICE = 4)
               PERFORM C400-NEW-RECORD.
       C000-EXIT.
   EXIT.

       C100-LIST-RECORDS SECTION.
      ************************************************************************
      *  C100-LIST-RECORDS
      *   Select  Social Security Numbers from LICENCSE_TABLE
      *   and display the list
      ************************************************************************
       C100-LST.

          EXEC SQL DECLARE SSS_CURSOR CURSOR FOR
SELECT SSS FROM LICENSE_TABLE
          END-EXEC.

          EXEC SQL OPEN SSS_CURSOR END-EXEC.

          DISPLAY "Available records:".

          PERFORM C110-DISPLAY-RECORDS UNTIL SQLCODE = 1403.
          EXEC SQL CLOSE SSS_CURSOR END-EXEC.
       C100-EXIT.
          EXIT.
       C110-DISPLAY-RECORDS SECTION.
      ************************************************************************
      *  C110-DISPLAY-RECORDS
      *    Fetch the next record from the cursor and display it.
      ************************************************************************
       C110-DSPLY.
            EXEC SQL FETCH SSS_CURSOR INTO :SSS END-EXEC.
            IF SQLCODE = 0 THEN
                DISPLAY SSS.
       C110-EXIT.
            EXIT.

       C200-GET-RECORD SECTION.
      ************************************************************************
      *  C200-GET-RECORD
```

```
*    Allocates the global clob LICENSE-TXT then selects
*    the name and text which corresponds to the client-supplied
*    sss.  It then calls Z200-PRINTCRIME to print the information and
*    frees the clob.
*****************************************************************
 C200-GTRECRD.
     PERFORM Z100-GET-SSS.
     IF (SSSEXISTS = 1)
       EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
       EXEC SQL SELECT NAME, TXT_SUMMARY
            INTO :NAME1, :LICENSE-TXT FROM LICENSE_TABLE
            WHERE SSS = :SSS END-EXEC
       DISPLAY "================================================
-         "======================="
       DISPLAY " "
       DISPLAY "NAME:  ", NAME1-ARR, "SSS:  ", SSS
       DISPLAY " "
       PERFORM Z200-PRINTCRIME
       DISPLAY " "
       DISPLAY "================================================
-         "======================="
       EXEC SQL FREE :LICENSE-TXT END-EXEC
     ELSE
        DISPLAY "SSS Number Not Found".
 C200-EXIT.
     EXIT.
 C310-GETNEWCRIME SECTION.
*****************************************************************
*  C310-GETNEWCRIME
*    Provides a list of the possible crimes to the user and
*    stores the user's correct response in the variable
*    NEWCRIME.
*****************************************************************
 C310-GTNWCRM.

      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

      DISPLAY " ".
      DISPLAY "Select from the following:".
      PERFORM C311-DISPLAY-CRIME
        VARYING CRIME-INDEX FROM 1 BY 1
        UNTIL CRIME-INDEX > 5.
      MOVE ZERO TO CHOICE.
      PERFORM Z300-ACCEPT-CHOICE UNTIL CHOICE < 6
                           AND CHOICE > 0.
```

```
           MOVE CRIME(CHOICE) TO NEWCRIME-ARR.
           MOVE 35 TO NEWCRIME-LEN.
           MOVE ZERO TO CHOICE.
       C310-EXIT.
           EXIT.
       C311-DISPLAY-CRIME SECTION.
      ******************************************************************
      *   C311-DISPLAY-CRIME
      *      Display an element of the crime table
      ******************************************************************
       C311-DSPLYCRM.
           DISPLAY "(", CRIME-INDEX, ") ", CRIME(CRIME-INDEX).
       C311-EXIT.
           EXIT.
       C320-APPENDTOCLOB SECTION.
      ******************************************************************
      * C320-APPENDTOCLOB
      *    Obtains the length of the global clob LICENSE-TXT and
      *    uses that in the LOB WRITE statement to append the NEWCRIME
      *    character buffer to the global clob LICENSE-TXT.
      *    The name corresponding the global SSS is then selected
      *    and displayed to the screen along with value of LICENSE-TXT.
      *    The caller to this function must allocate, select and later
      *    free the global clob LICENSE-TXT.
      ******************************************************************
       C320-PPNDTCLB.

           EXEC SQL
WHENEVER SQLERROR DO PERFORM Z900-SQLERROR
   END-EXEC.

           EXEC SQL LOB DESCRIBE :LICENSE-TXT GET LENGTH
               INTO :TXT-LEN END-EXEC.

           MOVE NEWCRIME-LEN TO CRIME-LEN.
           IF (TXT-LEN NOT = 0)
             ADD 3 TO TXT-LEN
           ELSE
             ADD 1 TO TXT-LEN.
           EXEC SQL LOB WRITE :CRIME-LEN FROM :NEWCRIME
                 INTO :LICENSE-TXT AT :TXT-LEN END-EXEC.

           EXEC SQL SELECT NAME INTO :NAME1 FROM LICENSE_TABLE
               WHERE SSS = :SSS END-EXEC.
           DISPLAY " ".
```

```
           DISPLAY "NAME:  ", NAME1-ARR, "SSS:  ", SSS.
           DISPLAY " ".
           PERFORM Z200-PRINTCRIME.
           DISPLAY " ".

       C320-EXIT.
          EXIT.

       C300-ADD-CRIME SECTION.
      ******************************************************************
      * ADD-CRIME
      *    Obtains a sss and crime from the user and appends
      *    the crime to the list of crimes of the corresponding sss.
      ******************************************************************
       C300-DDCRM.

           EXEC SQL
              WHENEVER SQLERROR DO PERFORM Z900-SQLERROR
END-EXEC.

           PERFORM Z100-GET-SSS.
           IF (SSSEXISTS = 1)
             EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
             PERFORM C310-GETNEWCRIME
             EXEC SQL SELECT TXT_SUMMARY INTO :LICENSE-TXT
                  FROM LICENSE_TABLE WHERE SSS = :SSS
                  FOR UPDATE END-EXEC
             PERFORM C320-APPENDTOCLOB
             EXEC SQL FREE :LICENSE-TXT END-EXEC
           ELSE
              DISPLAY "SSS Number Not Found".
       C300-EXIT.
           EXIT.

       C400-NEW-RECORD SECTION.
      ******************************************************************
      * C400-NEW-RECORD
      *    Obtains the sss and name of a new record and inserts them
      *    along with an empty_clob() for the clob in the table.
      ******************************************************************
       C400-NWRCRD.

           PERFORM Z100-GET-SSS.
           IF (SSSEXISTS = 1)
             DISPLAY "Record with that sss number already exists"
```

```
     ELSE
       DISPLAY "Name? " WITH NO ADVANCING
       ACCEPT NEWNAME
       DISPLAY " ".
       EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
       EXEC SQL INSERT INTO LICENSE_TABLE
            VALUES (:SSS, :NEWNAME, EMPTY_CLOB()) END-EXEC
       EXEC SQL SELECT TXT_SUMMARY INTO :LICENSE-TXT
            FROM LICENSE_TABLE WHERE SSS = :SSS END-EXEC
       DISPLAY "================================================
-         "======================="
       DISPLAY "NAME: ", NEWNAME,"SSS: ", SSS
       PERFORM Z200-PRINTCRIME
       DISPLAY "================================================
-         "======================="
       EXEC SQL FREE :LICENSE-TXT END-EXEC.
 C400-EXIT.
    EXIT.
 D000-LOGOFF SECTION.
*****************************************************************
*  D000-LOGOFF
*    Commit the work done to the database and log off
*****************************************************************
 D000-LGFF.
     EXEC SQL COMMIT WORK RELEASE END-EXEC.
     DISPLAY " ".
     DISPLAY "HAVE A GOOD DAY!".
     DISPLAY " ".
 D000-EXIT.
     STOP RUN.
 Z100-GET-SSS SECTION.
*****************************************************************
*  Z100-GET-SSS
*    Fills the global variable SSS with the client-supplied sss.
*    Sets the global variable SSSEXISTS to 0 if the sss does not
*    correspond to any entry in the database, else sets it to 1.
*****************************************************************
 Z100-GTSSS.
     DISPLAY "Social Security Number? " WITH NO ADVANCING.
     ACCEPT SSS.
     DISPLAY " ".

     EXEC SQL SELECT COUNT(*) INTO :SSSCOUNT FROM LICENSE_TABLE
         WHERE SSS = :SSS END-EXEC.
```

```
     IF (SSSCOUNT = 0)
        MOVE 0 TO SSSEXISTS
     ELSE
        MOVE 1 TO SSSEXISTS.
 Z100-EXIT.
    EXIT.
 Z200-PRINTCRIME SECTION.
*****************************************************************
*  Z200-PRINTCRIME
*    Obtains the length of the global clob LICENSE-TXT and
*    uses that in the LOB READ statement to read the clob
*    into a character buffer to display the contents of the clob.
*    The caller to this function must allocate, select and later
*    free the global clob LICENSE-TXT.
*****************************************************************
 Z200-PRNTCRM.
     DISPLAY "====================".
     DISPLAY " CRIME SHEET SUMMARY ".
     DISPLAY "====================".

     MOVE SPACE TO THE-STRING-ARR.
     EXEC SQL LOB DESCRIBE :LICENSE-TXT GET LENGTH
         INTO :TXT-LENGTH END-EXEC.

     IF (TXT-LENGTH = 0)
        DISPLAY "Record is clean"
      ELSE
        EXEC SQL LOB READ :TXT-LENGTH FROM :LICENSE-TXT
           INTO :THE-STRING END-EXEC
        DISPLAY THE-STRING-ARR.

 Z200-EXIT.
     EXIT.
 Z300-ACCEPT-CHOICE SECTION.
*****************************************************************
*   Z300-ACCEPT-CHOICE
*      Accept a choice between 1 and 5
*****************************************************************
 Z300-CCPT.
     DISPLAY "Your Selection (1-5)? " WITH NO ADVANCING.
     ACCEPT CHOICE.
     DISPLAY " ".
     IF CHOICE >5 OR CHOICE < 1 THEN
        DISPLAY "Invalid Selection"
        DISPLAY "Please Choose from the indicated list".
```

```
 Z300-EXIT.
    EXIT.

 Z900-SQLERROR SECTION.
*********************************************************************
* Z900-SQLERROR
*    Called whenever a SQLERROR occurs.
*    Display the Error, Roll Back any work done and Log Off
*********************************************************************
 Z900-SQLRRR.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
 Z900-EXIT.
     STOP RUN.
```

14

Precompiler Options

This chapter describes the precompiler options of Pro*COBOL. This chapter includes:

The procob Command

The location of Pro*COBOL differs from system to system. Typically, your system manager or DBA defines environment variables or aliases or uses other operating system-specific means to make the Pro*COBOL executable accessible.

To run the Oracle Pro*COBOL Precompiler, you issue the command

```
procob [option_name=value] [option_name=value] ...
```

The option value is always separated from the option name by an equals sign (=), with no white space around the equals sign.

For example, the INAME option specifies the source file to be precompiled. 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 myfile.cob
```

is equivalent to

```
procob MODE=ANSI INAME=myfile.pco ONAME=myfile.cob
```

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 (as in UNIX for example) you must specify filename values, including the name of Pro*COBOL executable, using the correct combination of upper and lowercase letters.

> **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 or lower 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.

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.

Consult your platform-specific documentation.

Actions 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 generated code also contains the calls to SQLLIB routines that perform the embedded SQL operations.

Pro*COBOL can issue warnings and error messages. These messages are described in *Oracle9i Database Error Messages*.

About the 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.

The *value* of an option is a 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.

For the option

```
... PREFETCH=100
```

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
```

You leave no white space around the equals (=) sign. This is 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 unless the abbreviation is ambiguous. For example, you cannot use the abbreviation MAX because it might stand for MAXLITERAL 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.

If the MAXOPNCURSORS 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 "Entering Precompiler Options" on page 14-6 and "Entering Precompiler Options" on page 14-6.

Macro and Micro Options

Option MODE is known as a *macro* option. 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 the macro option has precedence over micro options. This is the case if, and only if, the macro option is at a higher level of precedence than the micro option. (As described in the section "Precedence of Option Values" on page 14-4.) 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 END_OF_FETCH=1403 in your configuration file and MODE=ANSI on the command line, then 100 will be returned.

The following table lists the values of micro options set by the macro option values:

Table 14–1 How Macro Option Values Set Micro Option Values

Macro Option	Micro Option
MODE=ANSI | ISO	CLOSE_ON_COMMIT=YES
	DECLARE_SECTION=YES
	END_OF_FETCH=100
	DYNAMIC=ANSI
	TYPE_CODE=ANSI

Table 14–1 How Macro Option Values Set Micro Option Values

Macro Option	Micro Option
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
	DYNAMIC=ORACLE
	TYPE_CODE=ORACLE

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=?
```

Entering Precompiler Options

All Pro*COBOL options (except CONFIG) can be entered on the command line or 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 `... [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

Enter options inline by coding EXEC ORACLE OPTION 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, "Performance Tuning" 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. These 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.
    ...
```

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 *$ORACLE_HOME/precomp/admin* directory, where *$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 14-6.

> **Note:** You cannot nest configuration files. This means that CONFIG is not a valid option inside a configuration file.

Scope of Precompiler 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 14–2 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 14–2 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 before the first executable statement.
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.
DYNAMIC	ORACLE	Specifies Oracle or ANSI dynamic semantics in SQL Method 4.
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.

Table 14–2 Option List

Syntax	Default	Specifies
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.
MAXLITERAL=*integer* *	1024	Maximum length of strings.
MAXOPENCURSORS=*integer* *	10	Maximum number of OraCursors cached (1).
MODE={ORACLE | ANSI}	ORACLE	If ANSI, follow the ANSI/ISO SQL standard.
NESTED={YES | NO}	YES	If YES, nested programs are supported.
NLS_LOCAL={YES | NO}	NO	If YES, use NCHAR semantics of previous Pro*COBOL releases.
[ONAME=]*filename*	*iname*.cob	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 for each page in listing.
PICX	CHARF	Datatype of PIC X COBOL variables.
PREFETCH	1	Speed up queries by pre-fetching a given number of rows.

Table 14–2 Option List

Syntax	Default	Specifies
RELEASE_CURSOR={YES | NO} *	NO	If YES, release OraCursor after execute.
SELECT_ERROR={YES | NO}*	YES	If YES, generate FOUND error on SELECT.
SQLCHECK={SEMANTICS | SYNTAX}*	SYNTAX	SQL checking level.
THREADS={YES | NO}	NO	Indicates a multithreaded application.
TYPE_CODE	ORACLE	Use Oracle or ANSI type codes for dynamic SQL method 4.
UNSAFE_NULL={YES | NO}	NO	If YES, unsafe null fetches are allowed (disables the ORA-01405 message).
USERID=*username/password[@dbname]*		Oracle username, password, and optional database.
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 Precompiler Options

This section is organized for easy reference. It lists the Pro*COBOL precompiler 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=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 14-5.

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 "Entering Precompiler Options" on page 14-6.

You cannot nest configuration files. Therefore, you cannot specify the option CONFIG 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 14–3 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
installation-defined	LOCAL	Any installation-defined form.

'*fmt*' is a date format model, such as "Month dd, yyyy". See Oracle9i 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*i*, Oracle9*i*, or the native version of Oracle (that is, the version to which your application is connected).

Syntax

DBMS={V7 | V8 | V9 | NATIVE}

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.

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.

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 a discussion of precedence of this option, see "Precedence of Option Values" on page 14-4.

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 2-25.

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.
```

DYNAMIC

Purpose

This micro option of MODE specifies the descriptor behavior in dynamic SQL Method 4.

Syntax

DYNAMIC={ORACLE | ANSI}

Default

ORACLE

Usage Notes

Cannot be entered inline by use of the EXEC ORACLE OPTION statement.

See the DYNAMIC option settings in "ANSI Dynamic SQL Precompiler Options" on page 10-12.

END_OF_FETCH

Purpose

This micro option of MODE 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.

END_OF_FETCH is a micro option of MODE. For further discussion, see "Macro and Micro Options" on page 14-5.

If you specify MODE=ANSI in a configuration file, Pro*COBOL returns the SQLCODE 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, which has a higher precedence than your configuration file, Pro*COBOL will again return the SQLCODE value 1403 at the END_OF_FETCH.

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 VARCHAR 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 statements.

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, or your COBOL compiler.

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 can start in column 1. Example code in this book is in TERMINAL format. See "Coding Areas" on page 2-12 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, "Performance Tuning".

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, 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 and 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.

For information showing how the HOLD_CURSOR and RELEASE_CURSOR options interact, see Appendix D, "Performance Tuning", specifically Table D–1, "HOLD_CURSOR and RELEASE _CURSOR Interactions" on page D-13.

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.

All input file names must be unique at precompilation time.

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*.

You need not use a file extension when specifying INAME unless the extension is nonstandard. On the UNIX platform, Pro*COBOL assumes the default input file extension *pco*.

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.

Note: Pro*COBOL looks for a file in the current directory first—even if you specify a directory path. Therefore, 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 nondefault 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 132. If you specify a value below the range, 80 is used instead. If you specify a value above the range, an error occurs. 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 14–4 Types of Listings

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 1024.

Usage Notes

The maximum value of MAXLITERAL is compiler-dependent. The default value is language-dependent, but you may 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, "Performance Tuning".

When precompiling separately, use MAXOPENCURSORS as described in "Separate Precompilations" on page 2-26.

MAXOPENCURSORS specifies the *initial* size of the SQLLIB cursor cache.

When an implicit statement is executed and HOLD_CURSOR=NO, or an explicit cursor is closed, the cursor entry is marked as reusable. If this statement is issued again and the cursor entry has not been used for another statement, it is reused.

If a new cursor is needed and the number of cursors allocated is less than MAXOPENCURSORS, then the next one in the cache is allocated. Once MAXOPENCCURSORS has been exceeded, Oracle first tries to reuse a previous entry. If there are no free entries, then an additional cache entry is allocated. Oracle continues to do this until the program runs out of memory or the database parameter OPEN_CURSORS is exceeded.

During normal processing, when using HOLD_CURSOR=NO and RELEASE_CURSOR=NO (the default), it is advisable to set MAXOPENCURSORS to no more than 6 less than the database parameter OPEN_CURSORS to allow for the cursors used by the data dictionary to process statements.

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

This macro option 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 8-2.

NESTED

Purpose

Indicates whether GLOBAL clauses in nested programs are to be generated. If the compiler supports nested programs, use YES as the value of NESTED.

Syntax

NESTED={YES | NO}

Default

YES

Usage Notes

Cannot be entered inline.

NLS_LOCAL

Purpose

The NLS_LOCAL option determines whether Globalization Support (formerly called 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.

This option is for use when passing National Character Set variables to and from the server.

When NLS_LOCAL=YES, the runtime library (SQLLIB) locally performs blank-padding and blank-stripping for host variables that have multibyte

Globalization Support datatypes. Continue to use this value only for Pro*COBOL applications written for releases before releases 8.0.

When NLS_LOCAL=YES, because dynamic SQL statements are not processed at precompile time, this option has no effect on dynamic SQL statements.

Also, when NLS_LOCAL=YES, columns storing multibyte Globalization Support 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.

When NLS_LOCAL=NO, blank-padding and blank-stripping operations are performed by the Oracle Server for host variables that have multibyte Globalization Support datatypes. Use for all new release 8.0, or later, applications.

The environment variable NLS_NCHAR specifies the character set used for National Character Set data. (NCHAR, NVARCHAR2, NCLOB). If it is not specified, the character set defined or defined indirectly by NLS_LANG will be used. See: the NLS_LANG section in the *Oracle9i Globalization Support Guide* for details.

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 IRECLEN. The maximum value allowed is system-dependent.

PAGELEN

Purpose

Specifies the number of lines for each 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 was 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. See "Default for PIC X" on page 4-31 for an extensive discussion.

PREFETCH

Purpose

Use this option to speed up queries by pre-fetching a given number of rows.

Syntax

PREFETCH=*integer*

Default

1

Usage Notes

Can be used in a configuration file or on the command-line. The value of the integer is used for execution of all queries using explicit cursors, subject to the rules of precedence.

When used in-line it must be placed before OPEN statements with explicit cursors. Then the number of rows pre-fetched when that OPEN is done is determined by the last in-line PREFETCH option in effect.

The PREFETCH default is 1. To turn off prefetching, use PREFETCH=0 on the command line.

Prefetching is turned off when LONG or LOB columns are being accessed. PREFETCH is used to enhance the performance of single row fetches. PREFETCH values have no effect when doing array fetches, regardless of which value is assigned.

There is no single *perfect* prefetch number that can be used to assist all the fetches in an application.

Therefore, when using the PREFETCH option, you should test different values to give a general improvement across all statements in the program. Note that if certain statements need to be tuned individually, the PREFETCH option can be specified in line using EXEC ORACLE OPTION. Note that this will affect *all* fetch statements that follow the command in your program. Select the appropriate prefetch number to enhance the performance of any particular FETCH statement. To achieve this individual prefetch count, you should use the inline prefetch option. (Rather than from the command line.)

The maximum value is 9999. See "The PREFETCH Precompiler Option" on page 5-18 for further discussion.

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, "Performance Tuning".

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, 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. For information showing how these two options interact, see Appendix D, "Performance Tuning", specifically Table D–1, "HOLD_CURSOR and RELEASE _ CURSOR Interactions" on page D-13.

SELECT_ERROR

Purpose

Specifies whether your program generates an error when a 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}

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.

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

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 Oracle9*i* syntax rules, which are downwardly compatible. Specify SQLCHECK=SYNTAX when migrating your precompiled programs.

Table 14–5 summarizes the checking done by SQLCHECK. For more information about syntactic and semantic checking, see Appendix E, "Syntactic and Semantic Checking".

Table 14–5 Checking Done by SQLCHECK

	SQLCHECK=SEMANTICS		**SQLCHECK=SYNTAX**	
	Syntax	Semantics	Syntax	Semantics
DML	**X**	X	X	
Remote DML	X		X	
PL/SQL	X	X		

THREADS

Purpose

When THREADS=YES, the precompiler allows multithreaded applications.

Syntax

THREADS={YES | NO}

Default

NO

Usage Notes

Cannot be entered inline.

This precompiler option is required for any program that requires multithreading support.

With THREADS=YES, the precompiler generates an error if no EXEC SQL CONTEXT USE directive is encountered before the first context is visible and an executable SQL statement is found. For more information, see Chapter 12, "Multithreaded Applications".

TYPE_CODE

Purpose

This micro option of MODE specifies whether ANSI or Oracle datatype codes are used in ANSI dynamic SQL method 4. Its setting is the same as the setting of MODE option.

Syntax

TYPE_CODE={ORACLE | ANSI}

Default

ORACLE

Usage Notes

Cannot be entered inline.

See the possible option settings inTable 10–3 on page 10-13.

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.

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*[@*dbname*]

Default

None

Usage Notes

Cannot be entered inline.

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. The database alias is optional. Do not enter the brackets.

VARCHAR

Purpose

The VARCHAR option instructs Pro*COBOL to treat the COBOL group item described in Chapter 5, "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, "Embedded SQL" is accepted as a 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.

A

New Features

This appendix looks at the improvements and new features offered by the Oracle Pro*COBOL Precompiler. Each description includes a reference to more complete descriptions in the chapters is provided.

These topics are presented:

New Features of Release 9.0.1

The new features and terminology in release 9.0.1 include:

Globalization Support

National Language Support (NLS) was re-named Globalization Support.

New Datetime Datatypes

Pro*COBOL supports five new datetime datatypes: `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`, and `TIMESTAMP WITH LOCAL TIMEZONE`. You can select from columns of these datatypes onto the `OCIInterval` and `OCIDateTime` host variables and into objects where attributes are datetime types. For details, see "Datetime and Interval Datatype Descriptors" on page 4-13.

New Features of Release 8.1

Multithreading Applications Supported

Multithreaded COBOL applications are supported. A multithreaded Pro* C/C++ program can call a Pro*COBOL subprogram. See Chapter 12, "Multithreaded Applications".

CALL Statement

The CALL embedded SQL statement invokes a stored procedure. It can be used instead of an embedded PL/SQL block or a stored Java procedure in new applications. See "CALL (Executable Embedded SQL)" on page F-13.

Calling Java Methods

Stored procedures (methods) written in Java can be called from your application. For information on how to call a procedure written in Java, see "Stored PL/SQL and Java Subprograms" on page 6-21.

LOB Support

An embedded SQL statement interface enables LOBs (large objects) to be used in precompiler applications. How LOBs are used, the internal and external LOBs, and

comparisons with other ways to handle LOBs are presented. A presentation of each new SQL statement is made. Sample code shows how to use the LOB interface. See the chapter Chapter 13, "Large Objects (LOBs)" for complete details.

ANSI Dynamic SQL

The complete ANSI implementation of dynamic SQL Method 4 through embedded SQL statements is presented in Chapter 10, "ANSI Dynamic SQL". An overview with simple examples is presented. This is followed by a complete discussion of the new SQL statements. Sample programs from the `demo` directory are then shown.

PREFETCH Option

This precompiler option speeds up database access by *prefetching* values, thus cutting down the number of network round-trips. See "The PREFETCH Precompiler Option" on page 5-18.

DML Returning Clause

This clause, which enables you to save round-trips to the database server, is now allowed in INSERT, DELETE, and UPDATE statements. See "Inserting Rows" on page 5-9.

Universal ROWIDs

The support for universal ROWID datatype is presented. Index-organized tables use this concept. See Universal ROWIDs on page 4-34.

SYSDBA/SYSOPER Privileges in CONNECT Statements

To set these privileges using the CONNECT statement, see "Connecting to Oracle" on page 3-2.

Tables of Group Items

Tables of group items are now allowed as host variables in Pro*COBOL. See "Tables of Group Items as Host Variables" on page 7-20.

WHENEVER DO CALL Branch

The WHENEVER directive now has a DO CALL action: a subprogram is called. See "WHENEVER Directive" on page 8-15.

DECIMAL-POINT IS COMMA

The DECIMAL-POINT IS COMMA clause is supported. This permits commas to be used instead of decimal points in numeric literals. See "Decimal-Point is Comma" on page 2-14.

Optional Division Headers

The following divisions and their contents are now optional: IDENTIFICATION, ENVIRONMENT, DATA. See "Division Headers that are Optional" on page 2-15.

NESTED Option

When set to NO, the NESTED precompiler option will prevent generation of the GLOBAL clause for non-nested programs. See "NESTED" on page 14-31.

DB2 Compatibility Features of Release 8.0

These new features in Pro*COBOL release 8.0 help you migrate applications from DB2 to Oracle, 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. For more details, see "DECLARE_SECTION" on page 14-17.

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 for each byte of computer storage. The sign is held in the high order nibble of one of the bytes. It is called over-punch 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 over-punch.
- Display-1 Multibyte type (PIC G). This datatype is equivalent to PIC N and is used for multibyte characters.

See "Host Variables" on page 4-15.

Support of Group Items as Host Variables

Pro*COBOL now enables 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 4-23.

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.

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 4-30.

Explicit Control Over the END-OF-FETCH SQLCODE Returned

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 precompiler option must be used on the command line or in a configuration file. For more details see "END_OF_FETCH" on page 14-19.

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 5-12.

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 CLOSE_ON_COMMIT setting and the existing behavior which is associated with MODE option. For more details, see "Declaring a Cursor" on page 5-12 and "CLOSE_ON_COMMIT" on page 14-14.

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 8-14.

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 "mm, dd, yyyy". See the Oracle9i SQL Reference for the list of date format model elements. For more details, see "DATE_ FORMAT" on page 14-16.

Any Terminator Allowed After SQL Statements

A SQL statement now can be terminated by a comma, a period or another COBOL statement. For more details, see "Sentence Terminator" on page 2-19.

Other New Features of Release 8.0

New Name for Configuration File

The configuration file is now called *pcbcfg.cfg*, instead of *pccob.cfg*. See "Entering Precompiler Options" on page 14-6.

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 for each 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.

See "Host Variables" on page 4-15.

Support of Nested Programs

Pro*COBOL now enables 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 2-22.

Support for REDEFINES and FILLER

The REDEFINES clause can be used to redefine group items. For more details, see "REDEFINES Clause" on page 2-18

The word FILLER is now allowed in host variable declarations. For more details, see "Host Variables" on page 4-15

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 14-34.

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 4-47.

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 enables you to change the password:

```
EXEC SQL CONNECT ... [ALTER AUTHORIZATION :new_password] END-EXEC.
```

See "Changing Passwords at Runtime" on page 3-10 and "CONNECT (Executable Embedded SQL Extension)" on page F-17.

Error Message Codes

Error and warning codes are different between earlier releases of Pro*COBOL and the current release. See *Oracle9i Database Error Messages* for a complete list of codes and messages.

The runtime messages issued by SQLLIB now have the prefix SQL-, rather than the RTL- prefix used in earlier Pro*COBOL releases. The message codes remain the same as those of earlier releases.

When precompiling with SQLCHECK=SEMANTICS, PLS is the prefix used by the PL/SQL compiler. Such errors are not from Pro*COBOL.

Migration From Earlier Releases

Existing applications written in Pro*COBOL will work unchanged with an Oracle9*i* server. If you precompile again, you may have to change the settings of the precompiler options. See Chapter 14, "Precompiler Options".

See Also: *Oracle9i Database Migration* for additional details.

B

Operating System Dependencies

Some details of 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

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 4–6, "Host Variable Declarations" on page 4-16 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. They 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.

Due to a Pro*COBOL limitation, when interacting with SQLLIB (C routines), some unpredictable results may occur unless boundaries for host variables are properly aligned. Refer to your COBOL documentation for specific information on defining host variable boundary alignment. Work-arounds could include:

- Manual alignment using FILLER
- FORCE the boundary by using 01 level entries
- If the data source is third party code, then use temporary variables at 77 level entries or 01 level entries, and use those as host variables.

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 1024, 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 14, "Precompiler Options"

PIC N or Pic G Clause for Multi-byte Globalization Support Characters

Some COBOL compilers may not support the use of the PIC N or PIC G clause for declaring multibyte Globalization Support character variables. Check your COBOL user's guide before writing source code that uses these clauses to declare multibyte character variables.

RETURN-CODE Special Register May Be Unpredictable.

The contents of the RETURN-CODE special register (for those systems that support it) are unpredictable after any SQL statement or SQLLIB function.

Byte-Order of Binary Data

On some platforms such as NT, the COBOL compiler reverses the byte-ordering of binary data. See your platform-specific documentation for the COMP5 precompiler option.

C

Reserved Words, Keywords, and Namespaces

Topics in this appendix include:

Reserved Words and Keywords

Some words are reserved by Oracle. That is, they have a special meaning to Oracle and cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes. To view the lists of the Oracle reserved words for SQL and PL/SQL, see the *Oracle9i SQL Reference* and the *PL/SQL User's Guide and Reference*.

Like COBOL keywords, you cannot use Pro*COBOL keywords as variables in your program(s). Otherwise, an error will be generated. An error may result if they are used as the name of a database object such as a column. Here are the keywords used in Pro*COBOL:

all	allocate	alter
analyze	and	any
append	arraylen	as
asc	assign	at
audit	authorization	avg
begin	between	bind
both	break	buffer
buffering	by	call
cast	char	character
character_set_name	charf	charz
check	chunksize	close
comment	commit	connect
constraint	constraints	context
continue	convbufsz	copy
count	create	current
currval	cursor	data
database	date	datetime_interval_code
datetime_interval_precision	day	deallocate
decimal	declare	default
define	delete	desc

describe
disable
do
enable
endif
exec
explain
file
first
for
free
get
goto
having
hour
ifdef
in
indicator_stride_length
integer
interval
isopen
leading
like
lob
long
min
mode
national_character
nextval
descriptor
display
drop
end
erase
execute
extract
fileexists
float
force
from
global
grant
hold
iaf
ifndef
include
input
internal_length
into
istemporary
length
list
local
max
minus
month
nchar
noaudit
directory
distinct
else
end-exec
escape
exists
fetch
filename
flush
found
function
go
group
host_stride_length
identified
immediate
indicator
insert
intersect
is
last
level
load
lock
message
minute
name
next
not

notfound
nowait
null
nullable
number
nvarchar2
octet_length
of
one
only
open
option
or
oracle
order
output
overlaps
overpunch
package
partition
perform
precision
prepare
prior
procedure
put
raw
read
ref
reference
release
rename
replace
return
returned_length
returned_octet_length
returning
revoke
role
rollback
rowid
rownum
savepoint
scale
second
section
select
set
some
sql
sql-context
sql-cursor
sqlerror
sqlwarning
start
statement
stddev
stop
string
sum
sysdate
sysdba
sysoper
table
temporary
threads
time
timestamp
timezone_hour
timezone_minute
to
tools
trailing
transaction
trigger
trim
truncate
type
uid
union
unique
unsigned
user_defined_type_name
user_defined_type_name_length
user_defined_type_schema
user_defined_type_schema_length
user_defined_type_version

update	use	user
using	validate	value
values	var	varchar
varchar2	variables	variance
varnum	varraw	view
whenever	where	with
work	write	year
zone		

Reserved Namespaces

Table C–1 contains a list of namespaces that are reserved by Oracle. The initial characters of subprogram names in Oracle libraries are restricted to the character strings in this list. Because of potential name conflicts, use subprogram names that do not begin with these characters.

For example, the Oracle Net Transparent Network Service functions all begin with the characters "NS," so avoid writing subprograms with names begin with "NS."

Table C–1 Reserved Namespaces

Namespace	Library
XA	external functions for XA applications only
SQ	external SQLLIB functions used by Oracle Precompiler and SQL*Module applications
O, OCI	external OCI functions internal OCI functions
UPI, KP	function names from the Oracle UPI layer

Table C–1 Reserved Namespaces

Namespace	Library
NA	Oracle Net Native services product
NC	Oracle Net RPC project
ND	Oracle Net Directory
NL	Oracle Net Network Library layer
NM	Oracle Net Net Management Project
NR	Oracle Net Interchange
NS	Oracle Net Transparent Network Service
NT	Oracle Net Drivers
NZ	Oracle Net Security Service
OSN	Oracle Net V1
TTC	Oracle Net Two task
GEN, L, ORA	Core library functions
LI, LM, LX	function names from the Oracle Globalization Support layer
S	function names from system-dependent libraries

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 percent or more. Topics are:

Causes of Poor Performance

One cause of poor performance is high Oracle communication overhead. Oracle must process SQL statements one at a time. Thus, each statement results in another call to Oracle and higher overhead. In a networked environment, SQL statements must be sent over the network, adding to network traffic. Heavy network traffic can slow down your application significantly.

Another cause of poor performance is inefficient SQL statements. Because SQL is so flexible, you can get the same result using two different statements. Using 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.
```

Contrasted with:

```
EXEC SQL SELECT DNAME, DEPTNO
    FROM DEPT
    WHERE EXISTS
    (SELECT DEPTNO FROM EMP WHERE DEPT.DEPTNO = EMP.DEPTNO)
END-EXEC.
```

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.

Another cause of poor performance is unnecessary parsing and binding. Recall that before executing a SQL statement, Oracle must parse and bind it. Parsing means examining the SQL statement to make sure it follows syntax rules and refers to valid database objects. Binding means associating host variables in the SQL statement with their addresses so that Oracle can read or write their values.

Many applications manage cursors poorly. This results in unnecessary parsing and binding, which adds noticeably to processing overhead.

Improving Performance

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 tables
- 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
- Avoiding unnecessary reparsing

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, Oracle executes the INSERT statement once, inserting a single row into the EMP table. In that row, the SAL column has the value of *SALARY*. To insert 300 rows this way, you must execute the INSERT statement 300 times.

However, if *SALARY* is a host table of size 300, Oracle inserts all 300 rows into the EMP table at once. In each row, the SAL column has the value of an element in the *SALARY* table.

For more information, see Chapter 7, "Host Tables"

Using PL/SQL and Java

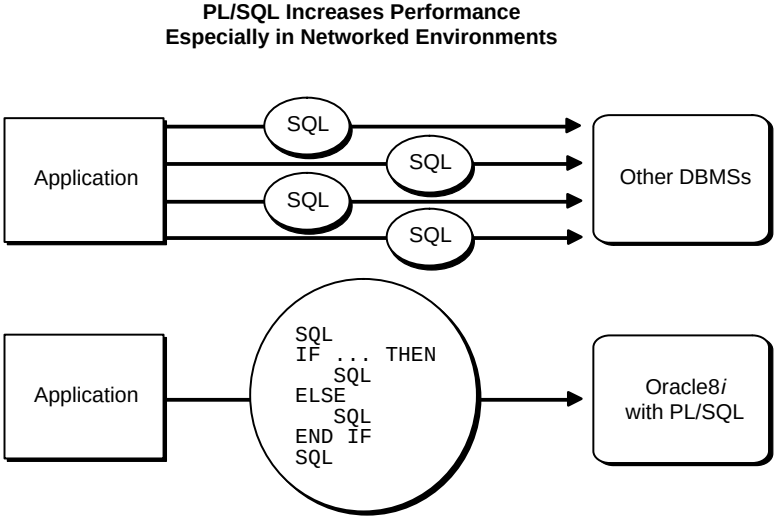
As Figure E-1 shows, if your application is database-intensive, you can use control structures to group SQL statements in a PL/SQL block, then send the entire block to

Oracle. This can drastically reduce communication between your application and the database.

Also, you can use PL/SQL and Java subprograms to reduce calls from your application to the database. 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 and Java subprograms can be compiled separately and stored in a 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



Optimizing SQL Statements

For every SQL statement, the optimizer generates an *execution plan*, which is a series of steps that Oracle takes to execute the statement. These steps are determined by rules given in the *Oracle9i Application Developer's Guide - Fundamentals*. Following these rules will help you write optimal SQL statements.

Optimizer Hints

For every SQL statement, the optimizer generates an *execution plan*, which is a series of steps that Oracle takes to execute the statement. In some cases, you can suggest 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 *Oracle9i Application Developer's Guide - Fundamentals*

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 Oracle. From these statistics, you can determine which statements take the most time to process. You can then 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.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for instructions on using trace tools and analyzing their output.

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 *Oracle9i 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 *Oracle9i Application Developer's Guide - Fundamentals*.

Taking Advantage of Row-Level Locking

By default, Oracle locks data at the row level rather than the table level. Row-level locking allows multiple users to access different rows in the same table concurrently. The resulting performance gain is significant.

You can specify table-level locking, but it lessens the effectiveness of the transaction processing option. For more information about table locking, see "Using the LOCK TABLE Statement" on "Using the LOCK TABLE Statement" on page 3-22.

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 8-23.

Handling Explicit Cursors

Recall that there are two types of cursors: implicit and explicit (see "Errors and Warnings" on page 2-9). Oracle implicitly declares a cursor for all data definition and data manipulation statements. However, for queries that return more than one row, you should explicitly declare a cursor and fetch in batches rather than select into a host table. 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.

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.

> **Note:** To make performance tuning easier, the precompiler lets you reopen an already open cursor. However, this is an Oracle extension to the ANSI/ISO embedded SQL standard. So, when MODE=ANSI, you must close a cursor before reopening it.

Cursor Control

In general, there are three factors that affect the control of an explicitly declared cursor:

- Using the DECLARE, OPEN, FETCH, and CLOSE statements.
- Using the PREPARE, DECLARE, OPEN, FETCH, 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, which is used in 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, 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 MAXOPENCURSORS options give you some control over how Oracle manages the parsing and re-parsing of SQL statements. Declaring an explicit cursor gives you maximum control over parsing.

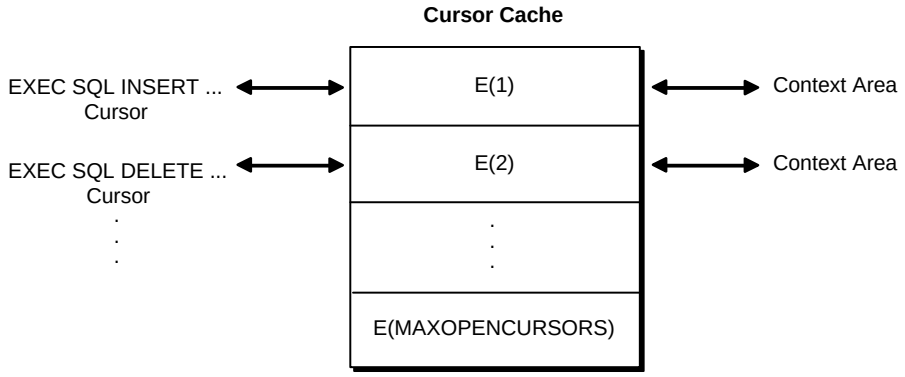
Private SQL Areas and Cursor Cache

When any statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache is a continuously updated area of memory used for cursor management. The cursor cache entry is in turn linked to a private SQL area.

The private SQL area, a work area created dynamically at run time by Oracle, contains the parsed SQL statement, the addresses of host variables, and other information needed to process the statement. Dynamic Method 3 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 for each user session is set by the initialization 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, 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.

If the value of MAXOPENCURSORS is less than the number of statements that need to be cached during the execution of the program, Oracle will search for cursor cache entries to reuse once MAXOPENCURSORS cache entries have been exhausted. 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 reexecute the INSERT statement, Oracle would have to re-parse it and reassign another cache entry.

Oracle allocates an additional cache entry if it cannot find one to reuse. For example, if MAXOPENCURSORS=8 and all eight entries are active, a ninth is created. If necessary, Oracle keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN_CURSORS. This dynamic allocation adds to processing overhead.

Thus, specifying a low value for MAXOPENCURSORS with HOLD_CURSOR=NO (the default) 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 Oracle 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 Oracle executes the SQL statement and the cursor is closed, the private SQL area is automatically freed and the parsed statement lost. This might be necessary if, for example, you wish to conserve memory.

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 tried to specify HOLD_CURSOR=YES, Oracle must still reallocate memory for a private SQL area and re-parse the SQL statement before executing it. Therefore, specifying RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES.

Frequent Execution

The links between a *frequently* executed SQL statement and its private SQL area should be maintained, because the private SQL area contains all the information needed to execute the statement. Maintaining access to this information makes subsequent execution of the statement much faster.

When HOLD_CURSOR=YES, the link between the cursor and cursor cache is maintained after Oracle executes the SQL statement. Thus, the parsed statement and allocated memory remain available. This is useful for SQL statements that you want to keep active because it avoids unnecessary re-parsing.

Effect on the Shared SQL Area

Oracle9*i* caches the parsed representations of SQL statements and PL/SQL in its Shared SQL Cache. These representations are maintained until aged out by the need for the space to be used for other statements. For more information, see the *Oracle9i Database Concepts* manual. The behavior of the Oracle server in this respect is

unaffected by the Precompiler's cursor management settings and so can have the following effects:

- When RELEASE_CURSOR=YES and a statement is re executed, a request will be sent to the server to parse the statement but a full parse may not be necessary since the statement may still be cached.
- When using HOLD_CURSOR=YES no locks are held on any objects referred to in the statement and so a redefinition of one of the objects in the statement will force the cached statement to become invalid and for the server to automatically reparse the statement. This may cause unexpected results.
- Nonetheless, when RELEASE_CURSOR=YES, the re-parse might not require extra processing because Oracle caches the parsed representations of SQL statements and PL/SQL blocks in its *Shared SQL Cache*. Even if its cursor is closed, the parsed representation remains available until it is aged out of the cache.

Embedded PL/SQL Considerations

For the purposes of cursor management, an embedded PL/SQL block is treated just like a SQL statement. When an embedded PL/SQL block is executed, a parent cursor is associated with the entire block and a link is created between the cache entry and the private SQL area in the PGA for the embedded PL/SQL block. Be aware that each SQL statement inside the embedded block also requires a private SQL area in the PGA. These SQL statements use child cursors that PL/SQL manages itself. The disposition of the child cursors is determined through its associated parent cursor. That is, the private SQL areas used by the child cursors are freed after the private SQL area for its parent cursor is freed.

Note:

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 Oracle9*i*, 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

Avoiding Unnecessary Reparsing

When an embedded SQL statement is executed in a loop, it gets parsed only once. However, the execute phase of the SQL statement can result in errors, and statements are reparsed, with the following exceptions:

- ORA-1403 (not found)
- ORA-1405 (truncation)
- ORA-1406 (null value)

By correcting the errors, you can eliminate this unnecessary reparsing.

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:

Syntactic and Semantic Checking Basics

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. It also applies to procedures and functions called from PL/SQL blocks. 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 *Oracle9i 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, or both syntactic and semantic. 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

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.

The precompiler gets the information for a semantic check from embedded DECLARE TABLE statements or, if you specify the option USERID, by connecting to the database and accessing the data dictionary.

If you connect to the database but some table 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 Oracle9*i* set of syntax rules found in the *Oracle9i 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 the database that maintains definitions of tables and views referenced in your host program. After connecting, 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 the database, specify the option USERID on the command line, using the syntax

```
USERID=username/password
```

where *username* and *password* comprise a valid Oracle9*i* 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 the database automatically with the userid

```
<prefix><username>
```

where *prefix* is the value of the initialization parameter OS_AUTHENT_PREFIX (the default value is OPS$) and *username* is your operating system user or task name.

If you try connecting, 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 the database as long as your program does not call any stored procedures or functions from an anonymous PL/SQL block. To do the check, the precompiler must get information about tables and views from embedded DECLARE TABLE directives. 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-29.

If you use DECLARE TABLE to define a database table that already exists, the precompiler uses your definition, ignoring the one in the data dictionary.

F

Embedded SQL Statements and Precompiler Directives

This appendix describes of both SQL92 embedded SQL statements and directives as well as the Oracle9*i* embedded SQL extensions. These statements 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 *Oracle9i SQL Reference*.

This appendix contains the following sections:

- 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 DESCRIPTOR (Executable Embedded SQL)
- ENABLE THREADS (Executable Embedded SQL Extension)
- EXECUTE ... END-EXEC (Executable Embedded SQL Extension)
- EXECUTE (Executable Embedded SQL)
- EXECUTE DESCRIPTOR (Executable Embedded SQL
- EXECUTE IMMEDIATE (Executable Embedded SQL)
- FETCH (Executable Embedded SQL)
- FETCH DESCRIPTOR (Executable Embedded SQL)
- FREE (Executable Embedded SQL Extension)
- GET DESCRIPTOR (Executable Embedded SQL)
- INSERT (Executable Embedded SQL)
- LOB APPEND (Executable Embedded SQL Extension)
- LOB ASSIGN (Executable Embedded SQL Extension)
- LOB CLOSE (Executable Embedded SQL Extension)
- LOB COPY (Executable Embedded SQL Extension)
- LOB CREATE TEMPORARY (Executable Embedded SQL Extension)
- LOB DESCRIBE (Executable Embedded SQL Extension)
- LOB DISABLE BUFFERING (Executable Embedded SQL Extension)
- LOB ENABLE BUFFERING (Executable Embedded SQL Extension)
- LOB ERASE (Executable Embedded SQL Extension)
- LOB FILE CLOSE ALL (Executable Embedded SQL Extension)
- LOB FILE SET (Executable Embedded SQL Extension)
- LOB FLUSH BUFFER (Executable Embedded SQL Extension)

- LOB FREE TEMPORARY (Executable Embedded SQL Extension)
- LOB LOAD (Executable Embedded SQL Extension)
- LOB OPEN (Executable Embedded SQL Extension)
- LOB READ (Executable Embedded SQL Extension)
- LOB TRIM (Executable Embedded SQL Extension)
- LOB WRITE (Executable Embedded SQL Extension)
- OPEN (Executable Embedded SQL)
- OPEN DESCRIPTOR (Executable Embedded SQL)
- PREPARE (Executable Embedded SQL)
- ROLLBACK (Executable Embedded SQL)
- SAVEPOINT (Executable Embedded SQL)
- SET DESCRIPTOR (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 Statements

Embedded SQL statements 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 statements 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 Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
ALLOCATE	O / E	To allocate memory for a cursor variable, LOB locator or ROWID.
ALLOCATE DESCRIPTOR	S / E	To allocate a descriptor for ANSI dynamic SQL.
CALL	S / E	Call a stored procedure.
CLOSE	S / E	To disable a cursor.
COMMIT	S / E	To make all database changes permanent.
CONNECT	O / E	To log on to a database instance.
CONTEXT ALLOCATE	O / E	To allocate memory for a SQLLIB runtime context.
CONTEXT FREE	O / E	To free memory for a SQLLIB runtime context.
CONTEXT USE	O / E	To specify a SQLLIB runtime context.
DEALLOCATE DESCRIPTOR	S / E	To deallocate a descriptor area to free memory.
DECLARE CURSOR	S / D	To declare a cursor, associating it with a query.
DECLARE DATABASE	O / D	To declare an identifier for a nondefault 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.

Table F–1 Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
DESCRIBE DECRIPTOR	S/E	To obtain information about an ANSI SQL statement, and store it in a descriptor.
ENABLE THREADS	O/E	To initialize a process that supports multiple threads.
EXECUTE...END-EXEC	O/E	To execute an anonymous PL/SQL block.
EXECUTE	S/E	To execute a prepared dynamic SQL statement.
EXECUTE DESCRIPTOR	S/E	To execute a prepared statement using ANSI Dynamic SQL.
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.
FETCH DESCRIPTOR	S/E	To retrieve rows selected by a query using ANSI Dynamic SQL.
FREE	S/E	To free memory used by a cursor, LOB locator, or ROWID.
GET DESCRIPTOR	S/E	To move information from an ANSI SQL descriptor area into host variables.
INSERT	S/E	To add rows to a table or to a view's base table.
LOB APPEND	O/E	To append a LOB to the end of another lOB.
LOB ASSIGN	O/E	To assign a LOB or BFILE locator to another locator.
LOB CLOSE	O/E	To close an open LOB or BFILE.
LOB COPY	O/E	To copy all or part of a LOB value into another LOB.
LOB CREATE TEMPORARY	O/E	To create a temporary LOB.
LOB DESCRIBE	O/E	To retrieve attributes from a LOB.
LOB DISABLE BUFFERING	O/E	To disable LOB buffering.
LOB ENABLE BUFFERING	O/E	To enable LOB buffering.
LOB ERASE	O/E	To erase a given amount of LOB data starting from a given offset.
LOB FILE CLOSE ALL	O/E	To close all open BFILE.
LOB FILE SET	O/E	To set DIRECTORY and FILENAME in a BFILE locator.
LOB FLUSH BUFFER	O/E	To write the LOB buffers to the database server.
LOB FREE TEMPORARY	O/E	To free temporary space for the LOB locator.

Table F–1 Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
LOB LOAD	O/E	To copy all or part of a BFILE into an internal LOB.
LOB OPEN	O/E	To open a LOB or BFILE to read or read/write access.
LOB READ	O/E	To read all or part of a LOB or BFILE into a buffer.
LOB TRIM	O/E	To truncate a lob value.
LOB WRITE	O/E	To write the contents of a buffer to a LOB.
OPEN	S/E	To execute the query associated with a cursor.
OPEN DESCRIPTOR	S/E	To execute the query associated with a cursor in ANSI Dynamic SQL.
PREPARE	S/E	To parse a dynamic SQL statement.
ROLLBACK	S/E	To end the current transaction and discard all changes.
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.
SET DESCRIPTOR	S/E	To set information in the ANSI SQL descriptor area from 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 Oracle9*i* external datatype to a host variable.
WHENEVER	S/D	To specify handling for error and warning conditions.

About the Statement Descriptions

The directives, and statements appear alphabetically. The description of each contains the following sections:

Purpose	Describes the basic uses of the statement.
Prerequisites	Lists privileges you must have and steps that you must take before using the statement. Unless otherwise noted, most statements also require that the database be open by your instance.
Syntax	Shows the syntax diagram with the keywords and parameters of the statement.
Keywords and Parameters	Describes the purpose of each keyword and parameter.
Usage Notes	Discusses how and when to use the statement.
Examples	Shows example statements of the statement.
Related Topics	Lists related statements, clauses, and sections of this manual.

How to Read Syntax Diagrams

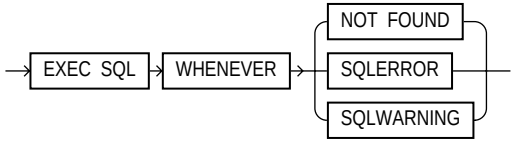
Syntax diagrams are used to illustrate embedded SQL syntax. They are drawings that depict valid syntax.

Trace each diagram from left to right, in the direction shown by the arrows.

Statements 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 ...
```

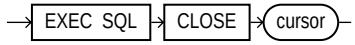
```
EXEC SQL WHENEVER SQLWARNING ...
```

Statement Terminator

In all Pro*COBOL EXEC SQL diagrams, each statement is understood to end with the token *END-EXEC*.

Required Keywords and Parameters

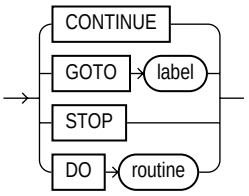
Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the main path, that is, on the horizontal line you are currently traveling. In the following example, cursor is a required parameter:



If there is a cursor named *EMPCURSOR*, then, according to the diagram, the following statement is valid:

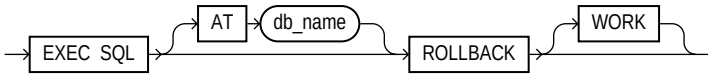
```
EXEC SQL CLOSE EMPCURSOR END-EXEC.
```

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:

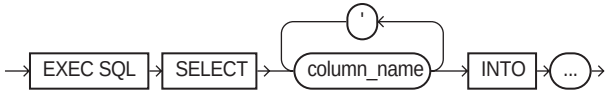


If there is a database named *oracle2*, then, according to the diagram, all of the following statements are valid:

```
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.

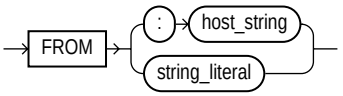


If DEBIT, CREDIT, and BALANCE are column names, then, according to the diagram, all of the following statements are valid:

```
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 statement_name FROM :host_string END-EXEC.
```

Oracle Names

The names of Oracle database objects, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if a name is enclosed by quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle names 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, or to allocate a LOB locator, or a ROWID .

Prerequisites

A cursor variable (see Chapter 6, "Embedded PL/SQL") of type SQL-CURSOR must be declared before allocating memory for the cursor variable.

Syntax

Keywords and Parameters

cursor_variable	A cursor variable of type SQL-CURSOR.
host_ptr	A variable of type SQL-ROWID for a ROWID, or SQL-BLOB, SQL-CLOB, or SQL-NCLOB for a LOB.

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 statement, see *PL/SQL User's Guide and Reference* and *Oracle9i SQL Reference*.

Example

This partial example illustrates the use of the ALLOCATE statement:

```
      ...
01  EMP-CUR       SQL-CURSOR.
01  EMP-REC.
    ...
    EXEC SQL ALLOCATE :EMP-CUR END-EXEC.
    ...
```

Related Topics

CLOSE (Executable Embedded SQL) on page F-14.

EXECUTE (Executable Embedded SQL) on page F-14.

FETCH (Executable Embedded SQL) on page F-47.

FREE (Executable Embedded SQL Extension) on page F-53.

ALLOCATE DESCRIPTOR (Executable Embedded SQL)

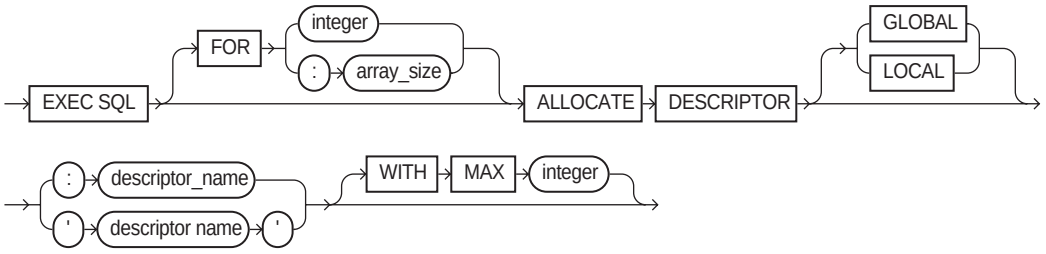
Purpose

An ANSI dynamic SQL statement that allocates a descriptor.

Prerequisites

None.

Syntax



Keywords and Parameters

array_size	Host variable containing number of rows to be processed.
integer	Number of rows to be processed.
descriptor_name	Host variable containing the name of the ANSI descriptor.
descriptor name	Name of the ANSI descriptor.
GLOBAL | LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
WITH MAX *integer*	Maximum number of host variables. The default is 100.

Usage Notes

Use DYNAMIC=ANSI precompiler option. For information on using this statement, see "ALLOCATE DESCRIPTOR" on page 10-13.

Example

```
EXEC SQL
```

```
    FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25
END-EXEC.
```

Related Topics

DESCRIBE DESCRIPTOR (Executable Embedded SQL) on page F-37.

GET DESCRIPTOR (Executable Embedded SQL) on page F-54.

SET DESCRIPTOR (Executable Embedded SQL) on page F-87.

CALL (Executable Embedded SQL)

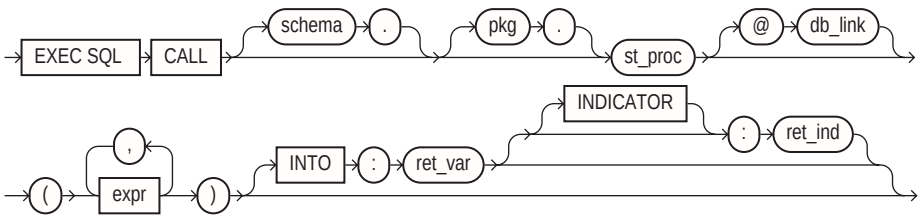
Purpose

To call a stored procedure.

Prerequisites

An active database connection must exist.

Syntax



Keywords and Parameters

schema	Is the schema containing the procedure. If you omit schema, Oracle9*i* assumes the procedure is in your own schema.
pkg	The package where the procedure is stored.
st_proc	The stored procedure to be called.

db_link	The complete or partial name of a database link to a remote database where the procedure is located. For information on referring to database links, see the *Oracle9i SQL Reference*.
expr	The list of expressions that are the parameters of the procedure.
ret_var	The host variable that receives the returned value of a function.
ret_ind	The indicator variable for ret_var.

Usage Notes

For more about this statement, see Calling a Stored PL/SQL or Java Subprogram on page 6-22.

For a complete discussion of stored procedures, see: *Oracle9i Application Developer's Guide - Fundamentals, "External Routines"* chapter.

Example

```
 ...
     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  D-EMP-NUMBER  PIC 9(4).
...
      ACCEPT D-EMP-NUMBER.
      EXEC SQL
        CALL mypkge.getsal(:EMP-NUMBER, :D-EMP-NUMBER, :EMP-NAME) INTO :SALARY
      END-EXEC.
...
```

Related Topics

None

CLOSE (Executable Embedded SQL)

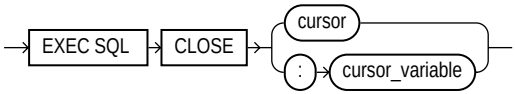
Purpose

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	The cursor to be closed.
cursor_variable	The 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 statement. For information on these options, see Chapter 14, "Precompiler Options".

Example

This example illustrates the use of the CLOSE statement:

```
EXEC SQL CLOSE EMP-CUR END-EXEC.
```

Related Topics

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

OPEN (Executable Embedded SQL) on page F-73.

PREPARE (Executable Embedded SQL) on page F-78.

COMMIT (Executable Embedded SQL)

Purpose

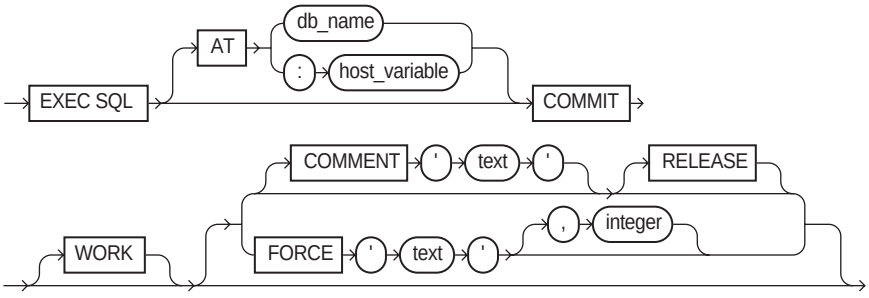
To end your current transaction, making permanent all its changes to the database and optionally freeing all resources and disconnecting from the database server.

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.

Syntax



Keyword and Parameters

AT — Identifies the database to which the COMMIT statement is issued. The database can be identified by either:

- *db_name* — A database identifier declared in a previous DECLARE DATABASE statement or used in a CONNECT statement.
- *host_variable* — A host variable whose value is a previously declared *db_name*.

If you omit this clause, Oracle9*i* 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 Oracle9*i* 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 Oracle9*i* 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 statement and the RELEASE option. Oracle9*i* automatically rolls back changes if the program terminates abnormally.

The COMMIT statement has no effect on host variables or on the flow of control in the program. For more information on this statement, see "Using the COMMIT Statement" on page 3-14.

Example

This example illustrates the use of the embedded SQL COMMIT statement:

```
EXEC SQL AT SALESDB COMMIT RELEASE END-EXEC.
```

Related Topics

ROLLBACK (Executable Embedded SQL) on page F-79.

SAVEPOINT (Executable Embedded SQL) on page F-82.

CONNECT (Executable Embedded SQL Extension)

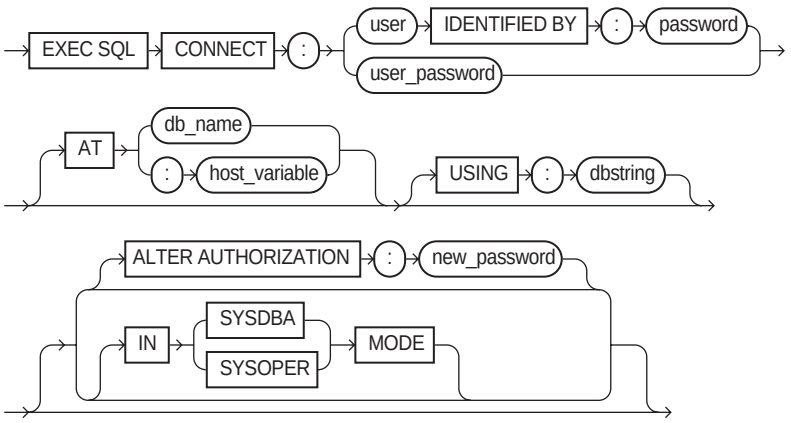
Purpose

To logon to an Oracle9*i* database.

Prerequisites

You must have CREATE SESSION system privilege in the specified database.

Syntax



Keyword and Parameters

user *password*	Specifies your username and password separately.
user_password	Is a single host variable containing the connect string *username*/*password*[@*dbname*]. To allow Oracle9*i* 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* — A database identifier declared in a previous DECLARE DATABASE statement. *host_variable* — A host variable whose value is a previously declared *db_name*.
USING	Specifies the Oracle Net database specification string used to connect to a nondefault 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.

IN SYSDBA MODE IN SYSOPER MODE	Connect with SYSDBA or SYSOPER system privileges. Not allowed when ALTER AUTHORIZATION is used, or precompiler option AUTO_CONNECT is set to YES.

Usage Notes

A program can have multiple connections, but can only connect once to your default database. For more information on this statement, see: "Concurrent Logons" on page 3-3.

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) on page F-15.

DECLARE DATABASE (Oracle Embedded SQL Directive) on page F-26.

ROLLBACK (Executable Embedded SQL) on page F-79.

CONTEXT ALLOCATE (Executable Embedded SQL Extension)

Purpose

To initialize a SQLLIB runtime context that is referenced in an EXEC SQL CONTEXT USE statement.

Prerequisites

The runtime context must be declared of type SQL-CONTEXT.

Syntax

Keywords and Parameters

context The SQLLIB runtime context for which memory is to be allocated.

Usage Notes

For more information on this statement, see "Embedded SQL Statements and Directives for Runtime Contexts" on page 12-8.

Example

This example illustrates the use of a CONTEXT ALLOCATE statement in a Pro*COBOL embedded SQL program:

```
EXEC SQL CONTEXT ALLOCATE :ctx1 END-EXEC.
```

Related Topics

CONTEXT FREE (Executable Embedded SQL Extension) on page F-20.

CONTEXT USE (Oracle Embedded SQL Directive) on page F-21.

CONTEXT FREE (Executable Embedded SQL Extension)

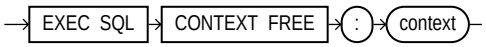
Purpose

To free all memory associated with a runtime context and place a null pointer in the host program variable.

Prerequisites

The CONTEXT ALLOCATE statement must be used to allocate memory for the specified runtime context before the CONTEXT FREE statement can free the memory allocated for it.

Syntax



Keywords and Parameters

context The allocated runtime context for which the memory is to be deallocated.

Usage Notes

For more information on this statement, see "Embedded SQL Statements and Directives for Runtime Contexts" on page 12-8.

Example

This example illustrates the use of a CONTEXT FREE statement in a Pro*COBOL embedded SQL program:

```
EXEC SQL CONTEXT FREE :ctx1 END-EXEC.
```

Related Topics

CONTEXT ALLOCATE (Executable Embedded SQL Extension) on page F-19.

CONTEXT USE (Oracle Embedded SQL Directive) on page F-21.

CONTEXT USE (Oracle Embedded SQL Directive)

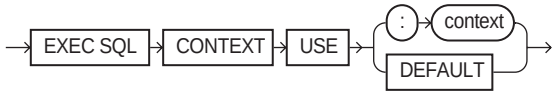
Purpose

To instruct the precompiler to use the specified SQLLIB runtime context on subsequent executable SQL statements

Prerequisites

The runtime context specified by the CONTEXT USE directive must be previously declared.

Syntax



Keywords and Parameters

context	The allocated runtime context to use for subsequent executable SQL statements that follow it. For example, after specifying in your source code which context to use (multiple contexts can be allocated), you can connect to the Oracle Server and perform database operations within the scope of that context.
DEFAULT	Indicates that the global context is to be used.

Usage Notes

This statement has no effect on declarative statements such as EXEC SQL INCLUDE or EXEC ORACLE OPTION. It works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given source file without regard to standard C scope rules.

For more information on this statement, see "Embedded SQL Statements and Directives for Runtime Contexts" on page 12-8.

Example

This example illustrates the use of a CONTEXT USE directive in a Pro*COBOL program:

```
EXEC SQL CONTEXT USE :ctx1 END-EXEC.
```

Related Topics

CONTEXT ALLOCATE (Executable Embedded SQL Extension) on page F-19.

CONTEXT FREE (Executable Embedded SQL Extension) on page F-20.

DEALLOCATE DESCRIPTOR (Embedded SQL Statement)

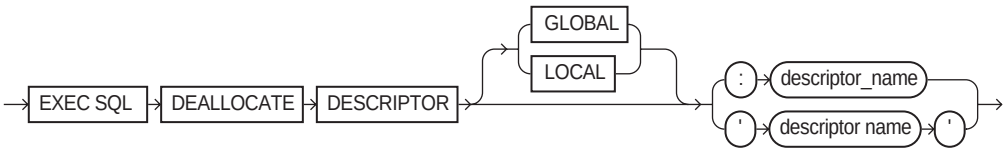
Purpose

An ANSI dynamic SQL statement that deallocates a descriptor area to free memory.

Prerequisites

The descriptor specified by the DEALLOCATE DESCRIPTOR statement must be previously allocated using the ALLOCATE DESCRIPTOR statement.

Syntax



Keywords and Parameters

GLOBAL | LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
descriptor_name	Host variable containing the name of the allocated ANSI descriptor.
'descriptor name'	Name of the allocated ANSI descriptor.

Usage Notes

Use DYNAMIC=ANSI precompiler option.

For more information on this statement, see "DEALLOCATE DESCRIPTOR" on page 10-14.

Example

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES'  END-EXEC.
```

Related Topics

ALLOCATE DESCRIPTOR (Executable Embedded SQL) on page F-12.

DESCRIBE DESCRIPTOR (Executable Embedded SQL) on page F-37.

GET DESCRIPTOR (Executable Embedded SQL) on page F-54.

PREPARE (Executable Embedded SQL) on page F-78.

SET DESCRIPTOR (Executable Embedded SQL) on page F-87.

DECLARE CURSOR (Embedded SQL Directive)

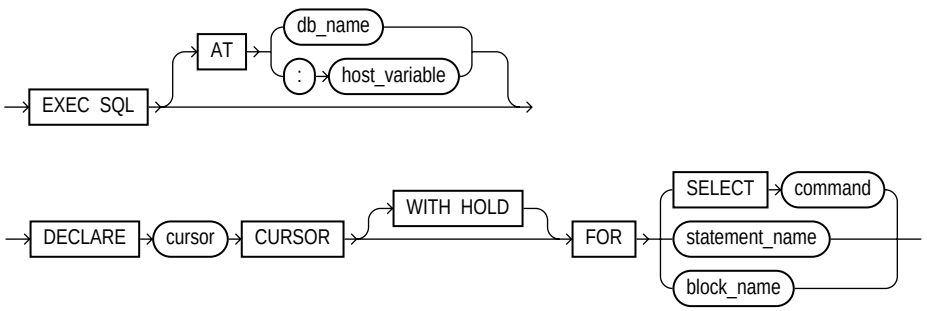
Purpose

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 — Identifies the database on which the cursor is declared. The database can be identified by either:

db_name — Database identifier declared in a previous DECLARE DATABASE statement.

	host_variable	Host variable whose value is a previously declared *db_name*.
	If you omit this clause, Oracle9*i* declares the cursor on your default database.	
cursor	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 *statement*	Is a SELECT statement to be associated with the cursor. The following statement cannot contain an INTO clause.	
statement_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, if the cursor has been opened with an OPEN statement and positioned on a row with a FETCH statement. For more information on this statement, see "WITH HOLD Clause in DECLARE CURSOR Statements" on page 3-15.

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
END-EXEC.
```

Related Topics

CLOSE (Executable Embedded SQL) on page F-14.

DECLARE DATABASE (Oracle Embedded SQL Directive) on page F-26.

DECLARE STATEMENT (Embedded SQL Directive) on page F-27.

DELETE (Executable Embedded SQL) on page F-31.

FETCH (Executable Embedded SQL) on page F-47.

OPEN (Executable Embedded SQL) on page F-73.

PREPARE (Executable Embedded SQL) on page F-78.

SELECT (Executable Embedded SQL) on page F-84.

UPDATE (Executable Embedded SQL) on page F-90.

DECLARE DATABASE (Oracle Embedded SQL Directive)

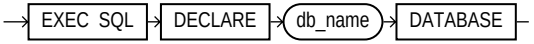
Purpose

To declare an identifier for a nondefault database to be accessed in subsequent embedded SQL statements.

Prerequisites

You must have access to a username on the nondefault database.

Syntax



Keywords and Parameters

db_name The identifier established for the nondefault database.

Usage Notes

You declare a *db_name* for a nondefault 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 nondefault database with a DECLARE DATABASE statement.

For more information on this statement, see "Using Username/Password" on page 3-5.

Example

This example illustrates the use of a DECLARE DATABASE directive:

```
EXEC SQL DECLARE ORACLE3 DATABASE END-EXEC.
```

Related Topics

COMMIT (Executable Embedded SQL) on page F-15

CONNECT (Executable Embedded SQL Extension) on page F-17.

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

DECLARE STATEMENT (Embedded SQL Directive) on page F-27.

DELETE (Executable Embedded SQL) on page F-31.

EXECUTE (Executable Embedded SQL) on page F-41.

EXECUTE IMMEDIATE (Executable Embedded SQL) on page F-45.

INSERT (Executable Embedded SQL) on page F-57.

SELECT (Executable Embedded SQL) on page F-84.

UPDATE (Executable Embedded SQL) on page F-90.

DECLARE STATEMENT (Embedded SQL Directive)

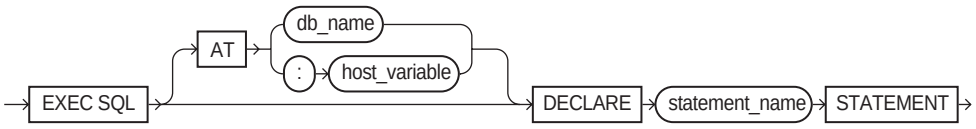
Purpose

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* — Database identifier declared in a previous DECLARE DATABASE statement.
- *host_variable* — Host variable whose value is a previously declared *db_name*.

If you omit this clause, Oracle9*i* declares the SQL statement or PL/SQL block on your default database.

statement_name — Is the declared identifier for the statement or 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 statement, see "DECLARE" on page 9-19.

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, 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) on page F-14.

DECLARE DATABASE (Oracle Embedded SQL Directive) on page F-26.

FETCH (Executable Embedded SQL) on page F-47.

OPEN (Executable Embedded SQL) on page F-73.

PREPARE (Executable Embedded SQL) on page F-78.

DECLARE TABLE (Oracle Embedded SQL Directive)

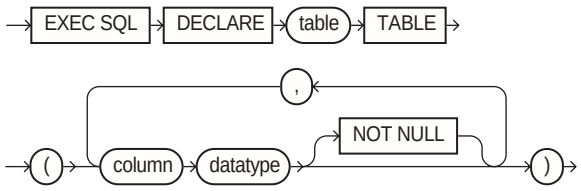
Purpose

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	The name of the declared table.
column	A column of the *table*.
datatype	The datatype of a *column*. For information on Oracle9*i* datatypes, see "The Oracle9i Datatypes" on page 4-2.
NOT NULL	Specifies that a *column* cannot contain nulls.

Usage Notes

Datatypes can only use integers (not expressions) for length, precision, scale. For more information on using this statement, see "Specifying SQLCHECK=SEMANTICS" 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.

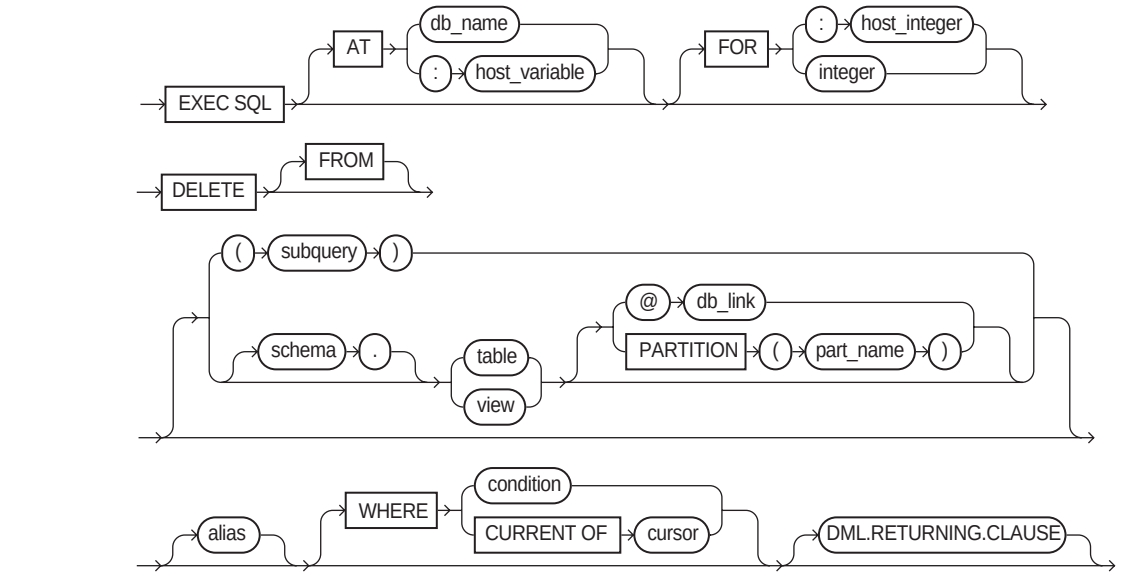
Prerequisites

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.

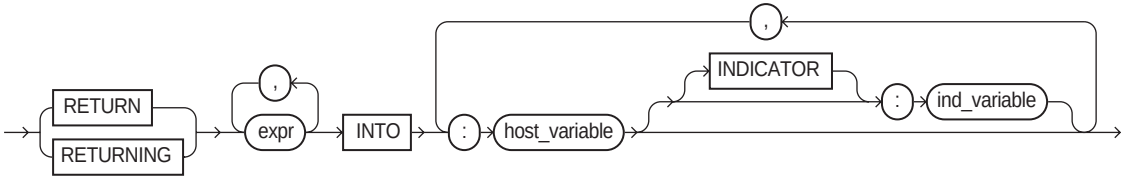
For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

The DELETE ANY TABLE system privilege also enables you to delete rows from any table or any view's base table.

Syntax



where the DML Returning clause is:



Keywords and Parameters

AT

Identifies the database to which the DELETE statement is issued. The database can be identified by either:

db_name

A database identifier declared in a previous DECLARE DATABASE statement.

	host_variable	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.	
host_integer *integer*	Limits the number of times the statement is executed if the WHERE clause contains array host variables. If you omit this clause, Oracle9*i* executes the statement once for each component of the smallest array.	
schema	The schema containing the table or view. If you omit *schema*, Oracle9*i* assumes the table or view is in your own schema.	
table view	The name of a table from which the rows are to be deleted. If you specify *view*, Oracle9*i* deletes rows from the view's base table.	
dblink	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 *Oracle9i SQL Reference*. You can only delete rows from a remote table or view if you are using Oracle9*i* with the distributed option.	
	If you omit *dblink*, Oracle9 assumes that the table or view is located on the local database.	
part_name	Name of partition in the table	
alias	The 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 *Oracle9i 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 you specify the FOR UPDATE clause to specifically lock only one table.
	If you omit this clause entirely, Oracle9*i* deletes all rows from the table or view.	
DML returning clause	See "DML Returning Clause" on page 5-9 for a discussion.	

Usage Notes

The host variables in the WHERE clause should be either all scalars or all arrays. If they are scalars, Oracle9*i* executes the DELETE statement only once. If they are arrays, Oracle9*i* 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 Oracle9*i* 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, Oracle9*i* returns an error through the SQLCODE of the SQLCA. If you omit the WHERE clause, Oracle9*i* raises a warning flag in the fifth component of SQLWARN in the SQLCA. For more information on this statement and the SQLCA, see "Using the SQL Communications Area" on page 8-6.

You can use comments in a DELETE statement to pass instructions, or *hints*, to the Oracle9*i* optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle9i Database Performance Guide and Reference.*

Example

This example illustrates the use of the DELETE statement:

```
 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) on page F-26.

DECLARE STATEMENT (Embedded SQL Directive) on page F-27.

DESCRIBE (Executable Embedded SQL)

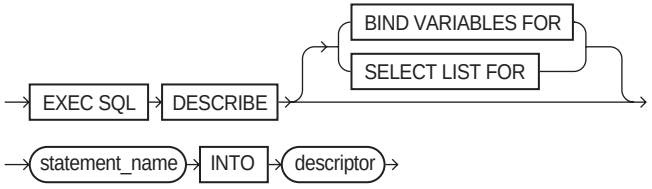
Purpose

To initialize a descriptor to hold descriptions of host variables for an Oracle 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 FOR	Initializes the descriptor to hold information about the input variables for the SQL statement or PL/SQL block.
SELECT LIST FOR	Initializes the descriptor to hold information about the select list of a SELECT statement. The default is SELECT LIST FOR.

statement_name	Identifies a SQL statement or PL/SQL block previously prepared with a PREPARE statement.
descriptor	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 statement, see "The DESCRIBE Statement" on page 9-25.

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) on page F-78.

DESCRIBE DESCRIPTOR (Executable Embedded SQL)

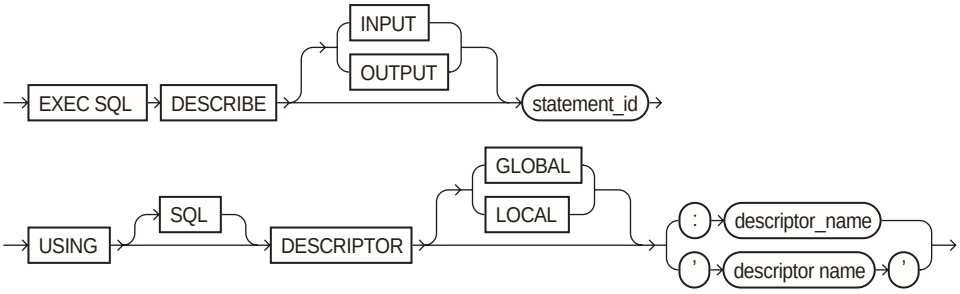
Purpose

Used to obtain information about an ANSI SQL statement, and to store it in a descriptor.

Prerequisites

You must have prepared the SQL statement in a previous embedded SQL PREPARE statement.

Syntax



Keywords and Parameters

statement_id	The name of the previously prepared SQL statement or PL/SQL block. OUTPUT is the default.
desc_name	Host variable containing the name of the descriptor that will hold information about the SQL statement.
'descriptor name'	Name of the descriptor
GLOBAL | LOCAL	LOCAL is the default. It means file scope, as opposed to GLOBAL, which means application scope.

Usage Notes

Use DYNAMIC=ANSI precompiler option. Only COUNT and NAME are implemented for the INPUT descriptor.

The number of variables found by a DESCRIBE statement is the total number of place-holders in the prepare SQL statement or PL/SQL block, rather than the total number of uniquely named place-holders. For more information on this statement, see Chapter 10, "ANSI Dynamic SQL".

Example

```
EXEC SQL PREPARE s FROM :my_stament END-EXEC.
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' END-EXEC.
```

Related Topics

ALLOCATE DESCRIPTOR (Executable Embedded SQL) on page F-12.

DEALLOCATE DESCRIPTOR (Embedded SQL Statement) on page F-23.

GET DESCRIPTOR (Executable Embedded SQL) on page F-54.

PREPARE (Executable Embedded SQL) on page F-87.

SET DESCRIPTOR (Executable Embedded SQL) on page F-87.

ENABLE THREADS (Executable Embedded SQL Extension)

Purpose

To initialize a process that supports multiple threads.

Prerequisites

You must be developing a precompiler application for and compiling it on a platform that supports multithreaded applications, and THREADS=YES must be specified on the command line.

Syntax

Keywords and Parameters

None.

Usage Notes

The ENABLE THREADS statement must be executed once, and only once, before any other executable SQL statement and before spawning any threads. This statement does not require a host-variable specification.

Example

This example illustrates the use of the ENABLE THREADS statement in a Pro*COBOL program:

```
EXEC SQL ENABLE THREADS END-EXEC.
```

Related Topics

CONTEXT ALLOCATE (Executable Embedded SQL Extension) on page F-19.

CONTEXT FREE (Executable Embedded SQL Extension) on page F-20.

CONTEXT USE (Oracle Embedded SQL Directive) on page F-21.

EXECUTE ... END-EXEC (Executable Embedded SQL Extension)

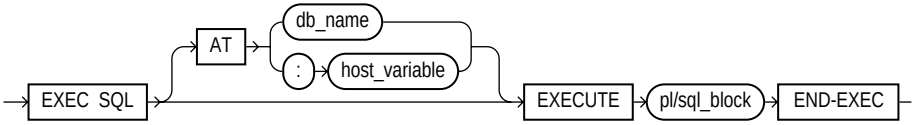
Purpose

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* — A database identifier declared in a previous DECLARE DATABASE statement.
- *host_variable* — 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.

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, "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) on page F-45.

EXECUTE (Executable Embedded SQL)

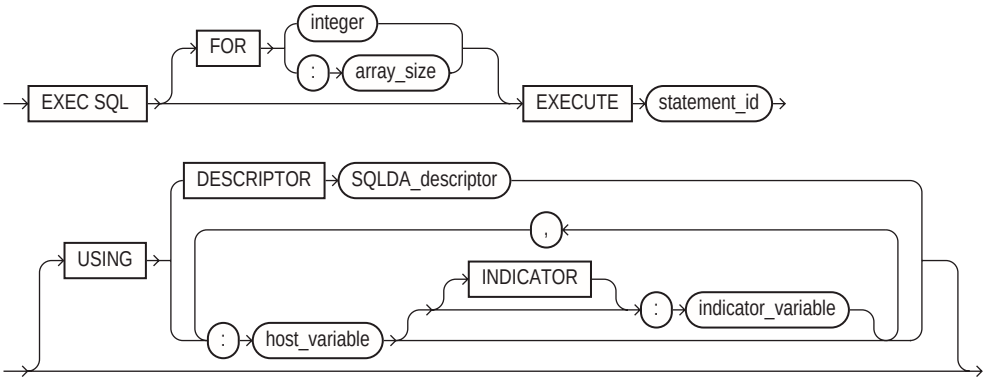
Purpose

In Oracle dynamic SQL, 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 :*array_size*	Host variable containing the number of rows to be processed.
FOR *integer*	Number of rows to be processed. Limits the number of times the statement is executed when the USING clause contains array host variables If you omit this clause, Oracle9*i* executes the statement once for each component of the smallest array.
statement_id	A precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE statement to associate the precompiler identifier with the statement or PL/SQL block.
USING DESCRIPTOR *SQLDA_descriptor*	Uses an Oracle descriptor. CANNOT be used together with an ANSI descriptor (INTO clause).
USING	Specifies a list of host variables with optional indicator variables that Oracle9*i* substitutes as input variables into the statement to be executed. The host and indicator variables must be either all scalars or all arrays.
host_variable	Host variables.
indicator_variable	Indicator variables.

Usage Note

For more information on this statement, see Chapter 9, "Oracle 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) on page F-26.

PREPARE (Executable Embedded SQL) on page F-78.

EXECUTE DESCRIPTOR (Executable Embedded SQL

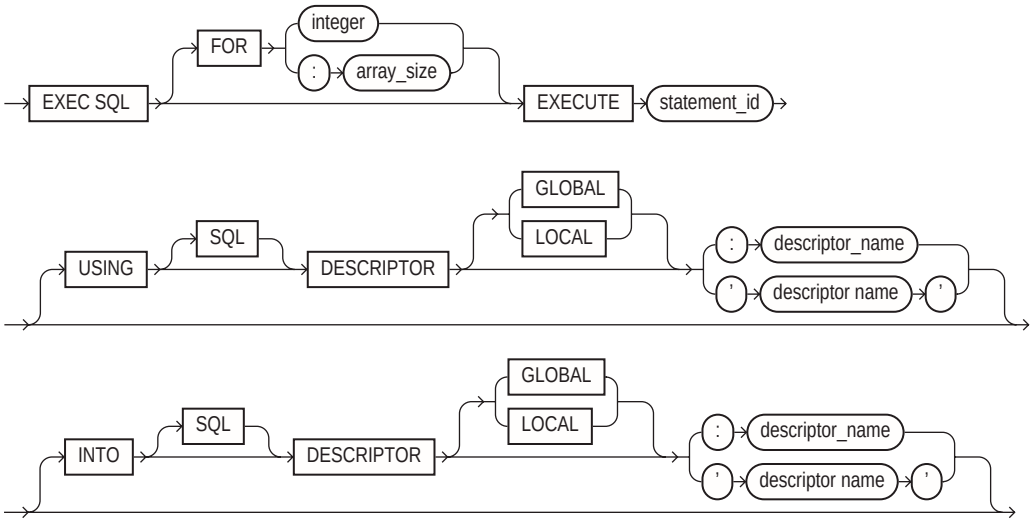
Purpose

In ANSI SQL Method 4, 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 *:array_size*	Host variable containing the number of rows to be processed.
FOR *integer*	Number of rows to be processed. Limits the number of times the statement is executed. Oracle9*i* executes the statement once for each component of the smallest array.
statement_id	A precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE statement to associate the precompiler identifier with the statement or PL/SQL block.
USING	An ANSI input descriptor.
descriptor_name	Host variable containing name of the input descriptor.
descriptor name	Name of the input descriptor.
INTO	An ANSI output descriptor.
descriptor_name	Host variable containing the name of the output descriptor.
descriptor name	Name of the output descriptor.

GLOBAL | LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

Usage Notes

For more information on this statement, see "EXECUTE" on page 10-23.

Examples

The ANSI dynamic SQL Method 4 enables DML RETURNING in a SELECT to be supported by the INTO clause in EXECUTE:

```
EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' END-EXEC.
```

Related Topics

DECLARE DATABASE (Oracle Embedded SQL Directive) on page F-26.

PREPARE (Executable Embedded SQL) on page F-78.

EXECUTE IMMEDIATE (Executable Embedded SQL)

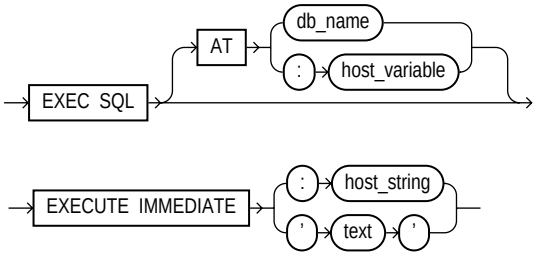
Purpose

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* — A database identifier declared in a previous DECLARE DATABASE statement.
- *host_variable* — 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 — A host variable whose value is the SQL statement or PL/SQL block to be executed.

text — A text literal containing the SQL statement or PL/SQL block to be executed. The quotes may be omitted.

The SQL statement can only be a DELETE, INSERT, or UPDATE statement.

Usage Notes

When you issue an EXECUTE IMMEDIATE statement, Oracle9*i* 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 statement, see "The EXECUTE IMMEDIATE Statement" on page 9-8.

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) on page F-78.

EXECUTE (Executable Embedded SQL) on page F-41.

FETCH (Executable Embedded SQL)

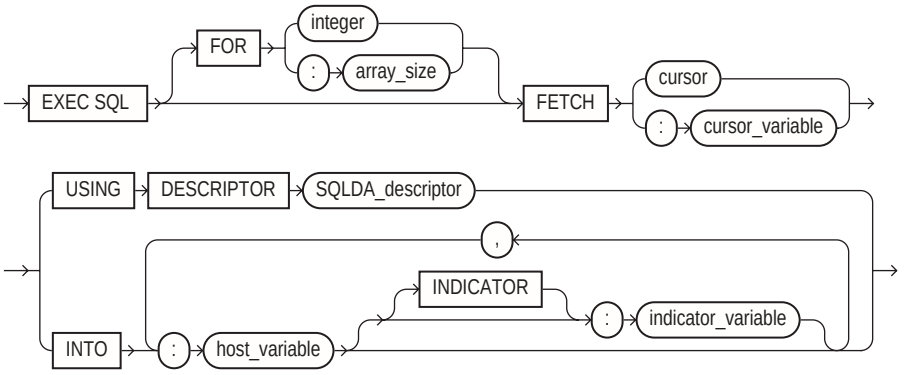
Purpose

To retrieve one or more rows returned by a query, assigning the select list values to host variables. For ANSI Dynamic SQL Method 4, see "FETCH DESCRIPTOR (Executable Embedded SQL)" on page F-50.

Prerequisites

You must first open the cursor with an the OPEN statement.

Syntax



Keywords and Parameters

FOR *:array_size*	Host variable containing the number of rows to be processed. Number of rows to be processed.
FOR *integer*	Limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle9*i* fetches enough rows to fill the smallest array.
cursor	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	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 *SQLDA_variable*	Specifies the Oracle descriptor referenced in a previous DESCRIBE statement. Only use this clause with dynamic embedded SQL, method 4. The USING clause does not apply when a cursor variable is used.
host_variable	The host variable into which data is returned.
i*ndicator_variable*	The host indicator variable.

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 number of rows retrieved is specified by the size of the output host variables or the value specified in the FOR clause. The host variables to receive the data should be either all scalars or all arrays. If they are scalars, Oracle9*i* fetches only one row. If they are arrays, Oracle9*i* fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle9*i* 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 warning code returned in the SQLCODE element of the SQLCA.

If the array is not completely filled then the warning is issued and you should check SQLERRD(3) to see how many rows were actually fetched.

Note that the FETCH statement 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 statement 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

CLOSE (Executable Embedded SQL) on page F-14.

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

OPEN (Executable Embedded SQL) on page F-73.

PREPARE (Executable Embedded SQL) on page F-78.

FETCH DESCRIPTOR (Executable Embedded SQL)

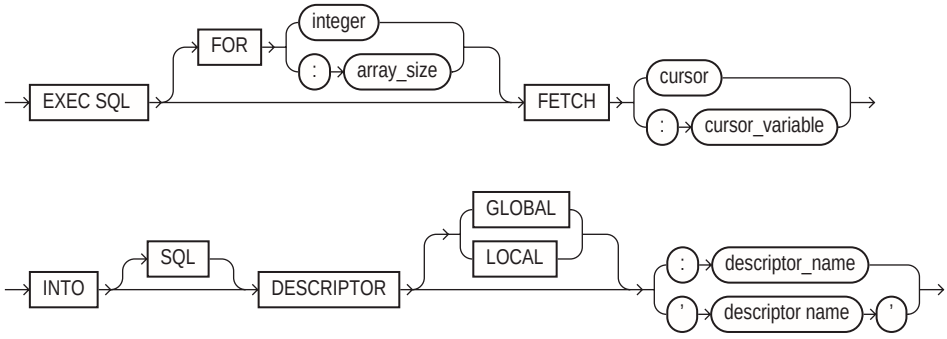
Purpose

To retrieve one or more rows returned by a query, assigning the select list values to host variables. Used in ANSI Dynamic SQL Method 4.

Prerequisites

You must first open the cursor with an the OPEN statement.

Syntax



Keywords and Parameters

FOR :array_size	Host variable containing the number of rows to be processed.
FOR integer	Number of rows to be processed. Limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle9**i** fetches enough rows to fill the smallest array.
cursor	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	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.
INTO *'descriptor name'*	Name of the output ANSI descriptor.
INTO *:descriptor_name*	Host variable containing the name of the output descriptor.
GLOBAL | LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

Usage Notes

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

should be either all scalars or all arrays. If they are scalars, Oracle9*i* fetches only one row. If they are arrays, Oracle9*i* fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle9*i* fetches is determined by the smaller of the following values:

- The size of the smallest array
- The value of the *array_size* 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 a warning code returned in the SQLCODE element of the SQLCA.

If the array is not completely filled then the warning is issued and you should check SQLERRD(3) to see how many rows were actually fetched.

Note that the FETCH statement 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.

Use DYNAMIC=ANSI precompiler option for the ANSI SQL Method 4 application. For more information, see "FETCH" on page 10-27 for the ANSI SQL Method 4 application.

Example

```
...
EXEC SQL ALLOCATE DESCRIPTOR 'output_descriptor' END-EXEC.
...
EXEC SQL PREPARE S FROM :dyn_statement END-EXEC.
EXEC SQL DECLARE mycursor CURSOR FOR S END-EXEC.
...
EXEC SQL FETCH mycursor INTO DESCRIPTOR 'output_descriptor' END-EXEC.
...
```

Related Topics

PREPARE statement on page F-78.

FREE (Executable Embedded SQL Extension)

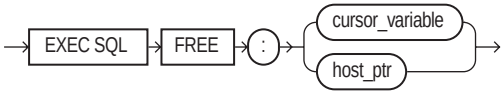
Purpose

To free memory used by a cursor, LOB locator, or ROWID.

Prerequisites

The memory has to have been already allocated.

Syntax



Keywords and Parameters

cursor_variable	A cursor variable that has previously been allocated in an ALLOCATE statement. It is of type SQL-CURSOR. The FETCH statement returns one of the rows selected by the query associated with the cursor variable.
host_ptr	A variable of type SQL-ROWID for a ROWID, or SQL-BLOB, SQL-CLOB, or SQL-NCLOB for a LOB.

Usage Notes

See "Cursors" on page 5-11 and "Cursor Variables" on page 6-30.

Example

```
* CURSOR VARIABLE EXAMPLE
...
 01  CUR      SQL-CURSOR.
...
     EXEC SQL ALLOCATE :CUR END-EXEC.
...
     EXEC SQL CLOSE :CUR END-EXEC.
     EXEC SQL FREE  :CUR END-EXEC.
...
```

Related Topics

ALLOCATE (Executable Embedded SQL Extension) on page F-10.

CLOSE (Executable Embedded SQL) on page F-14.

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

GET DESCRIPTOR (Executable Embedded SQL)

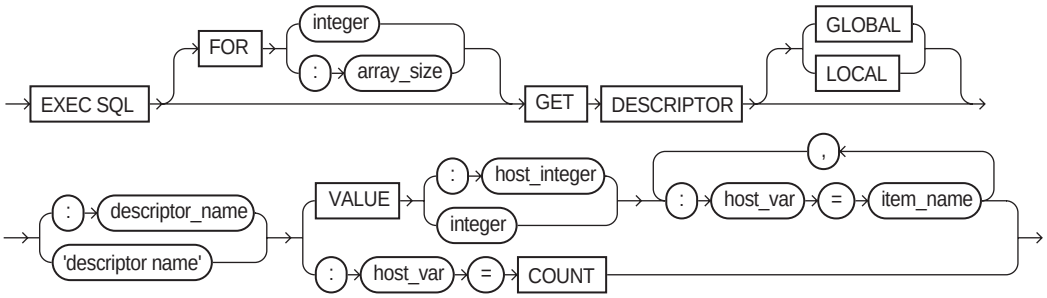
Purpose

To obtain information about host variables from a SQL descriptor area.

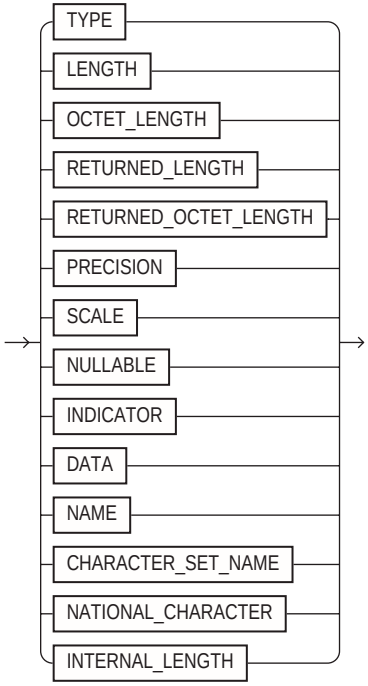
Prerequisites

Use only with value semantics and ANSI dynamic SQL Method 4.

Syntax



where `item_name` can be one of these choices:



Keywords and Parameters

array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
:descriptor_name	Host variable containing the name of the allocated ANSI descriptor.
'descriptor name'	Name of the allocated ANSI descriptor.
GLOBAL | LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
host_var=COUNT	Host variable containing the total number of input or output variables.
integer	Total number of input or output variables.
VALUE *:host_integer*	Host variable containing the position of the referenced input or output variable.
VALUE *integer*	The position of the referenced input or output variable.
host_var	Host variable which will receive the item's value.
item_name	The *item_name* is found in Table 10–4 on page 10-16, and Table 10–5 on page 10-17, under the "Descriptor Item Name" column heading.

Usage Notes

Use DYNAMIC=ANSI precompiler option. The array size clause can be used with DATA, RETURNED_LENGTH, and INDICATOR item names. See "GET DESCRIPTOR" on page 10-15.

Example

```
EXEC SQL GET DESCRIPTOR GLOBAL 'mydesc' :mydesc_num_vars = COUNT END-EXEC.
```

Related Topics

ALLOCATE DESCRIPTOR (Executable Embedded SQL) on page F-12.

DESCRIBE DESCRIPTOR (Executable Embedded SQL) on page F-37.

SET DESCRIPTOR (Executable Embedded SQL) on page F-87.

INSERT (Executable Embedded SQL)

Purpose

To add rows to a table or to a view's base table.

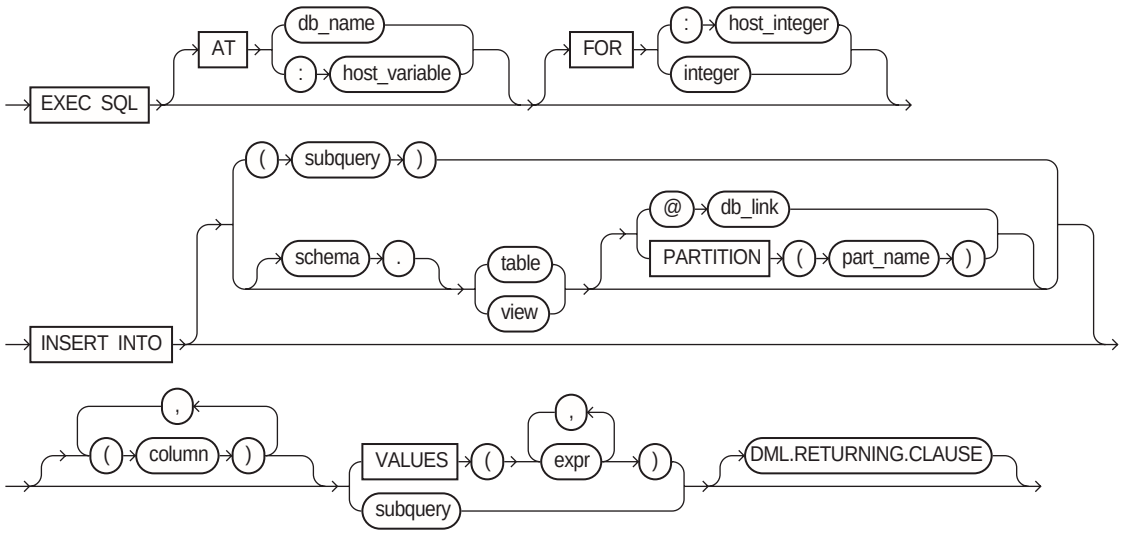
Prerequisites

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.

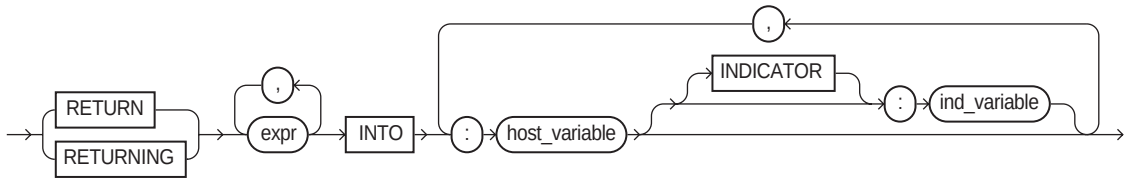
For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

The INSERT ANY TABLE system privilege also enables you to insert rows into any table or any view's base table.

Syntax



where DML returning clause is:



Keywords and Parameters

AT

Identifies the database on which the INSERT statement is executed. The database can be identified by either:

db_name — A database identifier declared in a previous DECLARE DATABASE statement.

host_variable — 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, Oracle9*i* executes the statement once for each component in the smallest array.
schema	The schema containing the table or view. If you omit *schema*, Oracle9*i* assumes the table or view is in your own schema.
table view	The name of the table into which rows are to be inserted. If you specify *view*, Oracle9*i* inserts rows into the view's base table.
db_link	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 *Oracle9i SQL Reference.* You can only insert rows into a remote table or view if you are using Oracle9*i* with the distributed option. If you omit *db_link*, Oracle9*i* assumes that the table or view is on the local database.
part_name	The name of partition in the table
column	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 *Oracle9i 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	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 *Oracle9i SQL Reference.*
DML returning clause	See "DML Returning Clause" on page 5-9 for a discussion.

Usage Notes

Any host variables that appear in the WHERE clause should be either all scalars or all arrays. If they are scalars, Oracle9 executes the INSERT statement once. If they are arrays, Oracle9*i* 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 Oracle9*i* 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 statement, see "The Basic SQL Statements" on page 5-7.

Example I

This example illustrates the use of the embedded SQL INSERT statement:

```
EXEC SQL
    INSERT INTO EMP (ENAME, EMPNO, SAL)
    VALUES (:ENAME, :EMPNO, :SAL)
END-EXEC.
```

Example II

This example shows an embedded SQL INSERT statement 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) on page F-26.

LOB APPEND (Executable Embedded SQL Extension)

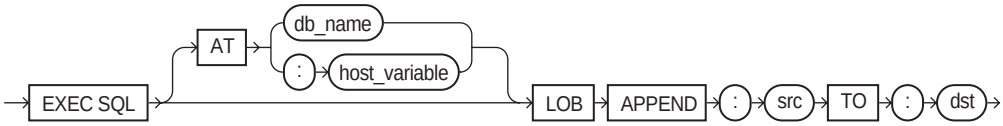
Purpose

To append a LOB to the end of another LOB.

Prerequisites

LOB buffering must not be enabled.The destination LOB must have been initialized.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "APPEND" on page 13-10.

Related Topics

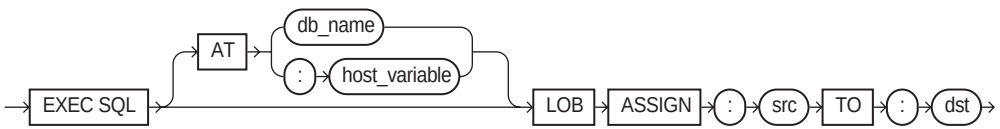
See the other LOB statements.

LOB ASSIGN (Executable Embedded SQL Extension)

Purpose

To assign a LOB or BFILE locator to another locator.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "ASSIGN" on page 13-11.

Related Topics

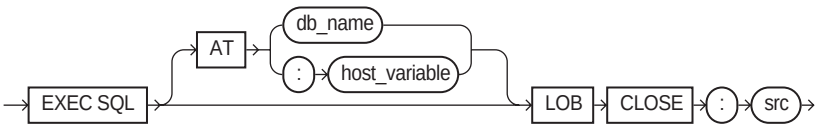
See the other LOB statements.

LOB CLOSE (Executable Embedded SQL Extension)

Purpose

To close an open LOB or BFILE.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "CLOSE" on page 13-12.

Related Topics

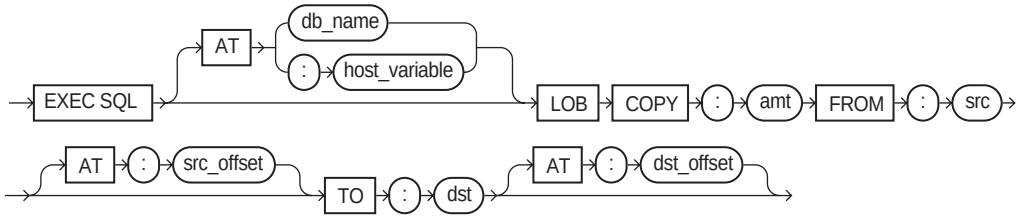
See the other LOB statements.

LOB COPY (Executable Embedded SQL Extension)

Purpose

To copy all or part of a LOB value into another LOB.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "COPY" on page 13-13.

Related Topics

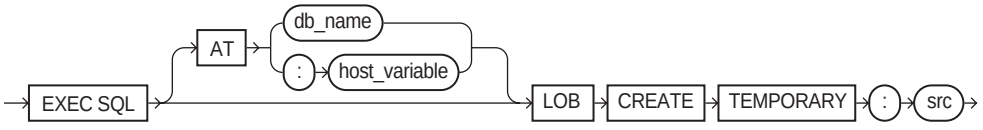
See the other LOB statements.

LOB CREATE TEMPORARY (Executable Embedded SQL Extension)

Purpose

To create a temporary LOB.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "CREATE TEMPORARY" on page 13-14.

Related Topics

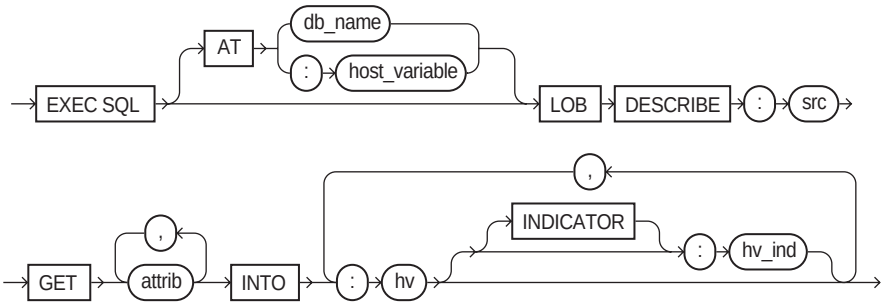
See the other LOB statements.

LOB DESCRIBE (Executable Embedded SQL Extension)

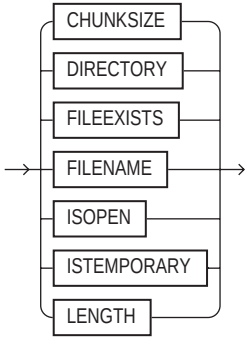
Purpose

To retrieve attributes from a LOB.

Syntax



where `attrib` is:



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "DESCRIBE" on page 13-24.

Related Topics

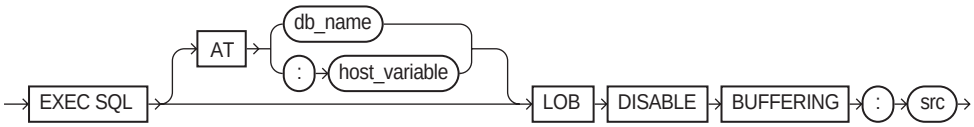
See the other LOB statements.

LOB DISABLE BUFFERING (Executable Embedded SQL Extension)

Purpose

To disable LOB buffering.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "DISABLE BUFFERING" on page 13-14.

Related Topics

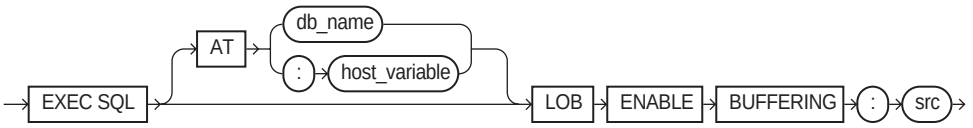
See the other LOB statements.

LOB ENABLE BUFFERING (Executable Embedded SQL Extension)

Purpose

To enable LOB buffering.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see ENABLE BUFFERING on page 13-15

Related Topics

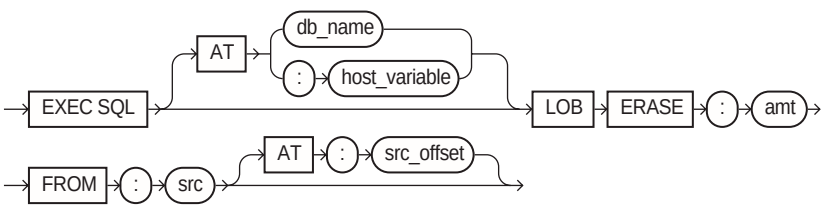
See the other LOB statements.

LOB ERASE (Executable Embedded SQL Extension)

Purpose

To erase a given amount of LOB data starting from a given offset.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "ERASE" on page 13-15.

Related Topics

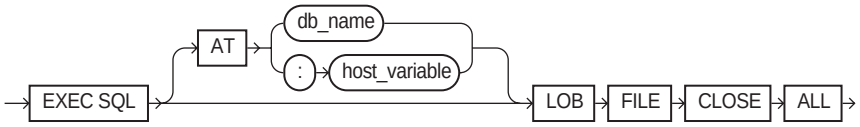
See the other LOB statements.

LOB FILE CLOSE ALL (Executable Embedded SQL Extension)

Purpose

To close all open BFILEs in the current session.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "FILE CLOSE ALL" on page 13-16.

Related Topics

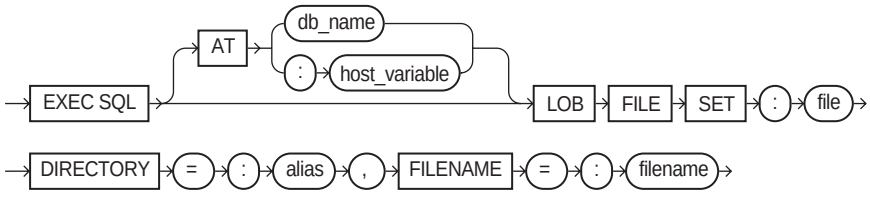
See the other LOB statements.

LOB FILE SET (Executable Embedded SQL Extension)

Purpose

To set DIRECTORY and FILENAME in a BFILE locator.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "FILE SET" on page 13-17.

Related Topics

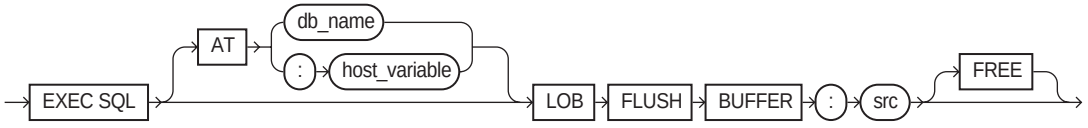
See the other LOB statements.

LOB FLUSH BUFFER (Executable Embedded SQL Extension)

Purpose

To write the LOB buffers to the database server.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "FLUSH BUFFER" on page 13-17.

Related Topics

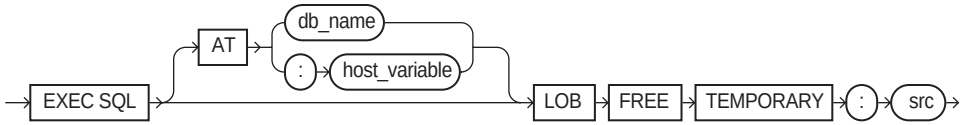
See the other LOB statements.

LOB FREE TEMPORARY (Executable Embedded SQL Extension)

Purpose

To free temporary space for the LOB locator.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "FREE TEMPORARY" on page 13-18.

Related Topics

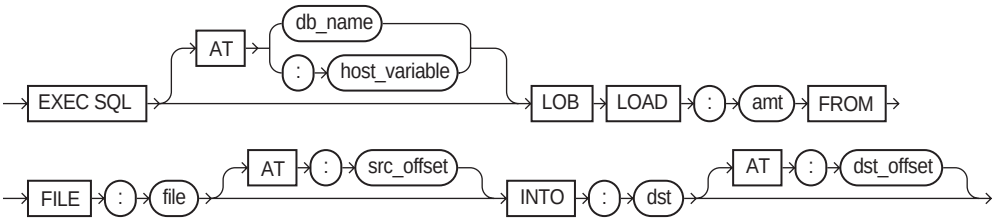
See the other LOB statements.

LOB LOAD (Executable Embedded SQL Extension)

Purpose

To copy all or part of a BFILE into an internal LOB.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "LOAD FROM FILE" on page 13-18.

Related Topics

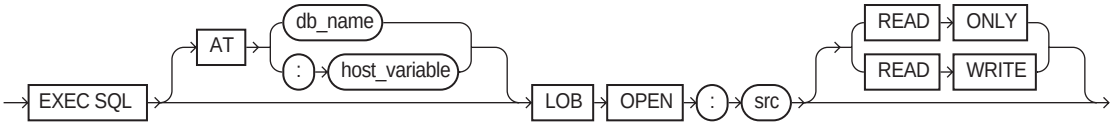
See the other LOB statements.

LOB OPEN (Executable Embedded SQL Extension)

Purpose

To open a LOB or BFILE for read or read / write access.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "OPEN" on page 13-20.

Related Topics

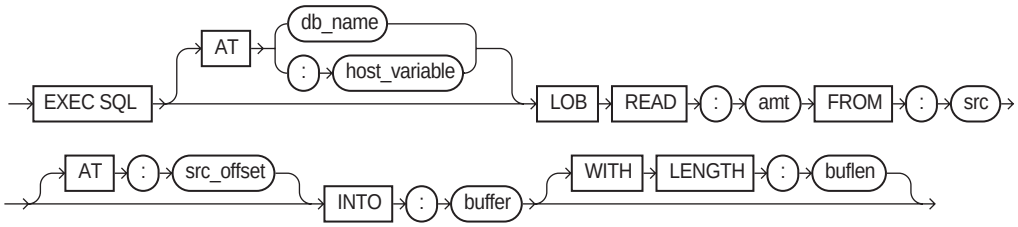
See the other LOB statements.

LOB READ (Executable Embedded SQL Extension)

Purpose

To read all or part of a LOB or BFILE into a buffer.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "READ" on page 13-20.

Related Topics

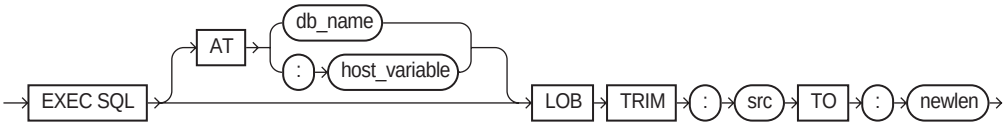
See the other LOB statements.

LOB TRIM (Executable Embedded SQL Extension)

Purpose

To truncate a LOB value.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "TRIM" on page 13-22.

Related Topics

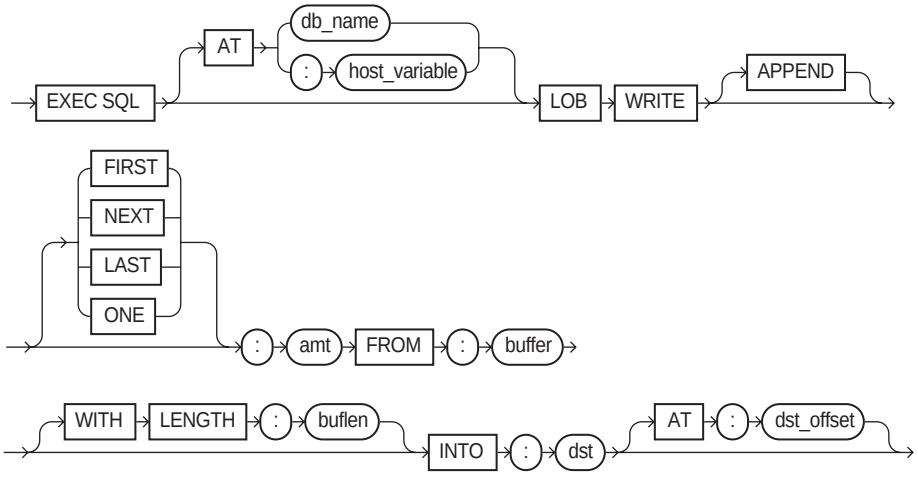
See the other LOB statements.

LOB WRITE (Executable Embedded SQL Extension)

Purpose

To write the contents of a buffer to a LOB.

Syntax



Usage Notes

For usage notes as well as keywords, parameters, and examples, see "WRITE" on page 13-23.

Related Topics

See the other LOB statements.

OPEN (Executable Embedded SQL)

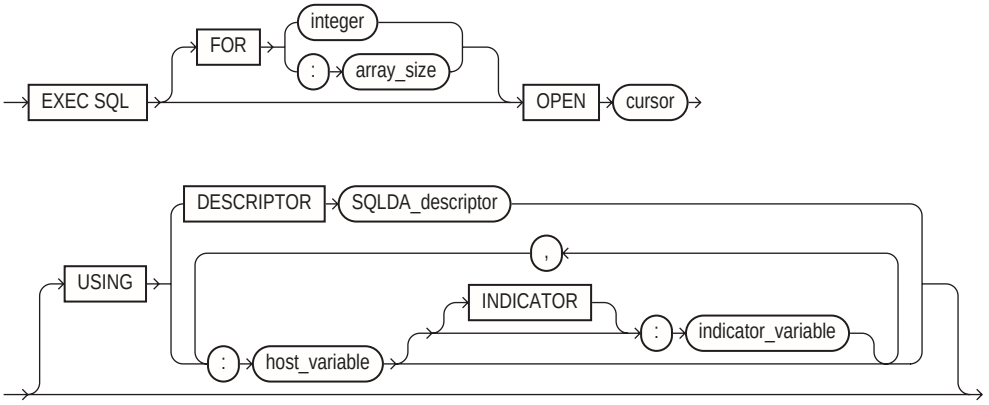
Purpose

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. It can be used in place of EXECUTE in dynamic SQL. For the ANSI Dynamic SQL syntax, see "OPEN DESCRIPTOR (Executable Embedded SQL)" on page F-75.

Prerequisites

You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.

Syntax



Keywords and Parameters

array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed. Can only be used when the OPEN is equivalent to EXECUTE.
cursor	The (previously declared) cursor to be opened.
host_variable	Specifies a host variable with an optional indicator variable to be substituted into the statement associated with the cursor. CANNOT be used together with an ANSI descriptor (INTO clause).
DESCRIPTOR *SQLDA_descriptor*	Specifies an Oracle 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. CANNOT be used together with an ANSI descriptor (INTO clause).

Usage Notes

The OPEN statement 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 statement does not actually retrieve rows; rows are retrieved by the FETCH statement.

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 statement.

You can reopen a cursor without first closing it. For more information on this statement, see "Opening a Cursor" on page 5-14.

Example

This example illustrates the use of the OPEN statement in a Pro*COBOL 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

CLOSE (Executable Embedded SQL) on page F-14.

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

EXECUTE (Executable Embedded SQL) on page F-41.

FETCH (Executable Embedded SQL) on page F-47.

PREPARE (Executable Embedded SQL) on page F-78.

OPEN DESCRIPTOR (Executable Embedded SQL)

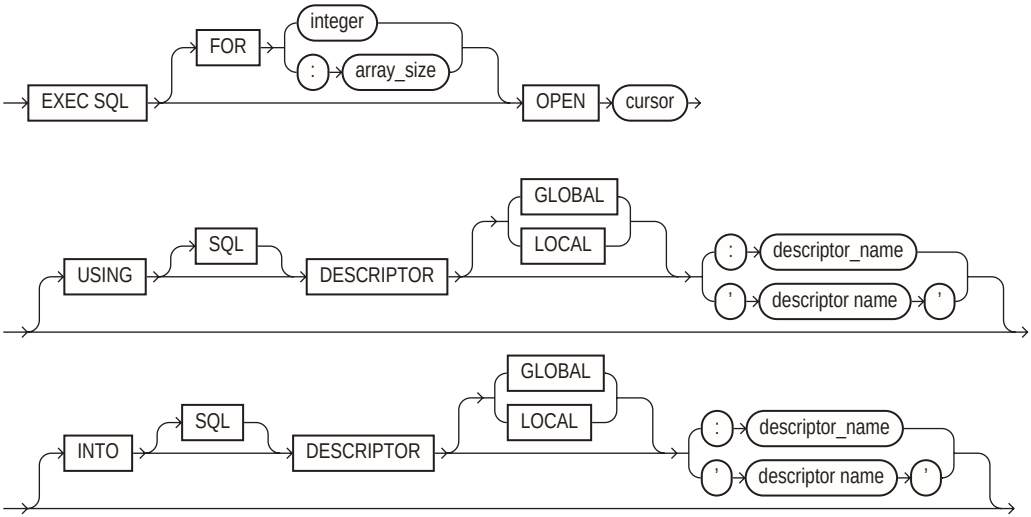
Purpose

To open a cursor (for ANSI Dynamic SQL Method 4), evaluating the associated query and substituting the input host variable names supplied by the USING clause into the WHERE clause of the query. The INTO clause denotes the output descriptor. It can be used in place of EXECUTE in dynamic SQL.

Prerequisites

You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.

Syntax



Keywords and Parameters

array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed. Used only with dynamic SQL when it is equivalent to EXECUTE.
cursor	The (previously declared) cursor to be opened.
USING DESCRIPTOR *descriptor_name* *'descriptor name'*	Specifies an ANSI input descriptor with the host variable containing the name of the ANSI descriptor, or the name of the ANSI descriptor.
INTO DESCRIPTOR *descriptor_name* *'descriptor name'*	Specifies an ANSI output descriptor with the host variable containing the name of the ANSI descriptor, or the name of the ANSI descriptor.
GLOBAL | LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

Usage Notes

Set the precompiler option DYNAMIC to ANSI.

The OPEN statement 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 statement does not actually retrieve rows; rows are retrieved by the FETCH statement.

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 statement.

You can reopen a cursor without first closing it. For more information on this statement, see "Inserting Rows" on page 5-9.

Example

```
01 DYN-STATEMENT PIC X(58) VALUE "SELECT ENAME, EMPNO FROM EMP WHERE
      DEPTNO =:DEPTNO-DAT".
01 DEPTNO-DAT PIC S9(9) COMP VALUE 10.
...
    EXEC SQL ALLOCATE DESCRIPTOR 'input-descriptor' END-EXEC.
    EXEC SQL ALLOCATE DESCRIPTOR 'output-descriptor'
    ...
    EXEC SQL PREPARE S FROM :DYN-STATEMENT END-EXEC.
    EXEC SQL DECLARE C CURSOR FOR S END-EXEC.
    ...
    EXEC SQL OPEN C USING DESCRIPTOR 'input-descriptor' END-EXEC.
    ...
```

Related Topics

CLOSE (Executable Embedded SQL) on page F-14.

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

FETCH DESCRIPTOR (Executable Embedded SQL) on page F-50.

PREPARE (Executable Embedded SQL) on page F-78.

PREPARE (Executable Embedded SQL)

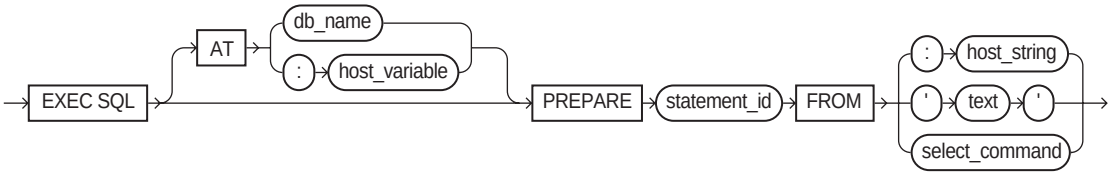
Purpose

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

db_name	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
host_variable	A host variable containing the name of the database connection.
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
statement_id	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	A host variable whose value is the text of a SQL statement or PL/SQL block to be prepared.
text	A text literal containing the SQL statement or PL/SQL block to be executed. The quotes may be omitted.
select_command	A SELECT statement.

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 statement (input host variables) or in the INTO clause of the FETCH statement (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

CLOSE (Executable Embedded SQL) on page F-14.

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

FETCH (Executable Embedded SQL) on page F-47.

OPEN (Executable Embedded SQL) on page F-75.

ROLLBACK (Executable Embedded SQL)

Purpose

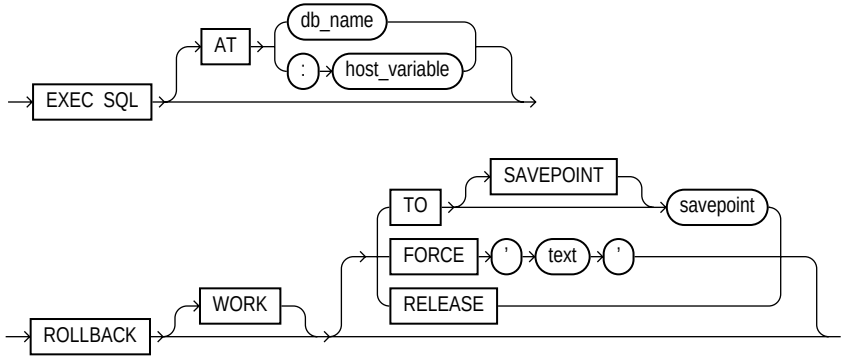
To undo work done in the current transaction. You can also use this statement to manually undo the work done by an in-doubt distributed transaction.

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

db_name	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
host_variable	A host variable containing the name of the database connection. If you omit this clause, the savepoint is created on your default database.
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 database server. The RELEASE clause is not allowed with SAVEPOINT and FORCE clauses.
savepoint	The name of the savepoint to be rolled back to.

Usage Notes

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle9*i* 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 Oracle9*i* issues an implicit COMMIT statement before and after processing any Data Definition Language statement.

Using the ROLLBACK statement 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 statement 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, Oracle9*i* 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

Oracle9*i* with the distributed option enables 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 ROLLBACK statement with the FORCE clause.

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) on page F-15.

SAVEPOINT (Executable Embedded SQL) on page F-82.

SAVEPOINT (Executable Embedded SQL)

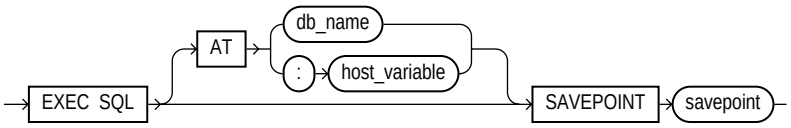
Purpose

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* — A database identifier declared in a previous DECLARE DATABASE statement.
- *host_variable* — 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 — The name of the savepoint to be created.

Usage Notes

For more information on this statement, see "Using the SAVEPOINT Statement" on page 3-17.

Example

This example illustrates the use of the embedded SQL SAVEPOINT statement:

```
EXEC SQL SAVEPOINT SAVE3 END-EXEC.
```

Related Topics

COMMIT (Executable Embedded SQL) on page F-15.

ROLLBACK (Executable Embedded SQL) on page F-79.

SELECT (Executable Embedded SQL)

Purpose

To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.

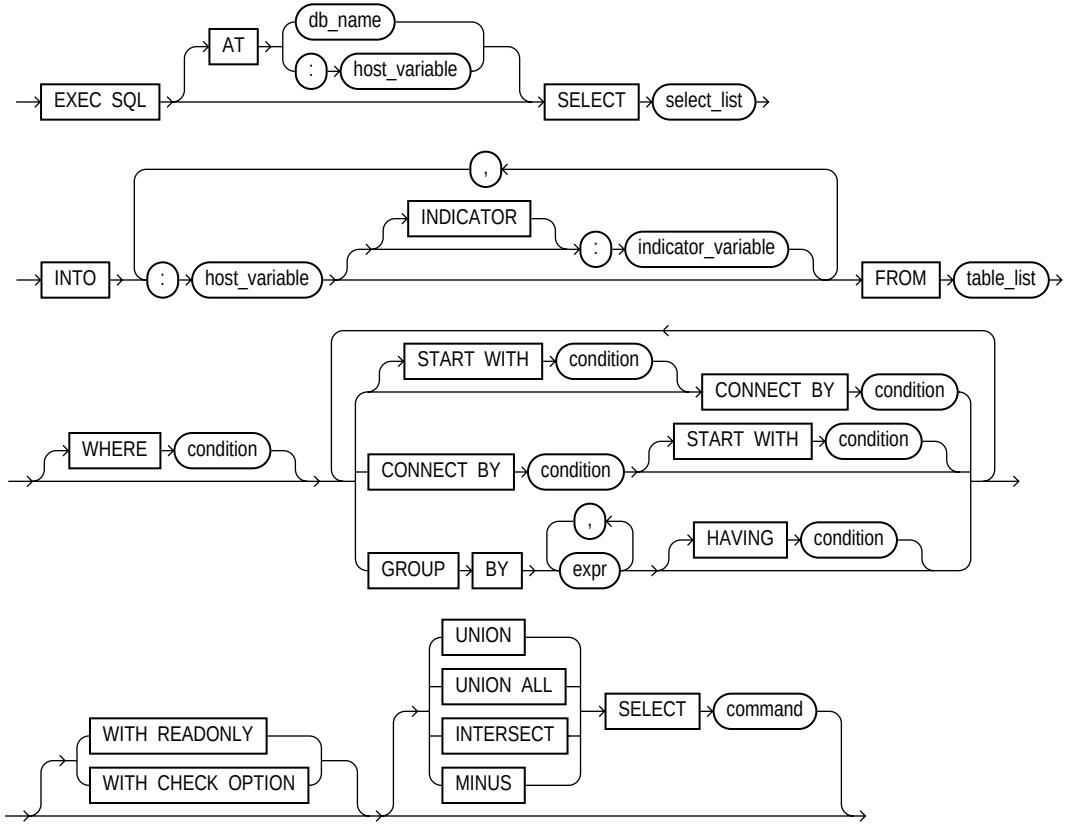
Prerequisites

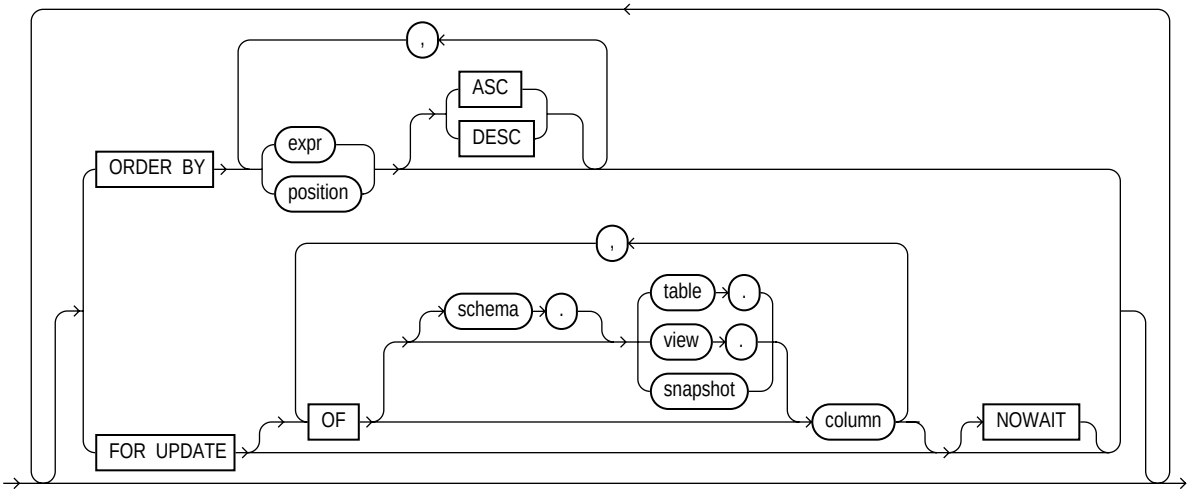
For you to select data from a table or snapshot, the table or snapshot must be in your own schema or you must have SELECT privilege on the table or snapshot.

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 enables you to select data from any table or any snapshot or any view's base table.

Syntax


EXEC SQL
AT
db_name
:
host_variable
SELECT
select_list
,
INTO
:
host_variable
INDICATOR
:
indicator_variable
FROM
table_list
WHERE
condition
START WITH
condition
CONNECT BY
condition
CONNECT BY
condition
START WITH
condition
GROUP
BY
,
expr
HAVING
condition
WITH READONLY
WITH CHECK OPTION
UNION
UNION ALL
INTERSECT
MINUS
SELECT
command



Keywords and Parameters

AT — Identifies the database to which the SELECT statement is issued. The database can be identified by either:

- *db_name* — A database identifier declared in a previous DECLARE DATABASE statement.
- *host_variable* — 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 statement 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 *Oracle9i SQL Reference*. The *condition* can contain host variables, but cannot contain indicator variables. These host variables can be either scalars or arrays.

All other keywords and parameters are identical to the non-embedded SQL SELECT statement.

Usage Notes

If no rows meet the WHERE clause condition, no rows are retrieved and Oracle9*i* 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 Oracle9*i* optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle9i Database Performance Guide and Reference.*

Example

This example illustrates the use of the embedded SQL SELECT statement:

```
EXEC SQL SELECT ENAME, SAL + 100, JOB
    INTO :ENAME, :SAL, :JOB
    FROM EMP
    WHERE EMPNO = :EMPNO
END-EXEC.
```

Related Topics

DECLARE CURSOR (Embedded SQL Directive) on page F-24.

DECLARE DATABASE (Oracle Embedded SQL Directive) on page F-26.

EXECUTE (Executable Embedded SQL) on page F-41.

FETCH (Executable Embedded SQL) on page F-47.

PREPARE (Executable Embedded SQL) on page F-78.

SET DESCRIPTOR (Executable Embedded SQL)

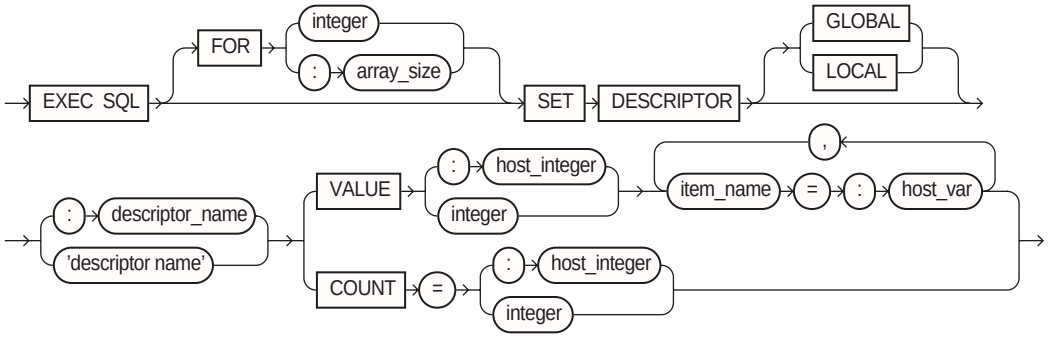
Purpose

Use this ANSI dynamic SQL statement to set information in the descriptor area from host variables.

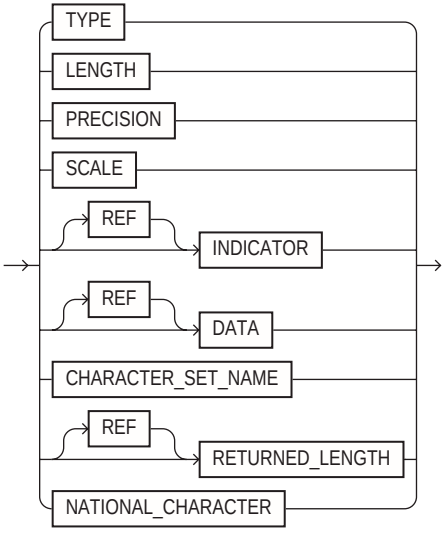
Prerequisites

Use after a DESCRIBE DESCRIPTOR.

Syntax



where `item_name` can be one of these choices:



Keywords and Parameters

array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed. The array size clause can only be used with DATA, RETURNED_LENGTH and INDICATOR item names.

GLOBAL | LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
descriptor_name	Host variable containing the name of the allocated ANSI descriptor.
'descriptor name'	Name of the allocated ANSI descriptor.
COUNT	The total number of input or output variables.
VALUE	The position of the referenced host variable in the statement.
item_name	See Table 10–6 on page 10-19, and Table 10–7 on page 10-19 for lists of the *item_names*, and their descriptions.
host_var	Host variable containing the total number of input or output variables.
integer	Total number of input or output variables.
host_var	The host variables used to set the item.
REF	Reference semantics are to be used. Can be used only with RETURNED_LENGTH, DATA, and INDICATOR item names. Must be used to set RETURNED_LENGTH.

Usage Notes

Use DYNAMIC=ANSI precompiler option. Set CHARACTER_SET_NAME to UTF16 for client-side Unicode support. See "SET DESCRIPTOR" on page 10-18 for complete details, including tables of descriptor item names.

Example

```
EXEC SQL SET DESCRIPTOR GLOBAL :mydescr COUNT = 3 END-EXEC.
```

Related Topics

ALLOCATE DESCRIPTOR (Executable Embedded SQL) on page F-12.

DEALLOCATE DESCRIPTOR (Embedded SQL Statement) on page F-23.

DESCRIBE DESCRIPTOR (Executable Embedded SQL) on page F-37.

GET DESCRIPTOR (Executable Embedded SQL) on page F-54.

PREPARE (Executable Embedded SQL) on page F-78.

UPDATE (Executable Embedded SQL)

Purpose

To change existing values in a table or in a view's base table.

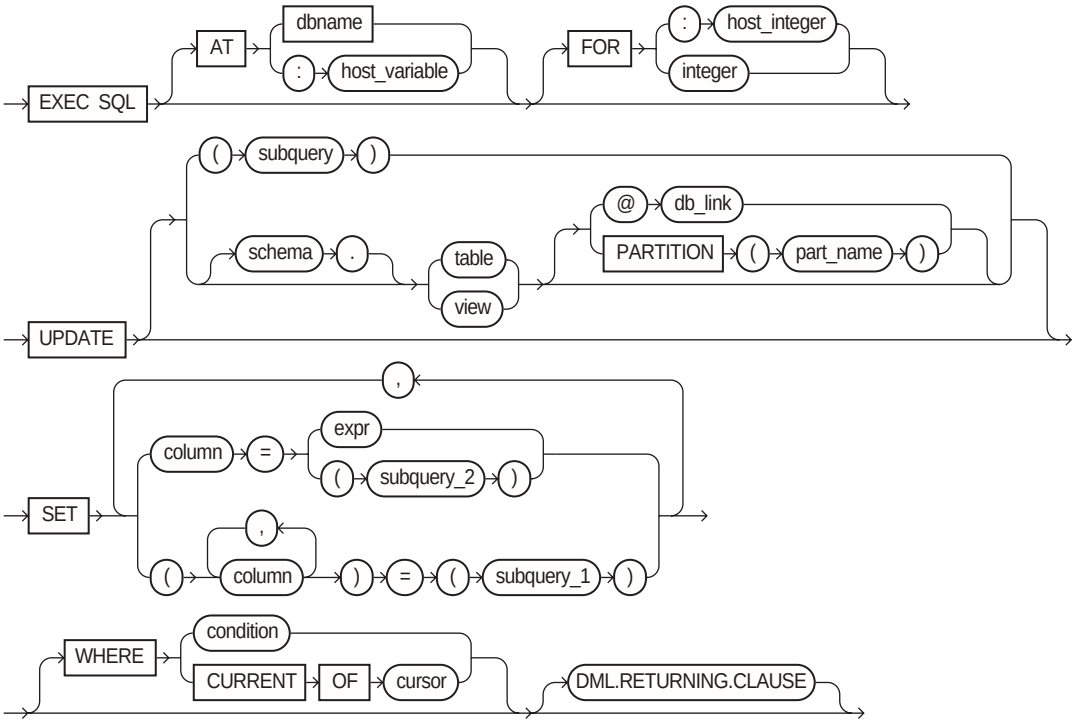
Prerequisites

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.

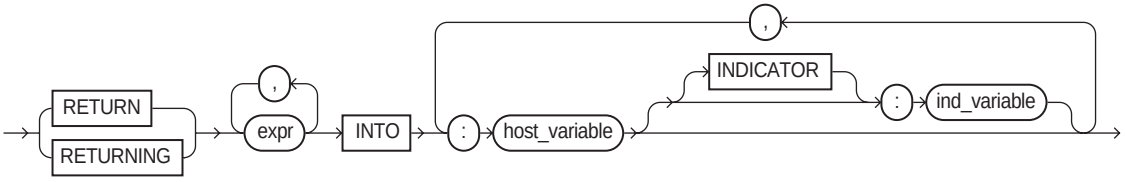
For you to update values in the base table of a view, the owner of the schema containing the view must have UPDATE privilege on the base table. Also, if the view is in a schema other than your own, you must have UPDATE privilege on the view.

The UPDATE ANY TABLE system privilege also enables you to update values in any table or any view's base table.

Syntax



where DML returning clause is:



Keywords and Parameters

AT identifies the database to which the UPDATE statement is issued. The database can be identified by either:

	dbname	A database identifier declared in a previous DECLARE DATABASE statement.
	host_variable	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, Oracle9*i* executes the statement once for each component of the smallest array.	
schema	The schema containing the table or view. If you omit *schema*, Oracle9*i* assumes the table or view is in your own schema.	
table view	The name of the table to be updated. If you specify *view*, Oracle9*i* updates the view's base table.	
dblink	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 *Oracle9i SQL Reference.* You can only use a database link to update a remote table or view if you are using Oracle9*i* with the distributed option.	
part_name	Name of partition in the table	
alias	A name used to reference the table, view, or subquery elsewhere in the statement.	
column	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	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 *Oracle9i SQL Reference.*	
subquery_1	A subquery that returns new values that are assigned to the corresponding columns. For the syntax of a subquery, see "SELECT" in the *Oracle9i SQL Reference.*	
subquery_2	A subquery that return a new value that is assigned to the corresponding column. For the syntax of a subquery, see "SELECT" in the *Oracle9i 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 *Oracle9i 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, Oracle9*i* updates all rows of the table or view.	
DML returning clause	See "DML Returning Clause" on page 5-9 for a discussion.	

Usage Notes

Host variables in the SET and WHERE clauses must be either all scalars or all arrays. If they are scalars, Oracle9*i* executes the UPDATE statement only once. If they are arrays, Oracle9*i* 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 Oracle9*i* 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 Oracle9*i* returns an error message through the SQLCODE element of the SQLCA. If you omit the WHERE clause, all rows are updated and Oracle9*i* 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 Oracle9*i* optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle9i Database Performance Guide and Reference.*

For more information on this statement, see "The Basic SQL Statements" on page 5-7 and Chapter 3, "Database Concepts".

Examples

The following examples illustrate the use of the embedded SQL UPDATE statement:

```
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) on page F-26.

VAR (Oracle Embedded SQL Directive)

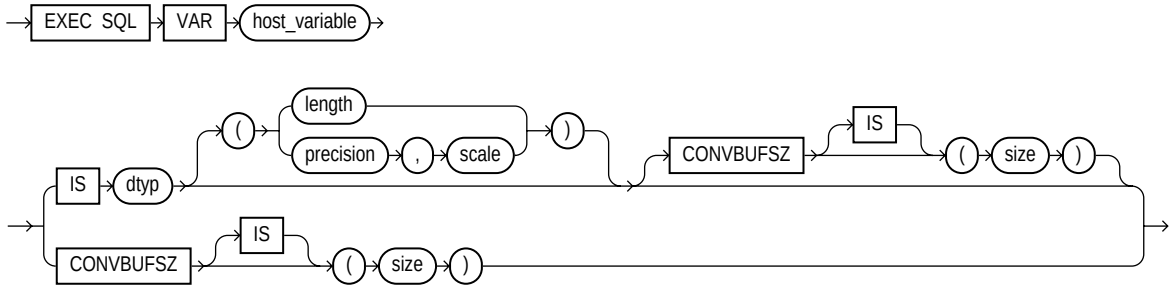
Purpose

To perform *host variable equivalencing*, to assign a specific Oracle9*i* 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	The host variable to be assigned an Oracle9*i* external datatype.
dtyp	An Oracle9*i* external datatype recognized by Pro*COBOL (not an Oracle9*i* 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 4-4.
size	The size in bytes of a buffer in the Oracle9*i* runtime library used to perform conversion between character sets of the *host_variable*.

Usage Notes

Datatype equivalencing is useful for any of the following purposes:

- To store program data as binary data in the database
- To override default datatype conversion

For more information about Oracle datatypes, see "Sample Program 4: Datatype Equivalencing" on page 4-51.

Example

This example equivalences the host variable DEPT_NAME to the datatype VARCHAR2 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 VARCHAR2 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)

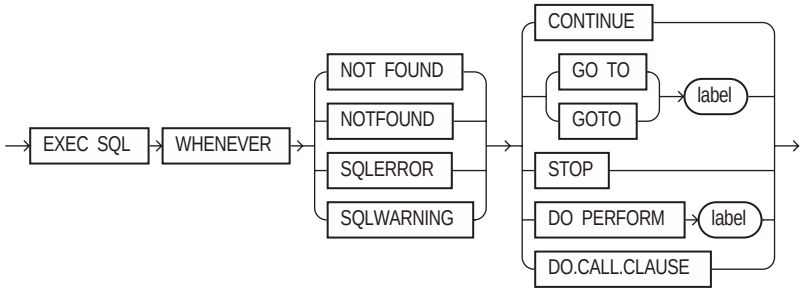
Purpose

To specify the action to be taken when an error or warning results from executing an embedded SQL program.

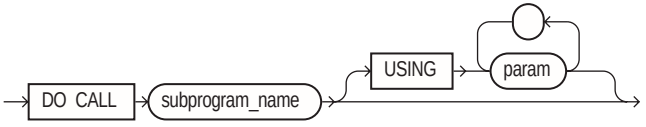
Prerequisites

None.

Syntax



where DO.CALL.CLAUSE is:



Keywords and Parameters

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 paragraph or section at *label*.
DO CALL	Indicates that the program should execute a subprogram.
subprogram_name	The subprogram to be executed. It may have to be inside quotes (").
USING	Indicates that the parameters of the subprogram follow.
param	A list of subprogram parameters separated by blanks.

The WHENEVER directive enables your program to take one of several possible actions 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 about and examples of the conditions and actions of this directive, see "WHENEVER Directive" on page 8-15.

Do not confuse the WHENEVER embedded SQL directive with the WHENEVER SQL*Plus command.

Example

The following example illustrates the use of the WHENEVER directive 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

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X