Oracle® Database

Programmer's Guide to the Oracle Precompilers

12*c* Release 1 (12.1)

E53283-01

May 2014

ORACLE®

Oracle Database Programmer's Guide to the Oracle Precompilers 12*c* Release 1 (12.1)

E53283-01

Copyright © 2008, 2014, Oracle and/or its affiliates. All rights reserved.

Primary Author: Simon Watt

Contributor: Radhakrishnan Hari, Nancy Ikeda, Ken Jacobs, Valarie Moore, Tim Smith, Scott Urman, Arun Desai, Mallikharjun Vemana, Subhranshu Banerjee

Contributor: The Database 12c documentation is dedicated to Mark Townsend, who was an inspiration to all who worked on this release.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface xxi

Intended Audience xxi
Documentation Accessibility xxi
Structure xxii
Related Documents xxiii
Conventions xxiv

1 Getting Acquainted

What Is an Oracle Precompiler? 1-1
Language Alternatives 1-2
Why Use an Oracle Precompiler? 1-2
Why Use SQL? 1-3
Why Use PL/SQL? 1-3
What Do the Oracle Precompilers Offer? 1-3
Do the Oracle Precompilers Meet Industry Standards? 1-5
Requirements 1-5
Compliance 1-5
FIPS Flagger 1-6
FIPS Option 1-6
Certification 1-6

2 Learning the Basics

Key Concepts of Embedded SQL Programming 2-1
Embedded SQL Statements 2-2
Executable versus Declarative Statements 2-2
Embedded SQL Syntax 2-4
Static versus Dynamic SQL Statements 2-4
Embedded PL/SQL Blocks 2-4
Host and Indicator Variables 2-4
Oracle Datatypes 2-5
Arrays 2-5
Datatype Equivalencing 2-5
Private SQL Areas, Cursors, and Active Sets 2-5
Transactions 2-6
Errors and Warnings 2-6

Steps in Developing an Embedded SQL Application 2-7
A Sample Program 2-7
Sample Tables 2-8
Sample Data 2-9

3 Meeting Program Requirements

The Declare Section 3-1
An Example 3-2
INCLUDE Statements 3-2
The SQLCA 3-3
Oracle Datatypes 3-3
Internal Datatypes 3-4
CHAR 3-4
DATE 3-5
LONG 3-5
LONG RAW 3-5
MLSLABEL 3-5
NUMBER 3-5
RAW 3-6
ROWID 3-6
VARCHAR2 3-6
SQL Pseudocolumns and Functions 3-6
ROWLABEL Column 3-8
External Datatypes 3-8
CHAR 3-9
CHARF 3-9
CHARZ 3-10
DATE 3-10
DECIMAL 3-10
DISPLAY 3-10
FLOAT 3-10
INTEGER 3-11
LONG 3-11
LONG RAW 3-11
LONG VARCHAR 3-11
LONG VARRAW 3-11
MLSLABEL 3-11
NUMBER 3-12
RAW 3-12
ROWID 3-12
STRING 3-13
UNSIGNED 3-13
VARCHAR 3-13
VARCHAR2 3-13
VARNUM 3-14
VARRAW 3-14
Datatype Conversion 3-14

DATE Values 3-16
RAW and LONG RAW Values 3-16
Declaring and Referencing Host Variables 3-16
Some Examples 3-17
VARCHAR Variables 3-17
Host Variable Guidelines 3-18
Declaring and Referencing Indicator Variables 3-18
INDICATOR Keyword 3-18
An Example 3-18
Indicator Variable Guidelines 3-19
Datatype Equivalencing 3-19
Why Equivalence Datatypes? 3-19
Host Variable Equivalencing 3-20
An Example 3-20
Using the CHARF Datatype Specifier 3-22
Guidelines 3-22
Globalization Support 3-22
Multibyte Globalization Support Character Sets 3-24
Character Strings in Embedded SQL 3-24
Dynamic SQL 3-25
Embedded DDL 3-25
Multibyteultibyte Globalization Support Host Variables 3-25
Restrictions 3-25
Blank Padding 3-25
Indicator Variables 3-26
Concurrent Logons 3-26
Some Preliminaries 3-27
Default Databases and Connections 3-27
Explicit Logons 3-27
Single Explicit Logons 3-28
Multiple Explicit Logons 3-30
Implicit Logons 3-31
Single Implicit Logons 3-32
Multiple Implicit Logons 3-32
Embedding OCI (Oracle Call Interface) Calls 3-33
Setting Up the LDA 3-33
Remote and Multiple Connections 3-33
Developing X/Open Applications 3-34
Oracle-Specific Issues 3-34
Connecting to Oracle 3-35
Transaction Control 3-35
OCI Calls 3-35
Linking 3-35

4 Using Embedded SQL

Using Host Variables 4-1
Output versus Input Host Variables 4-1

Using Indicator Variables 4-2
Input Variables 4-2
Output Variables 4-2
Inserting Nulls 4-3
Handling Returned Nulls 4-3
Fetching Nulls 4-3
Testing for Nulls 4-4
Fetching Truncated Values 4-4
The Basic SQL Statements 4-4
Selecting Rows 4-5
Available Clauses 4-6
Inserting Rows 4-6
Using Subqueries 4-6
Updating Rows 4-6
Deleting Rows 4-7
Using the WHERE Clause 4-7
Cursors 4-7
Declaring a Cursor 4-8
Opening a Cursor 4-8
Fetching from a Cursor 4-9
Closing a Cursor 4-9
Using the CURRENT OF Clause 4-10
Restrictions 4-10
A Typical Sequence of Statements 4-10
A Complete Example 4-11
Cursor Variables 4-12
Declaring a Cursor Variable 4-13
Allocating a Cursor Variable 4-13
Opening a Cursor Variable 4-13
Fetching from a Cursor Variable 4-14
Closing a Cursor Variable 4-14

5 Using Embedded PL/SQL

Advantages of PL/SQL 5-1
Better Performance 5-1
Integration with Oracle 5-2
Cursor FOR Loops 5-2
Subprograms 5-2
Parameter Modes 5-3
Packages 5-3
PL/SQL Tables 5-3
User-defined Records 5-4
Embedding PL/SQL Blocks 5-5
Using Host Variables 5-5
An Example 5-5
A More Complex Example 5-6
VARCHAR Pseudotype 5-8

Using Indicator Variables 5-8
Handling Nulls 5-9
Handling Truncated Values 5-9
Using Host Arrays 5-9
ARRAYLEN Statement........ 5-11
Using Cursors........ 5-12
An Alternative 5-13
Stored Subprograms 5-13
Creating Stored Subprograms........ 5-13
Calling a Stored Subprogram 5-15
Remote Access 5-17
Getting Information about Stored Subprograms........ 5-17
Using Dynamic PL/SQL 5-17
Restriction........ 5-17

6 Running the Oracle Precompilers

The Precompiler Command 6-1
What Occurs during Precompilation? 6-2
Precompiler Options........ 6-2
Default Values 6-3
Determining Current Values 6-3
Case Sensitivity........ 6-4
Configuration Files 6-4
Entering Options 6-5
On the Command Line........ 6-5
Inline 6-5
Advantages 6-5
Scope of EXEC ORACLE........ 6-5
From a Configuration File........ 6-6
Advantages 6-6
Using Configuration Files........ 6-6
Setting Option Values........ 6-7
Scope of Options 6-7
Quick Reference 6-7
Using the Precompiler Options 6-10
ASACC........ 6-11
ASSUME_SQLCODE........ 6-11
AUTO_CONNECT 6-11
CHAR_MAP 6-12
CINCR........ 6-12
CLOSE_ON_COMMIT........ 6-13
CMAX 6-13
CMIN 6-14
CNOWAIT 6-14
CODE 6-14
COMMON_NAME........ 6-15
COMMON_PARSER 6-16

COMP_CHARSET 6-16
COMP_CHARSET 6-17
CONFIG 6-18
CPOOL 6-18
CPP_SUFFIX 6-18
CTIMEOUT 6-19
DB2_ARRAY 6-19
DBMS 6-20
DEF_SQLCODE 6-21
DEFINE 6-21
DURATION 6-22
DYNAMIC 6-22
ERRORS 6-23
ERRTYPE 6-23
EVENTS 6-23
FIPS 6-24
FORMAT 6-24
Globalization Support_LOCAL 6-25
HEADER 6-25
HOLD_CURSOR 6-26
HOST 6-27
IMPLICIT_SVPT 6-27
INAME 6-27
INCLUDE 6-28
IRECLEN 6-29
INTYPE 6-29
LINES 6-29
LITDELIM 6-30
LNAME 6-31
LRECLEN 6-31
LTYPE 6-31
MAXLITERAL 6-32
MAXOPENCURSORS 6-32
MAX_ROW_INSERT 6-33
MODE 6-33
MULTISUBPROG 6-34
NATIVE_TYPES 6-35
NLS_CHAR 6-35
NLS_LOCAL 6-36
OBJECTS 6-36
ONAME 6-36
ORACA 6-37
ORECLEN 6-37
OUTLINE 6-38
OUTLNPREFIX 6-38
PAGELEN 6-39
PARSE 6-39

PREFETCH 6-40
RELEASE_CURSOR 6-40
RUNOUTLINE 6-41
SELECT_ERROR 6-41
SQLCHECK 6-42
STMT_CACHE 6-43
THREADS 6-43
TYPE_CODE 6-44
UNSAFE_NULL 6-44
USERID 6-45
UTF16_CHARSET 6-45
VARCHAR 6-46
VERSION 6-46
XREF 6-47
Conditional Precompilations 6-47
An Example 6-47
Defining Symbols 6-48
Separate Precompilations 6-48
Guidelines 6-48
Restrictions 6-49
Compiling and Linking 6-49
System-Dependent 6-49
Multibyte Globalization Support Compatibility 6-49

7 Defining and Controlling Transactions

Some Terms You Should Know 7-1
How Transactions Guard Your Database 7-2
How to Begin and End Transactions 7-2
Using the COMMIT Statement 7-3
Using the ROLLBACK Statement 7-3
Statement-Level Rollbacks 7-4
Using the SAVEPOINT Statement 7-5
Using the RELEASE Option 7-6
Using the SET TRANSACTION Statement 7-6
Overriding Default Locking 7-7
Using the FOR UPDATE OF Clause 7-7
Restrictions 7-8
Using the LOCK TABLE Statement 7-8
Fetching Across Commits 7-8
Handling Distributed Transactions 7-9
Guidelines 7-9
Designing Applications 7-9
Obtaining Locks 7-10
Using PL/SQL 7-10

8 Error Handling and Diagnostics

The Need for Error Handling ... 8-1
Error Handling Alternatives ... 8-1
SQLCODE and SQLSTATE ... 8-2
SQLCA ... 8-2
ORACA ... 8-3
Using Status Variables when MODE={ANSI | ANSI14} ... 8-3
Some Historical Information ... 8-3
Release 1.5 ... 8-3
Release 1.6 ... 8-3
Release 1.7 ... 8-4
Declaring Status Variables ... 8-4
Declaring SQLCODE ... 8-4
Declaring SQLSTATE ... 8-5
Status Variable Combinations ... 8-5
Status Variable Values ... 8-8
SQLCODE Values ... 8-8
SQLSTATE Values ... 8-9
Using the SQL Communications Area ... 8-15
Declaring the SQLCA ... 8-16
Declaring the SQLCA in Pro*COBOL ... 8-16
Declaring the SQLCA in Pro*FORTRAN ... 8-16
What's in the SQLCA? ... 8-16
Key Components of Error Reporting ... 8-17
Status Codes ... 8-17
Warning Flags ... 8-17
Rows-Processed Count ... 8-17
Parse Error Offset ... 8-17
Error Message Text ... 8-18
SQLCA Structure ... 8-18
SQLCAID ... 8-18
SQLCABC ... 8-18
SQLCODE ... 8-18
SQLERRM ... 8-19
SQLERRP ... 8-19
SQLERRD ... 8-19
SQLWARN ... 8-20
SQLEXT ... 8-21
PL/SQL Considerations ... 8-21
Getting the Full Text of Error Messages ... 8-21
Using the WHENEVER Statement ... 8-22
SQLWARNING ... 8-22
SQLERROR ... 8-22
NOT FOUND ... 8-22
CONTINUE ... 8-23
DO ... 8-23
GOTO ... 8-23

STOP 8-23
Some Examples 8-23
Scope 8-24
Guidelines 8-24
Getting the Text of SQL Statements 8-26
Using the Oracle Communications Area 8-28
Declaring the ORACA 8-29
Enabling the ORACA 8-29
What's in the ORACA? 8-29
Choosing Run-time Options 8-30
ORACA Structure 8-30
ORACAID 8-30
ORACABC 8-30
ORACCHF 8-30
ORADBGF 8-31
ORAHCHF 8-31
ORASTXTF 8-31
Diagnostics 8-31
ORASTXT 8-32
ORASFNM 8-32
ORASLNR 8-32
Cursor Cache Statistics 8-32
ORAHOC 8-32
ORAMOC 8-32
ORACOC 8-32
ORANOR 8-33
ORANPR 8-33
ORANEX 8-33
An Example 8-33

9 Using Host Arrays

What Is a Host Array? 9-1
Why Use Arrays? 9-1
Declaring Host Arrays 9-2
Dimensioning Arrays 9-2
Restrictions 9-2
Using Arrays in SQL Statements 9-2
Selecting into Arrays 9-3
Batch Fetches 9-3
Number of Rows Fetched 9-3
Restrictions 9-4
Fetching Nulls 9-4
Fetching Truncated Values 9-5
Inserting with Arrays 9-5
Updating with Arrays 9-5
Deleting with Arrays 9-6
Restrictions 9-7

Using Indicator Arrays 9-7
Using the FOR Clause 9-7
Restrictions 9-8
In a SELECT Statement 9-8
With the CURRENT OF Clause 9-8
Using the WHERE Clause 9-9
Mimicking the CURRENT OF Clause 9-9
Using SQLERRD(3) 9-10

10 Using Dynamic SQL

What Is Dynamic SQL? 10-1
Advantages and Disadvantages of Dynamic SQL 10-2
When to Use Dynamic SQL 10-2
Requirements for Dynamic SQL Statements 10-2
How Dynamic SQL Statements Are Processed 10-3
Methods for Using Dynamic SQL 10-3
Method 1 10-3
Method 2 10-4
Method 3 10-4
Method 4 10-4
Guidelines 10-4
Avoiding Common Errors 10-5
Using Method 1 10-6
The EXECUTE IMMEDIATE Statement 10-6
An Example 10-6
Using Method 2 10-7
The USING Clause 10-8
An Example 10-8
Using Method 3 10-9
PREPARE 10-9
DECLARE 10-10
OPEN 10-10
FETCH 10-10
CLOSE 10-10
An Example 10-11
Using Method 4 10-11
Need for the SQLDA 10-12
The DESCRIBE Statement 10-12
What Is a SQLDA? 10-12
Implementing Method 4 10-13
Using the DECLARE STATEMENT Statement 10-14
Usage of Host Arrays 10-14
Using PL/SQL 10-15
With Method 1 10-15
With Method 2 10-15
With Method 3 10-15
With Method 4 10-15

Caution 10-16

11 Writing User Exits

What Is a User Exit? 11-2
Why Write a User Exit? 11-2
Developing a User Exit 11-3
Writing a User Exit 11-3
Requirements for Variables 11-3
The IAF GET Statement 11-4
The IAF PUT Statement 11-4
Calling a User Exit 11-5
Passing Parameters to a User Exit 11-5
Returning Values to a Form 11-6
The IAP Constants 11-6
Using the SQLIEM Function 11-6
Using WHENEVER 11-7
An Example 11-7
Precompiling and Compiling a User Exit 11-7
Using the GENXTB Utility 11-7
Linking a User Exit into SQL*Forms 11-8
Guidelines for SQL*Forms User Exits 11-8
Naming the Exit 11-8
Connecting to Oracle 11-8
Issuing I/O Calls 11-9
Using Host Variables 11-9
Updating Tables 11-9
Issuing Commands 11-9
EXEC TOOLS Statements 11-9
EXEC TOOLS SET 11-10
EXEC TOOLS GET 11-10
EXEC TOOLS SET CONTEXT 11-10
EXEC TOOLS GET CONTEXT 11-11
EXEC TOOLS MESSAGE 11-11

A New Features

Fetching NULLs without Using Indicator Variables A-1
Using DBMS=V7 and MODE=ORACLE A-1
Related Error Messages A-1
Additional Array Insert/Select Syntax A-2
SQL99 Syntax Support A-2
Fixing Execution Plans A-2
Using Implicit Buffered Insert A-3
Dynamic SQL Statement Caching A-3
Scrollable Cursors A-6
Platform Endianness Support A-6
Flexible B Area Length A-6

B Oracle Reserved Words, Keywords, and Namespaces

Oracle Reserved Words B-1
Oracle Keywords B-2
PL/SQL Reserved Words B-3
Oracle Reserved Namespaces B-5

C Performance Tuning

What Causes Poor Performance? C-1
How Can Performance be Improved? C-2
Using Host Arrays C-2
Using Embedded PL/SQL C-2
Optimizing SQL Statements C-3
  Optimizer Hints C-3
  Giving Hints C-4
  Trace Facility C-4
Using Indexes C-4
Taking Advantage of Row-Level Locking C-4
Eliminating Unnecessary Parsing C-5
  Handling Explicit Cursors C-5
  Cursor Control C-5
  Using the Cursor Management Options C-6
  Private SQL Areas and Cursor Cache C-6
  Resource Use C-7
  Infrequent Execution C-7
  Frequent Execution C-8
  Parameter Interactions C-8

D Syntactic and Semantic Checking

What Is Syntactic and Semantic Checking? D-1
Controlling the Type and Extent of Checking D-1
Specifying SQLCHECK=SEMANTICS D-2
  Enabling a Semantic Check D-2
  Connecting to Oracle D-2
  Using DECLARE TABLE D-3

E Embedded SQL Commands and Directives

Summary of Precompiler Directives and Embedded SQL Commands E-2
About The Command Descriptions E-3
How to Read Syntax Diagrams E-3
  Required Keywords and Parameters E-4
  Optional Keywords and Parameters E-4
  Syntax Loops E-5
  Multi-part Diagrams E-5
  Database Objects E-5
ALLOCATE (Executable Embedded SQL Extension) E-5
  Purpose E-5

Prerequisites E-6
Syntax E-6
Keywords and Parameters E-6
Usage Notes E-6
Related Topics E-6
CLOSE (Executable Embedded SQL) E-6
Purpose E-6
Prerequisites E-6
Syntax E-7
Keywords and Parameters E-7
Usage Notes E-7
Example E-7
Related Topics E-7
COMMIT (Executable Embedded SQL) E-7
Purpose E-7
Prerequisites E-7
Syntax E-8
Keyword and Parameters E-8
Usage Notes E-8
Related Topics E-9
CONNECT (Executable Embedded SQL Extension) E-9
Purpose E-9
Prerequisites E-9
Syntax E-9
Keyword and Parameters E-9
Usage Notes E-10
Related Topics E-10
DECLARE CURSOR (Embedded SQL Directive) E-10
Purpose E-10
Prerequisites E-10
Syntax E-10
Keywords and Parameters E-10
Usage Notes E-11
Example E-11
Related Topics E-11
DECLARE DATABASE (Oracle Embedded SQL Directive) E-11
Purpose E-11
Prerequisites E-12
Syntax E-12
Keywords and Parameters E-12
Usage Notes E-12
Example E-12
Related Topics E-12
DECLARE STATEMENT (Embedded SQL Directive) E-12
Purpose E-12
Prerequisites E-12
Syntax E-13

Keywords and Parameters E-13
Usage Notes E-13
Example I E-13
Example II E-13
Related Topics E-14
DECLARE TABLE (Oracle Embedded SQL Directive) E-14
Purpose E-14
Prerequisites E-14
Syntax E-14
Keywords and Parameters E-14
Usage Notes E-15
Example E-15
Related Topics E-15
DELETE (Executable Embedded SQL) E-15
Purpose E-15
Prerequisites E-15
Syntax E-16
Keywords and Parameters E-16
Usage Notes E-17
Example E-17
Related Topics E-17
DESCRIBE (Executable Embedded SQL) E-17
Purpose E-18
Prerequisites E-18
Syntax E-18
Keywords and Parameters E-18
Usage Notes E-18
Example E-18
Related Topics E-19
EXECUTE ... END-EXEC (Executable Embedded SQL Extension) E-19
Purpose E-19
Prerequisites E-19
Syntax E-19
Keywords and Parameters E-19
Usage Notes E-19
Example E-20
Related Topics E-20
EXECUTE (Executable Embedded SQL) E-20
Purpose E-20
Prerequisites E-20
Syntax E-20
Keywords and Parameters E-20
Usage Notes E-21
Example E-21
Related Topics E-21
EXECUTE IMMEDIATE (Executable Embedded SQL) E-21
Purpose E-21

Prerequisites....... E-21
Syntax....... E-21
Keywords and Parameters....... E-21
Usage Notes E-22
Example E-22
Related Topics....... E-22
FETCH (Executable Embedded SQL) E-22
Purpose E-22
Prerequisites....... E-22
Syntax....... E-22
Keywords and Parameters....... E-22
Usage Notes E-23
Example E-24
Related Topics....... E-24
INSERT (Executable Embedded SQL) E-24
Purpose E-24
Prerequisites....... E-24
Syntax....... E-25
Keywords and Parameters....... E-25
Usage Notes E-26
Example I....... E-26
Example II E-26
Related Topics....... E-26
OPEN (Executable Embedded SQL)....... E-26
Purpose E-26
Prerequisites....... E-26
Syntax....... E-27
Keywords and Parameters....... E-27
Usage Notes E-27
Example E-27
Related Topics....... E-28
PREPARE (Executable Embedded SQL) E-28
Purpose E-28
Prerequisites....... E-28
Syntax....... E-28
Keywords and Parameters....... E-28
Usage Notes E-28
Example E-28
Related Topics....... E-29
ROLLBACK (Executable Embedded SQL)....... E-29
Purpose E-29
Prerequisites....... E-29
Syntax....... E-29
Keywords and Parameters....... E-29
Usage Notes E-30
Example I....... E-30
Example II E-30

Distributed Transactions E-30
Example III E-31
Related Topics E-31
SAVEPOINT (Executable Embedded SQL) E-31
Purpose E-31
Prerequisites E-31
Syntax E-31
Keywords and Parameters E-31
Usage Notes E-32
Related Topics E-32
SELECT (Executable Embedded SQL) E-32
Purpose E-32
Prerequisites E-32
Syntax E-33
Keywords and Parameters E-33
Usage Notes E-34
Example E-34
Related Topics E-34
UPDATE (Executable Embedded SQL) E-34
Purpose E-34
Prerequisites E-34
Syntax E-35
Keywords and Parameters E-35
Usage Notes E-36
Examples E-36
Related Topics E-37
VAR (Oracle Embedded SQL Directive) E-37
Purpose E-37
Prerequisites E-37
Syntax E-37
Keywords and Parameters E-37
Usage Notes E-37
Example E-37
Related Topics E-38
WHENEVER (Embedded SQL Directive) E-38
Purpose E-38
Prerequisites E-38
Syntax E-38
Keywords and Parameters E-38
Usage Notes E-39
Example E-39
Related Topics E-39

Index

List of Figures

1–1 Embedded SQL Program Development 1-2
1–2 Features and Benefits 1-4
2–1 Application Development Process 2-7
3–1 Updating the SQLCA 3-3
3–2 Connecting through SQL*Net 3-26
3–3 Hypothetical DTP Model 3-34
5–1 Maximum Cursors in Use 5-12
8–1 SQLSTATE Coding Scheme 8-9
8–2 SQLCA Variables 8-17
8–3 ORACA Variables 8-30
10–1 Choosing the Right Method 10-5
11–1 SQL*Forms 11-2
C–1 PL/SQL Boosts Performance C-3
C–2 Cursors Linked through the Cursor Cache C-7

List of Tables

2–1 Embedded SQL Statements 2-2
2–2 Executable SQL Statements and their Descriptions 2-3
3–1 Column and Pseudo Column Datatypes 3-4
3–2 Pseudo Column Datatypes 3-6
3–3 Parameterless Function Datatypes 3-7
3–4 External Datatypes 3-8
3–5 DATE Datatype Example 3-10
3–6 Conversion Between Internal and External Datatypes 3-15
3–7 External Datatype Parameters 3-21
3–8 Examples of VARNUM Values Returned 3-22
3–9 Globalization Support Parameters 3-23
5–1 Legal Conversions: PL/SQL Table Row and Host Array Elements 5-10
6–1 Precompiler Run Commands 6-1
6–2 System Configuration Files 6-4
6–3 Precompiler Options Quick Reference 6-7
6–4 Compatible DBMS and MODE Settings 6-20
6–5 Input File Extensions 6-28
6–6 SQLCHECK Checking 6-43
8–1 SQLCODE Declarations 8-4
8–2 SQLSTATE Declarations 8-5
8–3 Status Variable Combinations - SQLCODE = NO 8-5
8–4 Status Variable Combinations - SQLCODE = YES 8-7
8–5 Predefined SQL92 Classes 8-9
8–6 Oracle Error Mapping to SQLSTATE Status 8-10
8–7 SQLGLS Parameter Datatypes 8-27
8–8 SQL Command Function Codes 8-27
9–1 Valid Host Arrays for SELECT INTO 9-4
9–2 Valid Host Arrays for UPDATE 9-6
10–1 Dynamic SQL Method Applicability 10-3
B–1 Oracle Reserved Namespaces B-5
C–1 HOLD_CURSOR RELEASE_CURSOR Interactions C-8
E–1 Summary of Embedded SQL Commands and Directives E-2

Preface

This chapter contains the following:

- Intended Audience
- Documentation Accessibility
- Structure
- Related Documents
- Conventions

This manual is a comprehensive user's guide and reference to the Oracle Pro*COBOL and Pro*FORTRAN Precompilers. It shows you step-by-step how to develop applications that use the SQL to access and manipulate data. It explores underlying concepts to advanced programming techniques using clear examples.

Intended Audience

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

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

- Applications programming in a high-level language
- The SQL database language
- Oracle concepts and terminology

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Structure

This guide contains eleven chapters and five appendixes. Chapters 1 and 2 give you your bearings, then Chapters 3, 4, 5, and 6 lead you through the essentials of embedded SQL programming. After reading these chapters, you will be able to write and run useful embedded SQL applications. Chapters 7, 8, 9, 10, and 11 cover advanced topics. A brief summary of what you will find in each chapter and appendix follows.

This sample manual contains one part, two chapters, and one appendixes. (Insert this chapter, appendix, and parts as cross-references so that the links are apparent in HTML.)

Chapter 1, "Getting Acquainted"

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

Chapter 2, "Learning the Basics"

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

Chapter 3, "Meeting Program Requirements"

This chapter shows you how to meet embedded SQL program requirements. You learn the embedded SQL commands that declare variables, declare communications areas, and connect to an Oracle database. You also learn about the Oracle datatypes, Globalization Support (Globalization Support), data conversion, and how to take advantage of datatype equivalencing. In addition, this chapter shows you how to embed Oracle Call Interface (OCI) calls in your program and how to develop X/Open applications.

Chapter 4, " Using Embedded SQL"

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

Chapter 5, "Using Embedded PL/SQL"

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

Chapter 6, "Running the Oracle Precompilers"

This chapter details the requirements for running an Oracle Precompiler. You learn what happens during precompilation, how to issue the precompiler command, how to specify the many useful precompiler options, how to do conditional and separate precompilations, and how to embed OCI calls in your host program.

Chapter 7, "Defining and Controlling Transactions"

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

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, "Using Host Arrays"

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

Chapter 10, "Using Dynamic SQL"

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

Chapter 11, "Writing User Exits"

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

Appendix A, "New Features"

This appendix highlights the improvements and new features introduced with Release 1.8 of the Oracle Precompilers.

Appendix B, "Oracle Reserved Words, Keywords, and Namespaces"

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

Appendix C, "Performance Tuning"

This appendix gives you some simple, easy-to-apply methods for improving the performance of your applications.

Appendix D, "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 E, "Embedded SQL Commands and Directives"

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

Related Documents

For more information on Programmer's Guide to the Oracle Precompilers, refer to the Oracle Technology Network (OTN):

`http://www.oracle.com/technology/documentation/index.html`

Conventions

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.

1

Getting Acquainted

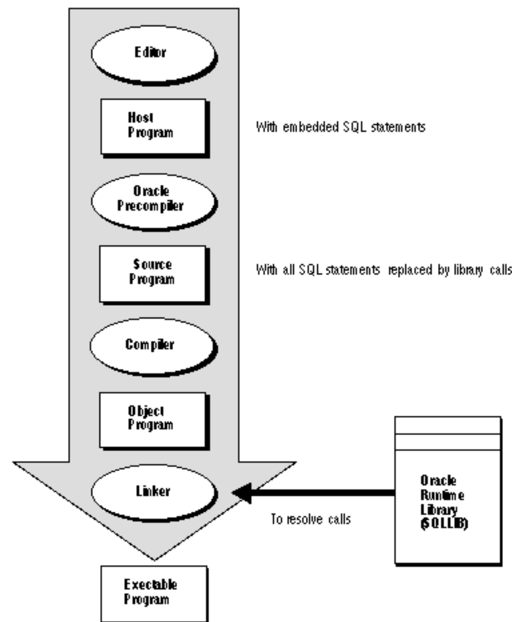
This chapter introduces you to the Oracle Precompilers. You look at their role in developing application programs that manipulate Oracle data and find out what they allow your applications to do. The following questions are answered:

- What Is an Oracle Precompiler?
- Why Use an Oracle Precompiler?
- Why Use SQL?
- Why Use PL/SQL?
- What Do the Oracle Precompilers Offer?
- Do the Oracle Precompilers Meet Industry Standards?

What Is an Oracle Precompiler?

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

Figure 1–1 Embedded SQL Program Development



Language Alternatives

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

- C/C++
- COBOL

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

Note: This guide is supplemented by companion books devoted to using precompilers with C/C++ and COBOL.

Pro*FORTRAN and SQL*Module for Ada are in "maintenance mode," which means that Version 1 of these products will not be enhanced with any additional features beyond those included with Release 1.6. However, Oracle will continue to issue patch releases as bugs are reported and corrected.

Why Use an Oracle Precompiler?

The Oracle Precompilers let you include the flexibility of SQL into your application programs. You can use SQL in popular high-level languages such as C and COBOL. A convenient, easy to use interface lets your application access Oracle directly.

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

Furthermore, with the Oracle Precompilers you can fine-tune your applications. They allow close monitoring of resource usage, SQL statement execution, and various

run-time indicators. With this information, you can adjust program parameters for maximum performance.

Why Use SQL?

If you want to access and manipulate Oracle data, you need SQL. Whether you use SQL interactively or embedded in an application program depends on the job at hand. If the job requires the procedural processing power of C or 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 nonprocedural, it lets you specify what you want done without specifying how to do it. A few English-like statements make it easy to manipulate Oracle data one row or many rows at a time.

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

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

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

Why Use PL/SQL?

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

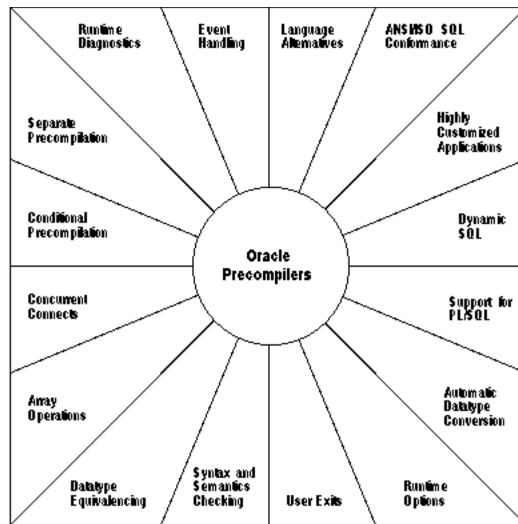
The main advantage of embedded PL/SQL is better performance. Unlike SQL, PL/SQL enables group SQL statements logically and send them to Oracle in a block rather than one by one. This reduces network traffic and processing overhead.

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

What Do the Oracle Precompilers Offer?

As Figure 1–2 shows, the Oracle Precompilers offer many features and benefits that help you to develop effective, reliable applications.

Figure 1–2 Features and Benefits



For example, the Oracle Precompilers allow you to

- program your application in any of six high-level languages
- confirm to the ANSI/ISO embedded SQL standard
- take advantage of dynamic SQL, an advanced programming technique that lets your program accept or build any valid SQL statement at run time
- design and develop highly customized applications
- convert automatically between Oracle internal datatypes and high-level language datatypes
- improve performance by embedding PL/SQL transaction processing blocks in your application program
- specify useful precompiler options and change their values during precompilation
- use datatype equivalencing to control the way Oracle interprets input data and formats output data
- precompile several program modules separately, then link them into one executable program
- check the syntax and semantics of embedded SQL data manipulation statements and PL/SQL blocks
- access Oracle databases on multiple nodes concurrently using SQL*Net
- 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 the SQL Communications Area (SQLCA) and WHENEVER statement
- use an enhanced set of diagnostics provided by the Oracle Communications Area (ORACA)

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

Do the Oracle Precompilers Meet Industry Standards?

SQL has become the standard language for relational database management systems. This section describes how the Oracle Precompilers conform to 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*
- International Standard ISO/IEC 9075:1992, *Database Language SQL*
- ANSI Document ANSI X3.168-1992, *Database Language Embedded 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, FORTRAN, Pascal, or PL/I.

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 standards. In addition, it specifies minimum sizing parameters for database constructs and requires a "FIPS Flagger" to identify ANSI extensions.

For copies of the ANSI standards, write to

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

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

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

Compliance

The Oracle Precompilers comply 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 Oracle Precompilers provide the *FIPS Flagger*, which flags ANSI extensions. An *extension* is any SQL element that violates ANSI format or syntax rules, except privilege enforcement rules. For a list of Oracle extensions to standard SQL, see the *Oracle Database SQL Language 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 6-24.

Certification

NIST tested the Oracle Precompilers 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 and FORTRAN embedded SQL standards. As a result, the Oracle Precompilers were certified 100% ANSI-compliant.

For more information about the tests, write to

National Computer Systems LaboratoryAttn: Software Standards Testing ProgramNational Institute of Standards and TechnologyGaithersburg, MD 20899, USA

2

Learning the Basics

This chapter explains the following:

- Key Concepts of Embedded SQL Programming
- Steps in Developing an Embedded SQL Application
- A Sample Program
- Sample Tables

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

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

Key Concepts of Embedded SQL Programming

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

- Embedded SQL Statements
- Executable versus Declarative Statements
- Embedded SQL Syntax
- Static versus Dynamic SQL Statements
- Embedded PL/SQL Blocks
- Host and Indicator Variables
- Oracle Datatypes
- Arrays
- Datatype Equivalencing
- Private SQL Areas, Cursors, and Active Sets
- Transactions
- Errors and Warnings

Embedded SQL Statements

The term *embedded SQL* refers to SQL statements placed within an application program. Because 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 the Pro*COBOL Precompiler you can embed SQL statements in a COBOL host program.

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

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

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

Executable versus Declarative Statements

Embedded SQL includes all the interactive SQL statements plus others that allow you to transfer data between Oracle and a host program. There are two types of embedded SQL statements: *executable* and *declarative*.

Executable statements result in calls to the run-time library `SQLLIB`. You use them to connect to Oracle, to define, query, and manipulate Oracle data, to control access to Oracle data, and to process transactions. They can be placed wherever any other host-language executable statements can be placed.

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

Table 2–1 groups the various embedded SQL statements and Table 2–2 groups the various executable SQL statements.

Table 2–1 Embedded SQL Statements

Declarative SQL	Description
`STATEMENT`	PURPOSE
`ARRAYLEN*`	To use host arrays with PL/SQL
`BEGIN DECLARE SECTION*` `END DECLARE SECTION*`	To declare host variables
`DECLARE*`	To name Oracle objects
`INCLUDE*`	To copy in files
`TYPE*`	To equivalence datatypes
`VAR*`	To equivalence variables
`WHENEVER*`	To handle run-time errors

*Has no interactive counterpart

Table 2–2 Executable SQL Statements and their Descriptions

Executable SQL	Descriptions
STATEMENT	PURPOSE
ALLOCATE*	To define and control Oracle data
ALTER	
ANALYZE	
AUDIT	
COMMENT	
CONNECT*	
CREATE	
DROP	
GRANT	
NOAUDIT	
RENAME	
REVOKE	
TRUNCATE	
CLOSE*	
DELETE	To query and manipulate Oracle data
EXPLAIN PLAN	
FETCH*	
INSERT	
LOCK TABLE	
OPEN*	
SELECT	
UPDATE	
COMMIT	To process transactions
ROLLBACK	
SAVEPOINT	
SET TRANSACTION	
DESCRIBE*	To use dynamic SQL
EXECUTE*	
PREPARE*	
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 keywords EXEC SQL and end them with the SQL statement terminator for your host language. The precompiler translates all executable EXEC SQL statements into calls to the run-time library SQLLIB.

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

```
ROLLBACK WORK; -- interactive
EXEC SQL ROLLBACK WORK; -- embedded
```

For a summary of embedded SQL syntax, see the *Oracle Database SQL Language Reference*.

Static versus Dynamic SQL Statements

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

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

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

Embedded PL/SQL Blocks

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

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

Host and Indicator Variables

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

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

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

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

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

Oracle Datatypes

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

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

External datatypes specify how data is stored in host variables. When your host program inputs data to Oracle, if necessary, Oracle converts between the external datatype of the input host variable and the internal datatype of the 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.

Arrays

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

Datatype Equivalencing

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

On a variable-by-variable basis, you can equivalence supported host language datatypes to Oracle external datatypes.

Private SQL Areas, Cursors, and Active Sets

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

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

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

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

Transactions

A *transaction* is a series of logically related SQL statements (two `UPDATE`s that credit one bank account and debit another, for example) that Oracle treats as a unit, so that all changes brought about by the statements are made permanent or undone at the same time. The current transaction consists of all data manipulation statements executed since the last data definition, `COMMIT`, or `ROLLBACK` statement was executed.

To help ensure the consistency of your database, the Oracle Precompilers let you define transactions by using the `COMMIT`, `ROLLBACK`, and `SAVEPOINT` statements. `COMMIT` makes permanent any changes made during the current transaction. `ROLLBACK` ends the current transaction and undoes any changes made since the transaction began. `SAVEPOINT` marks the current point in a transaction; used with `ROLLBACK`, it undoes part of a transaction.

Errors and Warnings

When you execute an embedded SQL statement, it either succeeds or fails, and might result in an error or warning. You need a way to handle these results. The Oracle Precompilers provide four error handling mechanisms:

- `SQLCODE` status variable
- `SQLSTATE` status variable
- SQL Communications Area (SQLCA) and `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 and WHENEVER Statement

The SQLCA is a data structure that defines program variables used by Oracle to pass run-time 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 verify if a `DELETE` statement succeeded and if so, how many rows were deleted.

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

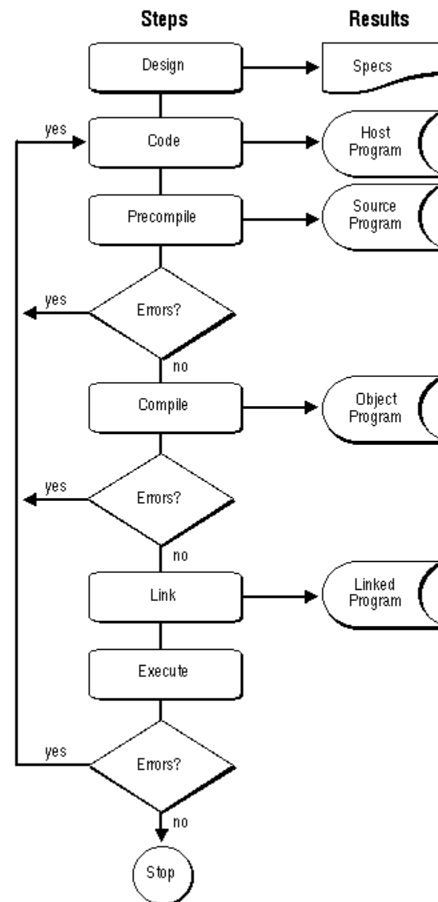
ORACA

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

Steps in Developing an Embedded SQL Application

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

Figure 2–1 Application Development Process



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

A Sample Program

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

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

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

```
EXEC SQL END DECLARE SECTION;
-- copy in the SQL Communications Area
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
-- handle processing errors
EXEC SQL WHENEVER SQLERROR DO sql_error;
-- log on to Oracle
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
display 'Employee number? ';
read emp_number;
-- query database for employee's name, salary, and commission
-- and assign values to host variables
EXEC SQL SELECT ENAME, SAL, COMM
 INTO :emp_name, :salary, :commission:ind_comm
 FROM EMP
 WHERE EMPNO = :emp_number;
display 'Employee Salary Commission';
display '-------- ------ ----------';
-- display employee's name, salary, and commission (if not null)
IF ind_comm = -1 THEN -- commission is null
 display emp_name, salary, 'Not applicable';
ELSE
 display emp_name, salary, commission;
ENDIF;
-- release resources and log off the database
EXEC SQL COMMIT WORK RELEASE;
display 'Have a good day';
exit program;
ROUTINE sql_error
BEGIN
 -- avoid an infinite loop if the rollback results in an error
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 -- release resources and log off the database
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
END sql_error;
```

Sample Tables

Most programming examples in this guide use two sample database tables: `DEPT` and `EMP`. Their definitions follow:

```
CREATE TABLE DEPT
 (DEPTNO NUMBER(2),
 DNAME VARCHAR2(14),
 LOC VARCHAR2(13))
CREATE TABLE EMP
 (EMPNO NUMBER(4) primary key,
 ENAME VARCHAR2(10),
 JOB VARCHAR2(9),
 MGR NUMBER(4),
 HIREDATE DATE,
 SAL NUMBER(7,2),
 COMM NUMBER(7,2),
```

```
 DEPTNO NUMBER(2))
```

Sample Data

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

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

3

Meeting Program Requirements

This chapter explains the following:

- The Declare Section
- INCLUDE Statements
- The SQLCA
- Oracle Datatypes
- Datatype Conversion
- Declaring and Referencing Host Variables
- Declaring and Referencing Indicator Variables
- Datatype Equivalencing
- Globalization Support
- Multibyte Globalization Support Character Sets
- Concurrent Logons
- Embedding OCI (Oracle Call Interface) Calls
- Developing X/Open Applications

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

The Declare Section

You must declare all program variables to be used in SQL statements (that is, all host variables) in the *Declare Section*. If you use an undeclared host variable in a SQL statement, the precompiler issues an error message. For a complete listing of error messages see *Oracle Database Error Messages*.

The Declare Section begins with the statement

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

and ends with the statement

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

In COBOL, the statement terminator is `END-EXEC`. In FORTRAN, it is a carriage return.

Between these two statements, only the following items are allowed:

- host-variable and indicator-variable declarations
- `EXEC SQL DECLARE` statements
- `EXEC SQL INCLUDE` statements
- `EXEC SQL VAR` statements
- `EXEC ORACLE` statements
- host-language comments

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

An Example

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

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

For more information about declaring host variables, see "Declaring and Referencing Host Variables".

INCLUDE Statements

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

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

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

You can include any file. If a file contains embedded SQL, you *must* include it because only included files are precompiled. If you do not specify a file extension, the precompiler assumes the default extension for source files, which is language-dependent (see your host-language supplement to this Guide).

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

```
INCLUDE=<path>
```

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

The precompiler searches first in the current directory, then in the directory specified by `INCLUDE`, and finally in a directory for standard `INCLUDE` files. So, you need not specify a directory path for standard files such as the SQLCA and ORACA. You must

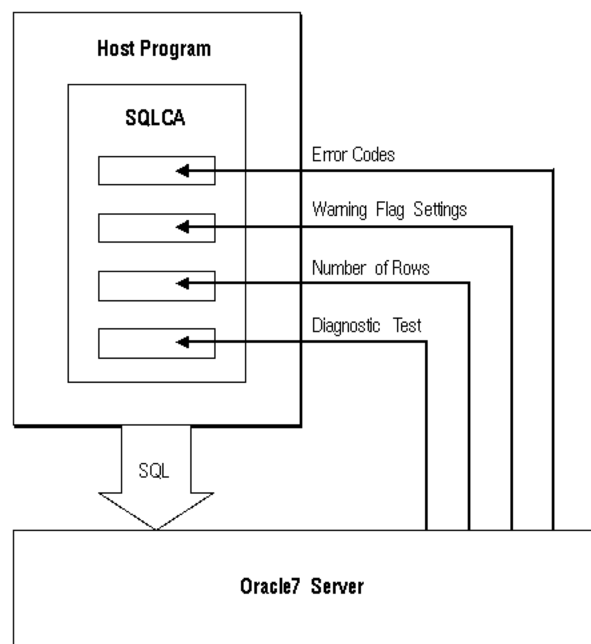
still use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory.

If your operating system is case-sensitive (like UNIX for example), be sure to specify the same upper/lowercase filename under which the file is stored. The syntax for specifying a directory path is system-specific. Check your system-specific Oracle manuals.

The SQLCA

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

Figure 3–1 Updating the SQLCA



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

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

Oracle Datatypes

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

At precompile time, each host variable in the Declare Section is associated with an external datatype code. At run time, the datatype code of every host variable used in a

SQL statement is passed to Oracle. Oracle uses the codes to convert between internal and external datatypes.

Note: You can override default datatype conversions by using dynamic SQL Method 4 or datatype equivalencing. For information about dynamic SQL Method 4, see "Using Method 4". For information about datatype equivalencing, see "Datatype Equivalencing".

Internal Datatypes

Table 3–1 lists the internal datatypes that Oracle uses for database columns and pseudocolumns.

Table 3–1 Column and Pseudo Column Datatypes

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

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

Brief descriptions of the internal datatypes follow. For more information, see the *Oracle Database SQL Language Reference*.

CHAR

You use the CHAR datatype to store fixed-length character data. How the data is represented internally depends on the database character set. The CHAR datatype takes an optional parameter that lets you specify a maximum width up to 255 bytes. The syntax follows:

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

You cannot use a constant or variable to specify the maximum width; you must use an integer literal. If you do not specify the maximum width, it defaults to 1. Remember, you specify the maximum width of a CHAR(*n*) column in bytes, not characters. So, if a CHAR(*n*) column stores multibyte (2-byte) characters, its maximum width is less than *n*/2 characters.

DATE

You use the DATE datatype to store dates and times in 7-byte, fixed-length fields. The date portion defaults to the first day of the current month; the time portion defaults to midnight.

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

LONG

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

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

LONG RAW

You use the LONG RAW datatype to store variable-length binary data or byte strings. The maximum width of a LONG RAW column is 2147483647 bytes or two gigabytes.

LONG RAW data is like LONG data, except that Oracle assumes nothing about the meaning of LONG RAW data and does no character set conversions when you transmit LONG RAW data from one system to another. The restrictions that apply to LONG data also apply to LONG RAW data.

MLSLABEL

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

You can use the MLSLABEL datatype to define a database column. You can insert any valid operating system label into a column of type MLSLABEL. If the label is in text format, Oracle converts it to a binary value automatically. The text string can be up to 255 bytes long. However, the internal length of an MLSLABEL value is between 2 and 5 bytes.

You can also select values from a MLSLABEL column into a character variable. Oracle converts the internal binary value to a VARCHAR2 value automatically.

NUMBER

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

The maximum precision of a NUMBER value is 38; the magnitude range is 1.0E-129 to 9.99E125. Scale can range from -84 to 127. For example, a scale of -3 means the number

is rounded to the nearest thousand (3456 becomes 3000). A scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46).

When you specify precision and scale, Oracle does extra integrity checks before storing the data. If a value exceeds the precision, Oracle issues an error message; if a value exceeds the scale, Oracle rounds the value.

RAW

You use the `RAW` datatype to store binary data or byte strings (a sequence of graphics characters, for example). `RAW` data is not interpreted by Oracle.

The `RAW` datatype takes a required parameter that lets you specify a maximum width up to 255 bytes. The syntax follows:

```
RAW(maximum_width)
```

You cannot use a constant or variable to specify the maximum width; you must use an integer literal.

`RAW` data is like `CHAR` data, except that Oracle assumes nothing about the meaning of `RAW` data and does no character set conversions (from 7-bit ASCII to EBCDIC Code Page 500 for example) when you transmit `RAW` data from one system to another.

ROWID

Internally, every table in an Oracle database has a pseudocolumn named `ROWID`, which stores binary values called *rowids*. `ROWID`s uniquely identify rows and provide the fastest way to access particular rows.

VARCHAR2

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

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

```
VARCHAR2(maximum_width)
```

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

You specify the maximum width of a `VARCHAR2`(*n*) column in bytes, not characters. So, if a `VARCHAR2`(*n*) column stores multibyte (2-byte) characters, its maximum width is less than *n*/2 characters.

SQL Pseudocolumns and Functions

SQL recognizes the pseudocolumns in Table 3–2, which return specific data items:

Table 3–2 Pseudo Column Datatypes

Pseudocolumn	Internal Datatype
CURRVAL	NUMBER
LEVEL	NUMBER
NEXTVAL	NUMBER
ROWID	ROWID

Table 3–2 Pseudo Column Datatypes

Pseudocolumn	Internal Datatype
ROWLABEL	MLSLABEL
ROWNUM	NUMBER

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

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

Table 3–3 Parameterless Function 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;
```

Brief descriptions of the SQL pseudocolumns and functions follow. For details, see the *Oracle Database SQL Language Reference*.

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

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

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

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

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

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

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

ROWID returns a row address in hexadecimal.

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

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

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

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;
```

SYSDATE returns the current date and time.

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

USER returns the username of the current Oracle user.

ROWLABEL Column

SQL also recognizes the special column ROWLABEL, which Oracle creates for every database table. Like other columns, ROWLABEL can be referenced in SQL statements. ROWLABEL returns the operating system label for a row.

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

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

For more information about the ROWLABEL column, see the *Oracle Database Administrator's Guide*.

External Datatypes

As Table 3–4 shows, the external datatypes include all the internal datatypes plus several datatypes found in other supported host languages. For example, the STRING external datatype refers to a C null-terminated string, and the DECIMAL datatype refers to COBOL packed decimals. You use the datatype names in datatype equivalencing, and you use the datatype codes in dynamic SQL Method 4.

Table 3–4 External Datatypes

Name	Code	Description
CHAR	1 96	<= 65535-byte, variable-length character string <=65535-byte, fixed-length character string (see note 1)
CHARF	96	<= 65535-byte, fixed-length character string
CHARZ	97	<= 65535-byte, fixed-length, null-terminated string (see note 2)
DATE	12	7-byte, fixed-length date/time value
DECIMAL	7	COBOL packed decimal

Table 3–4 External Datatypes

Name	Code	Description
DISPLAY	91	COBOL numeric character string
FLOAT	4	4-byte or 8-byte floating-point number
INTEGER	3	2-byte or 4-byte signed integer
LONG	8	<= 2147483647-byte, fixed-length string
LONG RAW	24	<= 217483647-byte, fixed-length binary data (see note 3)
LONG VARCHAR	94	<= 217483643-byte, variable-length string (see note 3)
LONG VARRAW	95	<= 217483643-byte, variable-length binary data
MLSLABEL	106	2..5-byte, variable-length binary data
NUMBER	2	integer or floating-point number
RAW	23	<= 65535-byte, fixed-length binary data (see note 2)
ROWID	11	(typically) 13-byte, fixed-length binary value
STRING	5	<= 65535-byte, variable-length, null-terminated character string (see note 2)
UNSIGNED	68	2-byte or 4-byte unsigned integer
VARCHAR	9	<= 65533-byte, variable-length character string (see note 3)
VARCHAR2	1	<= 65535-byte, variable-length character string (see note 2)
VARNUM	6	variable-length binary number
VARRAW	15	<= 65533-byte, variable-length binary data (see note 3)

Notes:

1. `CHAR` is datatype 1 when `MODE={ORACLE|ANSI13|ANSI14}` and datatype 96 when `MODE=ANSI`.
2. Maximum size is 32767 (32K) on some platforms.
3. Do not include the n-byte length field in an `EXEC SQL VAR` statement.

CHAR

CHAR behavior depends on the settings of the options DBMS and MODE.

CHARF

When MODE=ANSI, Oracle assigns the CHAR datatype to all character host variables. You use the CHAR datatype to store fixed-length character strings. On most platforms, the maximum length of a CHAR value is 65535 (64K) bytes. See Table 6–4 for more information about the relationship between the DBMS and MODE options.

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

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

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

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

CHARZ

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

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

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

DATE

You use the `DATE` datatype to store dates and times in 7-byte, fixed-length fields. As Table 3–5 shows, the century, year, month, day, hour (in 24-hour format), minute, and second are stored in that order from left to right.

Table 3–5 DATE Datatype Example

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

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

DECIMAL

With Pro*COBOL, you use the `DECIMAL` datatype to store packed decimal numbers for calculation. In COBOL, the host variable must be a signed COMP-3 field with an implied decimal point. If significant digits are lost during data conversion, Oracle fills the host variable with asterisks.

DISPLAY

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

FLOAT

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

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

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

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

LONG

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

LONG RAW

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

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

LONG VARCHAR

You use the LONG VARCHAR datatype to store variable-length character strings. LONG VARCHAR variables have a 4-byte length field followed by a string field. The maximum length of the string field is 2147483643 bytes. In an EXEC SQL VAR statement, do *not* include the 4-byte length field.

LONG VARRAW

You use the LONG VARRAW datatype to store binary data or byte strings. LONG VARRAW variables have a 4-byte length field followed by a data field. The maximum length of the data field is 2147483643 bytes. In an EXEC SQL VAR statement, do *not* include the 4-byte length field.

MLSLABEL

You use the MLSLABEL datatype to store variable-length, binary operating system labels. Oracle uses labels to control access to data. You can use the MLSLABEL datatype to define a column. You can insert any valid operating system label into a column of type MLSLABEL.

On Input. Oracle translates the input value into a binary label, which must be a valid operating system label. If the label is invalid, Oracle issues an error message. If the label is valid, Oracle stores it in the target database column.

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

NUMBER

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

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

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

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

RAW

You use the RAW datatype to store fixed-length binary data or byte strings. On most platforms, the maximum length of a RAW value is 65535 bytes. RAW data is like CHAR data, except that Oracle assumes nothing about the meaning of RAW data and does no character set conversions when you transmit RAW data from one system to another.

ROWID

You use the ROWID datatype to store binary rowids in (typically 13-byte) fixed-length fields. The field size is port-specific. So, check your system-specific Oracle manuals. You can use VARCHAR2 host variables to store rowids in a readable format. When you select or fetch a rowid into a VARCHAR2 host variable, Oracle converts the binary value to an 18-byte character string and returns it in the format

```
BBBBBBBB.RRRR.FFFF
```

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

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

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

Typically, you fetch a rowid into a VARCHAR2 host variable, then compare the host variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement. That way, you can identify the latest row fetched by a cursor.

> **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 null-terminated.

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

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

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

UNSIGNED

You use the `UNSIGNED` datatype to store unsigned integers. An unsigned integer is a binary number of 2 or 4 bytes. The order of the bytes in a word is system-dependent. You must specify a length for input and output host variables. On output, if the column value is a floating point number, Oracle truncates the fractional part. You should not need this external type in Pro*COBOL or Pro*FORTRAN.

VARCHAR

You use the `VARCHAR` datatype to store variable-length character strings. `VARCHAR` variables have a 2-byte length field followed by a <= 65533-byte string field. However, for `VARCHAR` array elements, the maximum length of the string field is 65530 bytes. When you specify the length of a `VARCHAR` variable, be sure to include 2 bytes for the length field. For longer strings, use the `LONG VARCHAR` datatype. In an `EXEC SQL VAR` statement, do *not* include the 2-byte length field.

VARCHAR2

When `MODE=ORACLE`, Oracle assigns the `VARCHAR2` datatype to all character host variables. You use the `VARCHAR2` datatype to store variable-length character strings. On most platforms, the maximum length of a `VARCHAR2` value is 65535 bytes.

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

On Input. Oracle reads the number of bytes specified for the input host variable, strips any trailing blanks, then stores the input value in the target database column. Be careful. An uninitialized host variable can contain nulls. So, always blank-pad a character input host variable to its declared length. (COBOL PIC X(*n*) and FORTRAN CHARACTER*n* variables do this automatically.)

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

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

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

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

Oracle can convert `NUMBER` column values to character values. The length of the character host variable determines precision. If the host variable is too short for the number, scientific notation is used. For example, if you select the column value abcdefg89 into a host variable of length 6, Oracle returns the value "1.2E08" to the host variable.

VARNUM

The `VARNUM` datatype is like the `NUMBER` datatype, except that the first byte of a `VARNUM` variable stores the length of the value. On input, you must set the first byte of the host variable to the length of the value. On output, the host variable contains the length followed by the number as represented internally by Oracle. To accommodate the largest possible number, the host variable must be 22 bytes long. After selecting a column value into a `VARNUM` host variable, you can check the first byte to get the length of the value.

VARRAW

You use the `VARRAW` datatype to store variable-length binary data or byte strings. The `VARRAW` datatype is like the RAW datatype, except that `VARRAW` variables have a 2-byte length field followed by a <=65533-byte data field. For longer strings, use the `LONG VARRAW` datatype. In an `EXEC SQL VAR` statement, do *not* include the 2-byte length field. To get the length of a `VARRAW` variable, simply refer to its length field.

Datatype Conversion

At precompile time, an external datatype is assigned to each host variable in the Declare Section. For example, the precompiler assigns the `INTEGER` external datatype to integer host variables. At run time, the datatype code of every host variable used in a SQL statement is passed to Oracle. Oracle uses the codes to convert between internal and external datatypes.

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

However, 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 (Globalization Support) parameters in the Oracle 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 *Oracle Database Advanced Application Developer's Guide*.

Table 3–6 shows the supported conversions between internal and external datatypes.

Note: Legend:

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

  I = input only On output, column value is returned in same format.

  O = output only
- On input, host string must be the default DATE character format.

  I/O = input or output On output, column value is returned in same format
- On input, host string must be in hexadecimal format. On output, column value is returned in same format.
- On output, column value must represent a valid number.
- On input, length must be less than or equal to 2000.
- On input, column value is stored in hexadecimal format. On output, column value must be in hexadecimal format.
- On input, host string must be a valid operating system label in text format. On output, column value is returned in same format.
- On input, host string must be a valid operating system label in raw format. On output, column value is returned in same format.

Table 3–6 Conversion Between Internal and External Datatypes

External	Internal								
	CHAR	DATE	LONG	LONG RAW	MLSLABEL	NUMBER	RAW	ROWID	VARCHAR2
CHAR	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
CHARF	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
CHARZ	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
DATE	I/O	I/O	I						I/O
DECIMAL	I/O		I			I/O			I/O
DISPLAY	I/O		I			I/O			I/O
FLOAT	I/O		I			I/O			I/O
INTEGER	I/O		I			I/O			I/O
LONG	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O

Table 3–6 (Cont.) Conversion Between Internal and External Datatypes

External	Internal								
LONG RAW	O		I	I/O			I/O		O
LONG VARCHAR	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
LONG VARRAW	I/O		I	I/O			I/O		I/O
MLSLABEL	I/O		I/O		I/O				I/O
NUMBER	I/O		I			I/O			I/O
RAW	I/O		I	I/O			I/O		I/O
ROWID	I		I					I/O	I
STRING	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
UNSIGNED	I/O		I			I/O			I/O
VARCHAR	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
VARCHAR2	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
VARNUM	I/O		I			I/O			I/O
VARRAW	I/O		I	I/O			I/O		I/O

DATE Values

When you select a `DATE` column value into a character host variable, Oracle must convert the internal binary value to an external character value. So, Oracle implicitly calls the SQL function `TO_CHAR`, which returns a character string in the default date format. The default is set by the Oracle initialization parameter Globalization `Support_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. Oracle 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.

RAW and LONG RAW Values

When you select a `RAW` or `LONG RAW` column value into a character host variable, Oracle must convert the internal binary value to an external character value. In this case, Oracle returns each binary byte of `RAW` or `LONG RAW` data as a pair of characters. Each character represents the hexadecimal equivalent of a nibble (half a byte). For example, Oracle returns the binary byte 11111111 as the pair of characters "FF". The SQL function `RAWTOHEX` 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, Oracle issues the following error message:

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

Declaring and Referencing Host Variables

Every program variable used in a SQL statement must be declared as a host variable. You declare a host variable in the Declare Section according to the rules of the host language. Normal scoping rules apply. Host variable names can be any length, but only the first 31 characters are significant. For ANSI/ISO compliance, a host variable

name must be <= 18 characters long, begin with a letter, and not contain consecutive or trailing underscores.

The external datatype of a host variable and the internal datatype of its source or target database column need not be the same, but they must be compatible. Table 3–6 shows the compatible datatypes between which Oracle converts automatically when necessary.

The Oracle Precompilers support most built-in host language datatypes. For a list of supported datatypes, see your host-language supplement. User-defined datatypes are not supported. Datatype equivalencing is discussed in the next section.

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

Some Examples

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

```
-- declare host variables
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 emp_name CHARACTER(10);
 dept_number INTEGER;
EXEC SQL END DECLARE SECTION;
...
display 'Employee number? ';
read emp_number;
EXEC SQL SELECT DEPTNO, ENAME INTO :dept_number, :emp_name
 FROM EMP
 WHERE EMPNO = :emp_number;
```

For more information about using host variables, see "Using Host Variables".

VARCHAR Variables

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

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 ENAME PIC X(20) VARYING.
 EXEC SQL END DECLARE SECTION END-EXEC.
```

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

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

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

For more information about `VARCHAR`s, see your host-language supplement to this Guide.

Host Variable Guidelines

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

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

A host variable must *not* be

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

A host variable can be

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

Declaring and Referencing Indicator Variables

You can associate every host variable with an optional indicator variable. An indicator variable must be defined in the Declare Section as a 2-byte integer and, in SQL statements, must be prefixed with a colon and must directly follows its host variable unless you use the keyword `INDICATOR`.

INDICATOR Keyword

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

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

which is equivalent to

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

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

An Example

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

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

```
 emp_number INTEGER;
 salary REAL;
 commission REAL;
 ind_comm SMALLINT; -- indicator variable
EXEC SQL END DECLARE SECTION;
 pay REAL; -- not used in a SQL statement
display 'Employee number? ';
read emp_number;
EXEC SQL SELECT SAL, COMM
 INTO :salary, :commission:ind_comm
 FROM EMP
 WHERE EMPNO = :emp_number;
IF ind_comm = -1 THEN -- commission is null
 set pay = salary;
ELSE
 set pay = salary + commission;
ENDIF;
```

For more information, see "Using Indicator Variables".

Indicator Variable Guidelines

The following guidelines apply to declaring and referencing indicator variables. An indicator variable must be

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

An indicator variable must *not* be

- prefixed with a colon in host language statements
- appended to its host variable in host language statements
- an Oracle reserved word

Datatype Equivalencing

Datatype equivalencing lets you customize the way Oracle interprets input data and the way Oracle formats output data. On a variable-by- variable basis, you can equivalence supported host language datatypes to the Oracle external datatypes.

Why Equivalence Datatypes?

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

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

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

```
DD-MON-YY
```

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

Host Variable Equivalencing

By default, the Oracle Precompilers assign a specific external datatype to every host variable. (These default assignments are tabulated in your supplement to this Guide.) You can override the default assignments by equivalencing host variables to Oracle external datatypes in the Declare Section. This is called *host variable equivalencing*.

The syntax you use is:

```
EXEC SQL VAR <host_variable>
 IS <ext_type_name> [({<length> | <precision>,<scale>})];
```

where, `host_variable` is an input or output host variable (or host array) declared *earlier* in the Declare Section. The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by an *n*-byte data field, where *n* lies in the range 1 .. 65533. So, if *type_name* is VARCHAR or VARRAW, *host_variable* must be at least 3 bytes long.

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

`ext_type_name` is the name of a valid external datatype such as RAW or STRING.

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

When *type_name* is DECIMAL or DISPLAY, you must specify *precision* and *scale* instead of *length*. When *type_name* is VARNUM, ROWID, or DATE, you cannot specify *length* because it is 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. The precompiler 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. However, 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.

Table 3–7 shows which parameters to use with each external datatype.

An Example

Suppose you want to select employee names from the EMP table, then pass them to a routine that expects null-terminated strings. You need not explicitly null-terminate the

names. Simply equivalence a host variable to the STRING external datatype, as follows:

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

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

Table 3–7 External Datatype Parameters

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
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 (see note 1)	n/a	n/a	none
LONG VARRAW	required (see note 1)	n/a	n/a	none
MLSLABEL	required	n/a	n/a	none
NUMBER	n/a	n/a	n/a	not available
STRING	optional	n/a	n/a	declared length of variable
RAW	optional	n/a	n/a	declared length of variable
ROWID	n/a	n/a	n/a	13 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

Note:

1. If the data field exceeds 65,533 bytes, pass -1.
2. This length is typical but the default is port-specific.

Using the CHARF Datatype Specifier

You can use the datatype specifier CHARF in VAR and TYPE statements to equivalence host-language datatypes to the fixed-length ANSI datatype CHAR--regardless of the DBMS setting.

When MODE=ANSI, specifying the datatype CHAR in a TYPE statement equivalences the host-language datatype to the fixed-length ANSI datatype CHAR (Oracle external datatype code 96). However, when MODE=ORACLE, the host-language datatype is equivalenced to the variable-length datatype VARCHAR2 (code 1), which might not be what you want.

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 MODE=ORACLE.

Guidelines

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

After selecting a column value into a VARNUM host variable, you can check the first byte to get the length of the value. Table 3–8 gives some examples of returned VARNUM values.

Table 3–8 Examples of VARNUM Values Returned

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

Convert DATE values to a character format such as "DD-MON-YY" because, normally, that is how your program outputs (displays for example) or inputs them.

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

Globalization Support

Although the widely-used 7- or 8-bit ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, some Asian languages, such as Japanese, contain

thousands of characters. These languages require 16 bits (two bytes) to represent each character. How does Oracle deal with such dissimilar languages?

Oracle provides Globalization Support (Globalization Support), 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 allows users around the world to interact with Oracle 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 Oracle initialization file. Table 3–9 shows what each Globalization Support parameter specifies.

Table 3–9 Globalization Support Parameters

Globalization Support Parameter	Specifies ...
Globalization Support_LANGUAGE	language-dependent conventions
Globalization Support_TERRITORY	territory-dependent conventions
Globalization Support_DATE_FORMAT	date format
Globalization Support_DATE_LANGUAGE	language for day and month names
Globalization Support_NUMERIC_CHARACTERS	decimal character and group separator
Globalization Support_CURRENCY	local currency symbol
Globalization Support_ISO_CURRENCY	ISO currency symbol
Globalization Support_SORT	sort sequence

The main parameters are Globalization Support_LANGUAGE and Globalization Support_TERRITORY. Globalization Support_LANGUAGE specifies the default values for language-dependent features, which include

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

Globalization Support_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 Globalization Support_LANG as follows:

```
Globalization Support_LANG = <language>_<territory>.<character set>
```

where *language* specifies the value of Globalization Support_LANGUAGE for the user session, *territory* specifies the value of Globalization Support_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 Globalization Support_LANG as an environment variable (or the equivalent on your system). For example, on UNIX using the C shell, you might define Globalization Support_LANG as follows:

```
setenv Globalization Support_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 <Globalization Support_parameter> = <value>
```

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

You can use the functions Globalization Support_INITCAP, Globalization Support_LOWER, and Globalization Support_UPPER to handle special instances of case conversion. And, you can use the function Globalization SupportSORT 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 *Oracle Database Advanced Application Developer's Guide*.

Multibyte Globalization Support Character Sets

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

- recognition of multibyte character strings by the precompiler in embedded SQL statements.
- the ANSI standard COBOL PIC N datatype declaration clause, which instructs the precompiler to interpret host character variables as strings of double-byte characters.

Oracle supports multibyte strings through the precompiler run-time library, SQLLIB.

Character Strings in Embedded SQL

A multibyte Globalization Support character string in an embedded SQL statement consists of a character literal that identifies the string as a multibyte string, followed by the string enclosed in single quotes.

For example, an embedded SQL statement like

```
EXEC SQL
 SELECT empno INTO :emp_num FROM emp
 WHERE ename=N'Kuroda'
 END-EXEC.
```

contains a multibyte character string, since the N character literal preceding the string "Kuroda" identifies it as a multibyte string.

Dynamic SQL

Because dynamic SQL statements are not processed at precompile time, and Oracle does not process multibyte Globalization Support strings itself, you cannot embed multibyte Globalization Support strings in dynamic SQL statements.

Embedded DDL

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 extended column types, such as NCHAR, within embedded DDL statements results in an execution error rather than a precompile error.

Multibyteultibyte Globalization Support Host Variables

The Pro*COBOL Precompiler uses the ANSI standard PIC N clause to declare host variables for multibyte character data. Variables declared using the PIC N clause are recognized as string variables of double-byte characters.

- Globalization Support_LOCAL
- VARCHAR

For more information about these options, see Chapter 6, "Running the Oracle Precompilers".

Restrictions

Tables Disallowed.

Host variables declared using the PIC N datatype must not be tables.

No Odd Byte Widths. Oracle `CHAR` columns should not be used to store multibyte Globalization Support 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 Globalization Support (PIC N) host variable.

No Host Variable Equivalencing. multibyte Globalization Support character variables cannot be equivalenced using an `EXEC SQL VAR` statement.

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

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 "External Datatypes" in *Pro*COBOL Programmer's Guide*.

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

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

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

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

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

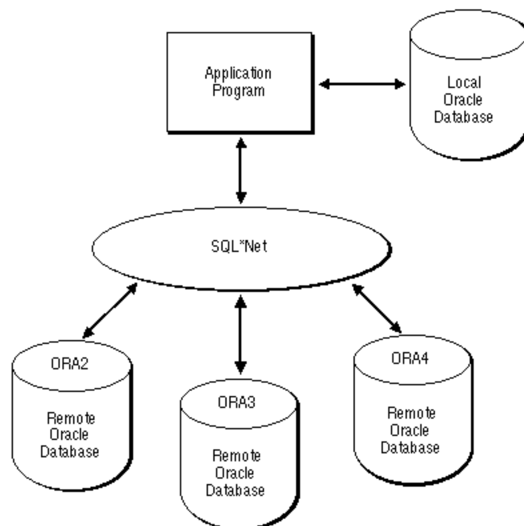
Indicator Variables

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".

Concurrent Logons

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

Figure 3–2 Connecting through SQL*Net



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

Some Preliminaries

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

A *protocol* is a set of rules for accessing a network. The rules establish such things as procedures for recovering after a failure and formats for transmitting data and checking errors.

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

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

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

Default Databases and Connections

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

A *default* connection is made using a CONNECT statement without an AT clause. The connection can be to any default or nondefault database at any local or remote node. SQL statements without an AT clause are executed against the default connection. Conversely, a *nondefault* connection is made by a CONNECT statement that has an AT clause. A SQL statement with an AT clause is executed against the nondefault connection.

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

Explicit Logons

Usually, you establish a connection to Oracle as follows:

```
EXEC SQL CONNECT :userid IDENTIFIED BY :password
```

Or, you might use

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

where *usr_pwd* contains *username*/*password*.

You can also log on automatically as shown. If you do not specify a database and node, you are connected to the default database at the current node. If you want to connect to a different database, you must explicitly identify that database.

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

Single Explicit Logons

In the following example, you connect to a single nondefault database at a remote node:

> **Note:** For simplicity in demonstrating this feature, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

```
-- Declare necessary host variables.
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_string CHARACTER(20);
EXEC SQL END DECLARE SECTION;
set username = 'scott';
set password = 'tiger';
set db_string = 'd:newyork-nondef';
-- Assign a unique name to the database connection.
EXEC SQL DECLARE db_name DATABASE;
-- Connect to the nondefault database
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name USING :db_string;
```

The identifiers in this example serve the following purposes:

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

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

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

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_name CHARACTER(10);
 db_string CHARACTER(20);
EXEC SQL END DECLARE SECTION;
set username = 'scott';
set password = 'tiger';
set db_name = 'oracle1';
set db_string = 'd:newyork-nondef';
-- connect to the nondefault database
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT :db_name USING :db_string;
...
```

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

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

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

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

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

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

If *db_name* is a host variable, its declaration must be within the scope of all SQL statements that refer to the declared cursor. For example, if you open the cursor in one subprogram, then fetch from it in another, 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 change the connection associated with a cursor. However, you cannot change the association while the cursor is open. Consider the following example:

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ...
set db_name = 'oracle1';
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO ...
set db_name = 'oracle2';
EXEC SQL OPEN emp_cursor; -- illegal, cursor still open
EXEC SQL FETCH emp_cursor INTO ...
```

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

```
EXEC SQL CLOSE emp_cursor; -- close cursor first
set db_name = 'oracle2';
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO ...
```

Dynamic SQL. Dynamic SQL statements are similar to cursor control statements in that some never use the AT clause. For dynamic SQL Method 1, you must use the AT clause if you want to execute the statement at a nondefault connection. An example follows:

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

For Methods 2, 3, and 4, you use the AT clause only in the DECLARE STATEMENT statement if you want to execute the statement at a 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;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL EXECUTE sql_stmt;
```

The following example shows Method 3:

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

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;
```

Multiple Explicit Logons

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

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_string1 CHARACTER(20);
 db_string2 CHARACTER(20);
EXEC SQL END DECLARE SECTION;
...
set username = 'scott';
set password = 'tiger';
set db_string1 = 'New_York';
set db_string2 = 'Boston';
-- give each database connection a unique name
EXEC SQL DECLARE db_name1 DATABASE;
EXEC SQL DECLARE db_name2 DATABASE;
-- connect to the two nondefault databases
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name1 USING :db_string1;
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name2 USING :db_string2;
```

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

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

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_name CHARACTER(10);
 db_string CHARACTER(20);
EXEC SQL END DECLARE SECTION;
...
set username = 'scott';
set password = 'tiger';
FOR EACH nondefault database
 -- get next database name and SQL*Net string
 display 'Database Name? ';
 read db_name;
 display 'SQL*Net String? ';
 read db_string;
 -- connect to the nondefault database
 EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT :db_name USING :db_string;
ENDFOR;
```

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

```
set username = 'scott';
set password = 'tiger';
set db_string = 'd:newyork-nondef';
FOR EACH nondefault database
 -- get next database name
 display 'Database Name? ';
 read db_name;
 -- connect to the nondefault database
 EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT :db_name USING :db_string;
ENDFOR;
```

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

Implicit Logons

Implicit logons are supported through the Oracle distributed database option, which does not require explicit logons. For example, a distributed query allows a single SELECT statement to access data on one or more 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 run time, the embedded SELECT statement is executed by the specified Oracle Server, which connects implicitly to the nondefault database(s) to get the required data.

Single Implicit Logons

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

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

Then, the program can query the nondefault EMP table using the database link, as follows:

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

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

In our example, the default Oracle Server logs on to the nondefault database through SQL*Net using the database link *db_link*. The query is submitted to the default server, but is "forwarded" to the nondefault database for execution.

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

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

Then, your program can query the nondefault EMP table, as follows:

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

This provides location transparency for *emp*.

Multiple Implicit Logons

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

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

Then, your program can query the nondefault EMP and DEPT tables, as follows:

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

Oracle executes the query by performing a join between the nondefault EMP table at *db_link1* and the nondefault DEPT table at *db_link2*.

Embedding OCI (Oracle Call Interface) Calls

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

1. Declare the OCI Logon Data Area (LDA) outside the Declare Section.
2. Connect to Oracle using the embedded SQL statement `CONNECT`, not the OCI call `OLOG`.
3. Call the Oracle run-time library routine `SQLLDA` to store the connect information in the LDA.

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

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

Setting Up the LDA

You set up the LDA by issuing the OCI call

```
SQLLDA(lda);
```

where *lda* identifies the LDA data structure. The format of this call is language-dependent. If the `CONNECT` statement fails, the *lda_rc* field in the *lda* is set to 1012 to indicate the error.

Remote and Multiple Connections

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

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_string1 CHARACTER(20);
 db_string2 CHARACTER(20);
EXEC SQL END DECLARE SECTION;
lda1 INTEGER(32);
lda2 INTEGER(32);
set username = 'SCOTT';
set password = 'TIGER';
set db_string1 = 'D:NEWYORK-NONDEF1';
set db_string2 = 'D:CHICAGO-NONDEF2';
-- give each database connection a unique name
EXEC SQL DECLARE db_name1 DATABASE;
EXEC SQL DECLARE db_name2 DATABASE;
-- connect to first nondefault database
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name1 USING :db_string1;
-- set up first LDA for OCI use
SQLLDA(lda1);
-- connect to second nondefault database
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name2 USING :db_string2;
-- set up second LDA for OCI use
SQLLDA(lda2);
```

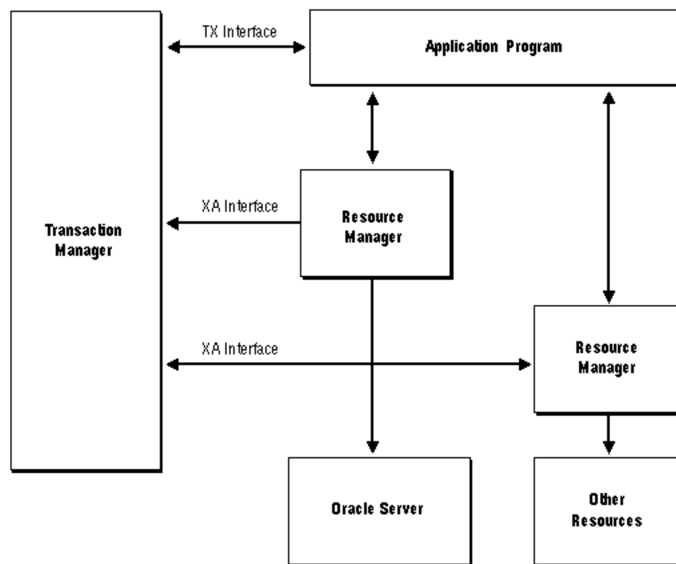
Remember, do not declare *db_name1* and *db_name2* in the Declare Section because they are not host variables. You use them only to name the default databases at the two nondefault nodes so that later SQL statements can refer to the databases by name.

Developing X/Open Applications

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

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

Figure 3–3 Hypothetical DTP Model



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

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

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

Oracle-Specific Issues

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

Connecting to Oracle

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

Transaction Control

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

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

OCI Calls

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

Linking

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

4
Using Embedded SQL

This chapter contains the following:

- Using Host Variables
- Using Indicator Variables
- The Basic SQL Statements
- Cursors
- Cursor Variables

This chapter helps you to understand and apply the basic techniques 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.

Using Host Variables

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

Output versus Input Host Variables

Depending on how they are used, host variables are called output or input host variables. Host variables in the `INTO` clause of a `SELECT` or `FETCH` statement are called *output* host variables because they hold column values output by Oracle. Oracle assigns the column values to corresponding output host variables in the `INTO` clause.

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

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;
```

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

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

```
EXEC SQL BEGIN DECLARE SECTION;
 table_name CHARACTER(30);
EXEC SQL END DECLARE SECTION;
display 'Table name? ';
read table_name;
EXEC SQL DROP TABLE :table_name; -- host variable not allowed
```

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

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 emp_name CHARACTER(20);
EXEC SQL END DECLARE SECTION;
-- get values for input host variables
display 'Employee number? ';
read emp_number;
display 'Employee name? ';
read emp_name;
EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
 VALUES (:emp_number, :emp_name);
```

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

Using Indicator Variables

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

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

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

```
set ind_comm = -1;
EXEC SQL INSERT INTO EMP (EMPNO, COMM)
 VALUES (:emp_number, :commission:ind_comm);
```

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

You can hardcode the null instead, as follows:

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

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

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

```
display 'Enter employee number or 0 if not available: ';
read emp_number;
IF emp_number = 0 THEN
 set ind_empnum = -1; ELSE
 set ind_empnum = 0;
ENDIF;
EXEC SQL INSERT INTO EMP (EMPNO, SAL)
 VALUES (:emp_number:ind_empnum, :salary);
```

Handling Returned Nulls

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

```
EXEC SQL SELECT ENAME, SAL, COMM
 INTO :emp_name, :salary, :commission:ind_comm
 FROM EMP
 WHERE EMPNO = :emp_number;
IF ind_comm = -1 THEN
 set pay = salary; -- commission is null; ignore it
ELSE
 set pay = salary + commission;
ENDIF;
```

Fetching Nulls

When DBMS=NATIVE, V7, or V8, if you select or fetch nulls into a host variable that lacks an indicator variable, Oracle issues the following error message:

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

Testing for Nulls

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

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

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

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

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

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

Fetching Truncated Values

If you select or fetch a truncated column value into a host variable that lacks an indicator variable, no error is generated.

The Basic SQL Statements

Executable SQL statements let you query, manipulate, and control Oracle data and create, define, and maintain Oracle objects such as tables, views, and indexes. This chapter focuses on data manipulation statements (sometimes called *DML*) and cursor control statements. The following SQL statements let you query and manipulate Oracle data:

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

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

- Implicit checking with the `WHENEVER` statement

- Explicit checking of SQLCA variables

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

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

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

Queries that return more than one row require an explicitly declared cursor or cursor variable (or the use of host arrays, which are discussed in Chapter 9, "Using Host Arrays"). The following embedded SQL statements let you define and control an explicit cursor:

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

In the coming sections, first you learn how to code INSERT, UPDATE, DELETE, and single-row SELECT statements. Then, you progress to multi-row SELECT statements.

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;
```

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

```
ORA-01422: exact fetch returns more than requested number of rows
```

When SELECT_ERROR=NO, a row is returned and Oracle generates no error.

Available Clauses

You can use all of the following standard SQL clauses in your SELECT statements: INTO, FROM, WHERE, CONNECT BY, START WITH, GROUP BY, HAVING, ORDER BY, and FOR UPDATE OF.

Inserting Rows

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

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

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

The number of values in the VALUES clause must equal the number of names in the column list. However, you can omit the column list if the VALUES clause contains a value for each column in the table in the same order they were defined by CREATE TABLE.

Using Subqueries

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

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

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;
```

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;
```

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

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;
```

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;
```

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

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

- Process beyond the first row returned by the query
- Keep track of which row is currently being processed

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

- DECLARE
- OPEN
- FETCH
- CLOSE

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

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

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

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

Declaring a Cursor

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

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

The cursor name is an identifier used by the precompiler, *not* a host or program variable, and should not be defined in the Declare Section. Therefore, cursor names cannot be passed from one precompilation unit to another. Also, cursor names cannot be hyphenated. They can be any length, but only the first 31 characters are significant. For ANSI compatibility, use cursor names no longer than 18 characters.

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

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

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

The cursor control statements must all occur within the same precompiled unit. For example, you cannot declare a cursor in file A, then open it in file B.

Your host program can declare as many cursors as it needs. However, in a given file, every DECLARE statement must be unique. That is, you cannot declare two cursors with the same name in one precompilation unit, even across blocks or procedures, because the scope of a cursor is global within a file. If you will be using many cursors, you might want to specify the MAXOPENCURSORS option. For more information, see "MAXOPENCURSORS".

Opening a Cursor

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

```
EXEC SQL OPEN emp_cursor;
```

OPEN positions the cursor just before the first row of the active set. It also zeroes the rows-processed count kept by SQLERRD(3) in the SQLCA. However, none of the rows is actually retrieved at this point. That will be done by the FETCH statement.

After you open a cursor, the query's input host variables are not reexamined until you reopen the cursor. Thus, the active set does not change. To change the active set, you must reopen the cursor.

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

The amount of work done by OPEN depends on the values of three precompiler options: HOLD_CURSOR, RELEASE_CURSOR, and MAXOPENCURSORS. For more

information, see "Using the Precompiler Options".

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 emp_cursor
 INTO :emp_name, :emp_number, :salary;
```

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

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

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

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL FROM EMP WHERE DEPTNO = 20;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name1, :salary1;
 EXEC SQL FETCH emp_cursor INTO :emp_name2, :salary2;
 EXEC SQL FETCH emp_cursor INTO :emp_name3, :salary3;
 ...
ENDLOOP;
```

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

Closing a Cursor

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

```
EXEC SQL CLOSE emp_cursor;
```

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

When MODE={ANSI13|ORACLE}, issuing a commit or rollback closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by a commit or rollback and if open, remain open. However, when MODE={ANSI|ANSI14}, issuing a commit or rollback closes *all* explicit cursors.

Using the CURRENT OF Clause

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

The FOR UPDATE OF clause is optional when you declare a cursor that is 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 "Using the FOR UPDATE OF Clause".

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

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL FROM EMP WHERE JOB = 'CLERK'
 FOR UPDATE OF SAL;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 ...
 EXEC SQL UPDATE EMP SET SAL = :new_salary
 WHERE CURRENT OF emp_cursor;
ENDLOOP;
```

Restrictions

An explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. All rows are locked at the open, not as they are fetched, and are released when you commit or rollback. If you try to fetch from a FOR UPDATE cursor after a commit, Oracle generates the following error:

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

You cannot use host arrays with the CURRENT OF clause. For an alternative, see "Mimicking the CURRENT OF Clause". Also, you cannot reference multiple tables in an associated FOR UPDATE OF clause, which means that you cannot do joins with the CURRENT OF clause. Finally, you cannot use the CURRENT OF clause in dynamic SQL.

A Typical Sequence of Statements

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

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

-- Open the cursor and identify the active set.
```

```
EXEC SQL OPEN emp_cursor;
-- Exit if the last row was already fetched.
EXEC SQL WHENEVER NOT FOUND DO no_more;

-- Fetch and process data in a loop.
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :job_title;
 -- host-language statements that operate on the fetched data
 EXEC SQL UPDATE EMP
 SET JOB = :new_job_title
 WHERE CURRENT OF emp_cursor;
ENDLOOP;
...
ROUTINE no_more
BEGIN
-- Disable the cursor.
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL COMMIT WORK RELEASE;
 exit program;
END no_more;
```

A Complete Example

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

All fetches except the final one return a row and, if no errors were detected during the fetch, a success status code. The final fetch fails and returns the "no data found" Oracle warning code to `SQLCODE` in the SQLCA. The cumulative number of rows actually fetched is found in `SQLERRD(3)` in the SQLCA.

```
-- declare host variables
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 emp_name CHARACTER(10);
 dept_number INTEGER;
EXEC SQL END DECLARE SECTION;
-- copy in the SQL Communications Area
EXEC SQL INCLUDE SQLCA;

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

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

-- log on to Oracle
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';

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

display 'Department number? ';
```

```
read dept_number;

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

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

display 'Employee Name';
display '-------------';

-- fetch and process data in a loop
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name; display emp_name;
ENDLOOP;
ROUTINE no_more
BEGIN
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL COMMIT WORK RELEASE;
 display 'End of program';
 exit program;
END no_more;

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

Cursor Variables

This section gives a brief overview of cursor variables. For more information, see your host language supplement and the *Oracle Database PL/SQL Language Reference*.

When using static embedded SQL with the Pro*COBOL and Pro*FORTRAN Precompilers, you can declare 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.

Also, you can assign new values to a cursor variable and pass it as a parameter to subprograms, including subprograms stored in an Oracle database. This gives you a convenient way to centralize data retrieval.

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

- `ALLOCATE`
- `OPEN ... FOR`
- `FETCH`
- `CLOSE`

After you declare the cursor variable and allocate memory for it, you must pass it as an input host variable (bind variable) to PL/SQL, OPEN it FOR a multi-row query on the server side, FETCH from it on the client side, then CLOSE it on either side.

Declaring a Cursor Variable

How you declare a cursor variable is dependent on your host language. For instructions about declaring a cursor variable, see your host-language supplement.

Allocating a Cursor Variable

You use the ALLOCATE statement to allocate memory for the cursor variable. The syntax follows:

```
EXEC SQL ALLOCATE <cursor_variable>;
```

Opening a Cursor Variable

You use the OPEN ... FOR statement to associate a cursor variable with a multi-row query, execute the query, and identify the active set. The syntax follows:

```
EXEC SQL OPEN <cursor_variable> FOR <select_statement>;
```

The SELECT statement can reference input host variables and PL/SQL variables, parameters, and functions but cannot be FOR UPDATE. In the following example, you open a cursor variable named *emp_cv*:

```
EXEC SQL OPEN emp_cv FOR SELECT * FROM EMP;
```

You must open a cursor variable on the server side. You do that by passing it as an input host variable to an anonymous PL/SQL block. At run time, the block is sent to the Oracle Server for execution. In the following example, you declare and initialize a cursor variable, then pass it to a PL/SQL block, which opens the cursor variable:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
-- declare cursor variable
 emp_cur SQL_CURSOR;
EXEC SQL END DECLARE SECTION;

-- initialize cursor variable
EXEC SQL ALLOCATE :emp_cur;

EXEC SQL EXECUTE
 -- pass cursor variable to PL/SQL block
 BEGIN
 -- open cursor variable
 OPEN :emp_cur FOR SELECT * FROM EMP;
 END;
END-EXEC;
```

Generally, you pass a cursor variable to PL/SQL by calling a stored procedure that declares a cursor variable as one of its formal parameters. For example, the following packaged procedure opens a cursor variable named *emp_cv*:

```
CREATE PACKAGE emp_data AS
 -- define REF CURSOR type
 TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
 -- declare formal paramter of that type
```

```
 PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp);
END emp_data;

CREATE PACKAGE BODY emp_data AS
 PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
 BEGIN
 -- open cursor variable
 OPEN emp_cv FOR SELECT * FROM emp;
 END open_emp_cv;
END emp_data;
```

You can call the procedure from any application, as follows:

```
EXEC SQL EXECUTE
 BEGIN
 emp_data.open_emp_cv(:emp_cur);
 END;
END-EXEC;
```

Fetching from a Cursor Variable

After opening a cursor variable for a multi-row query, you use the FETCH statement to retrieve rows from the active set one at a time. The syntax follows:

```
EXEC SQL FETCH cursor_variable_name
 INTO {record_name | variable_name[, variable_name, ...]};
```

Each column value returned by the cursor variable is assigned to a corresponding field or variable in the INTO clause, providing their datatypes are compatible.

The FETCH statement must be executed on the client side. In the following example, you fetch rows into a host record named *emp_rec*:

```
-- exit loop when done fetching
EXEC SQL WHENEVER NOT FOUND DO no_more;
LOOP
 -- fetch row into record
 EXEC SQL FETCH :emp_cur INTO :emp_rec;
 -- process the data
ENDLOOP;
```

Closing a Cursor Variable

You use the CLOSE statement to close a cursor variable, at which point its active set becomes undefined. The syntax follows:

```
EXEC SQL CLOSE cursor_variable_name;
```

The CLOSE statement can be executed on the client side or the server side. In the following example, when the last row is processed, you close the cursor variable *emp_cur*:

```
-- close cursor variable
EXEC SQL CLOSE :emp_cur;
```

5

Using Embedded PL/SQL

This chapter contains the following sections:

- Advantages of PL/SQL
- Embedding PL/SQL Blocks
- Using Host Variables
- Using Indicator Variables
- Using Host Arrays
- Using Cursors
- Stored Subprograms
- Using Dynamic PL/SQL

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

Advantages of PL/SQL

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

- Better Performance
- Integration with Oracle
- Cursor FOR Loops
- Subprograms
- Parameter Modes
- Packages
- PL/SQL Tables
- User-defined Records

Better Performance

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

Integration with Oracle

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

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

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

Cursor FOR Loops

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

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

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

Subprograms

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

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

```
PROCEDURE create_dept
 (new_dname IN CHAR(14),
 new_loc IN CHAR(13),
 new_deptno OUT NUMBER(2)) IS
BEGIN
 SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
 INSERT INTO dept VALUES (new_deptno, new_dname, new_loc);
END create_dept;
```

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

You can store subprograms in the database (using CREATE FUNCTION and CREATE PROCEDURE) that can be called from multiple applications without needing to be recompiled 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 `7IN` parameter lets you pass values to the subprogram being called. An `OUT` parameter lets you return values to the caller of a subprogram. An `IN OUT` parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

The datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Table 3–6 shows the legal conversions between datatypes.

Packages

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

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

```
PACKAGE emp_actions IS -- package specification
 PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);
 PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;
PACKAGE BODY emp_actions IS -- package body
 PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
 BEGIN
 INSERT INTO emp VALUES (empno, ename, ...);
 END hire_employee;
 PROCEDURE fire_employee (emp_id NUMBER) IS
 BEGIN
 DELETE FROM emp WHERE empno = emp_id;
 END fire_employee;
END emp_actions;
```

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

PL/SQL Tables

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

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

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

```
END;
```

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

```
num_tab NumTabTyp;
```

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

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

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

User-defined Records

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

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

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

```
DECLARE
 TYPE DeptRecTyp IS RECORD
 (deptno NUMBER(4) NOT NULL := 10, -- must initialize
 dname CHAR(9),
 loc CHAR(14));
```

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

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

```
dept_rec DeptRecTyp;
```

The identifier *dept_rec* represents an entire record.

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

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

Embedding PL/SQL Blocks

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

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

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

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

When your program embeds PL/SQL blocks, you must specify the precompiler option `SQLCHECK=SEMANTICS` because PL/SQL must be parsed by Oracle. To connect to Oracle, you must also specify the option `USERID`. For more information, see "Using the Precompiler Options".

Using Host Variables

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

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

Inside a PL/SQL block, host variables are treated as global to the entire block and can be used anywhere within the block wherever a PL/SQL variable is allowed. However, character host variables cannot exceed 255 characters in length. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.

An Example

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

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 emp_number INTEGER;
 job_title CHARACTER(20);
 hire_date CHARACTER(9);
 salary REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
```

```
EXEC SQL WHENEVER SQLERROR DO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
LOOP
 display 'Employee Number (0 to end)? ';
 read emp_number;
 IF emp_number = 0 THEN
 EXEC SQL COMMIT WORK RELEASE;
 display 'Exiting program';
 exit program;
 ENDIF;
 ---------------- begin PL/SQL block -----------------
 EXEC SQL EXECUTE
 BEGIN
 SELECT job, hiredate, sal
 INTO :job_title, :hire_date, :salary
 FROM emp
 WHERE empno = :emp_number;
 END;
 END-EXEC;
 ---------------- end PL/SQL block -----------------
 display 'Number Job Title Hire Date Salary';
 display '------------------------------------';
 display emp_number, job_title, hire_date, salary;
ENDLOOP;
...
ROUTINE sql_error
BEGIN
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
END sql_error;
```

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

A More Complex Example

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

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 acct_num INTEGER;
 trans_type CHARACTER(1);
 trans_amt REAL;
 status CHARACTER(80);
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR DO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
LOOP
```

```
 display 'Account Number (0 to end)? ';
 read acct_num;
 IF acct_num = 0 THEN
 EXEC SQL COMMIT WORK RELEASE;
 display 'Exiting program';
 exit program;
 ENDIF;
 display 'Transaction Type - D)ebit or C)redit? '
 read trans_type;
 display 'Transaction Amount? '
 read trans_amt;
 --------------------- begin PL/SQL block -------------------
 EXEC SQL EXECUTE
 DECLARE
 old_bal NUMBER(9,2);
 err_msg CHAR(70);
 nonexistent EXCEPTION;
 BEGIN
 :trans_type := UPPER(:trans_type);
 IF :trans_type = 'C' THEN -- credit the account
 UPDATE accts SET bal = bal + :trans_amt
 WHERE acctid = :acct_num;
 IF SQL%ROWCOUNT = 0 THEN -- no rows affected
 RAISE nonexistent;
 ELSE
 :status := 'Credit applied';
 END IF;
 ELSIF :trans_type = 'D' THEN -- debit the account
 SELECT bal INTO old_bal FROM accts
 WHERE acctid = :acct_num;
 IF old_bal >= :trans_amt THEN -- enough funds
 UPDATE accts SET bal = bal - :trans_amt
 WHERE acctid = :acct_num;
 :status := 'Debit applied';
 ELSE
 :status := 'Insufficient funds';
 END IF;
 ELSE
 :status := 'Invalid type: ' || :trans_type;
 END IF;
 COMMIT;
 EXCEPTION
 WHEN NO_DATA_FOUND OR nonexistent THEN
 :status := 'Nonexistent account';
 WHEN OTHERS THEN
 err_msg := SUBSTR(SQLERRM, 1, 70);
 :status := 'Error: ' || err_msg;
 END;
 END-EXEC;
 ------------------- end PL/SQL block -----------------------
 display 'Status: ', status;
ENDLOOP;
ROUTINE sql_error
BEGIN
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
END sql_error;
```

VARCHAR Pseudotype

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

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

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 emp_name VARCHAR(10);
 salary REAL;
 ...
EXEC SQL END DECLARE SECTION;
...
set emp_name.len = 10; -- initialize length field
EXEC SQL EXECUTE
 BEGIN
 SELECT ename, sal INTO :emp_name, :salary
 FROM emp
 WHERE empno = :emp_number;
 ...
 END;
END-EXEC;
```

Using Indicator Variables

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

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

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

```
variable := NULL;
```

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

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

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

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

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

```
EXEC SQL EXECUTE
 BEGIN
```

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

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

Handling Nulls

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

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

Handling Truncated Values

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

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

Using Host Arrays

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

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

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

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

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 salary (100) REAL;
 median_salary REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host array
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, n);
 ...
 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 array.

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

Table 5–1 Legal Conversions: PL/SQL Table Row and Host Array Elements

PL/SQL Table	Host Array							
	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	_/							
CHARZ	_/							
DATE		_/						
DECIMAL					_/			

Table 5–1 Legal Conversions: PL/SQL Table Row and Host Array Elements

PL/SQL Table	Host Array						
DISPLAY				_/			
FLOAT				_/			
INTEGER				_/			
LONG	_/	_/					
LONG VARCHAR		_/	_/		_/		_/
LONG VARRAW			_/		_/		
NUMBER				_/			
RAW			_/		_/		
ROWID						_/	
STRING		_/	_/		_/		_/
UNSIGNED				_/			
VARCHAR		_/	_/		_/		_/
VARCHAR2		_/	_/		_/		_/
VARNUM				_/			
VARRAW			_/		_/		

ARRAYLEN Statement

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

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

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

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

In the following example, you use ARRAYLEN to override the default dimension of a host array named *bonus*:

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

```
 RETURN REAL IS
 BEGIN
 -- compute median
 END;
 BEGIN
 median_bonus := median(:bonus, :my_dim);
 ...
 END;
END-EXEC;
```

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

Using Cursors

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

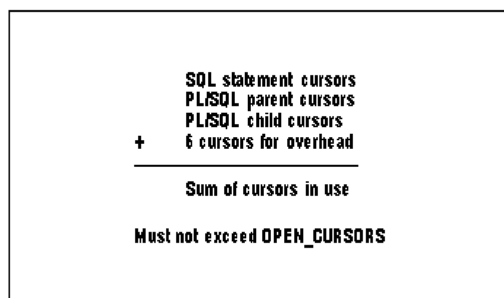
Various precompiler options, including MAXOPENCURSORS, HOLD_CURSOR, and RELEASE_CURSOR, let you manage the cursor cache to improve performance. For example, RELEASE_CURSOR controls what happens to the link between the cursor cache and private SQL area. If you specify RELEASE_CURSOR=YES, the link is removed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

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

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

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

Figure 5–1 Maximum Cursors in Use



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

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

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

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

An Alternative

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

Stored Subprograms

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

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

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

Creating Stored Subprograms

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

```
EXEC SQL CREATE
 FUNCTION sal_ok (salary REAL, title CHAR)
 RETURN BOOLEAN AS
 min_sal REAL;
 max_sal REAL;
 BEGIN
 SELECT losal, hisal INTO min_sal, max_sal
 FROM sals
```

```
 WHERE job = title;
 RETURN (salary >= min_sal) AND
 (salary <= max_sal);
 END sal_ok;
END-EXEC;
```

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

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

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

```
EXEC SQL CREATE OR REPLACE PACKAGE emp_actions AS
 TYPE CharArrayTyp IS TABLE OF VARCHAR2(10)
 INDEX BY BINARY_INTEGER;
 TYPE NumArrayTyp IS TABLE OF FLOAT
 INDEX BY BINARY_INTEGER;
 PROCEDURE get_employees(
 dept_number IN INTEGER,
 batch_size IN INTEGER,
 found IN OUT INTEGER,
 done_fetch OUT INTEGER,
 emp_name OUT CharArrayTyp,
 job-title OUT CharArrayTyp,
 salary OUT NumArrayTyp);
 END emp_actions;
END-EXEC;
EXEC SQL CREATE OR REPLACE PACKAGE BODY emp_actions AS
 CURSOR get_emp (dept_number IN INTEGER) IS
 SELECT ename, job, sal FROM emp
 WHERE deptno = dept_number;
 PROCEDURE get_employees(
 dept_number IN INTEGER,
 batch_size IN INTEGER,
 found IN OUT INTEGER,
 done_fetch OUT INTEGER,
 emp_name OUT CharArrayTyp,
 job_title OUT CharArrayTyp,
 salary OUT NumArrayTyp) IS
 BEGIN
 IF NOT get_emp%ISOPEN THEN
 OPEN get_emp(dept_number);
 END IF;
 done_fetch := 0;
 found := 0;
 FOR i IN 1..batch_size LOOP
 FETCH get_emp INTO emp_name(i),
 job_title(i), salary(i);
 IF get_emp%NOTFOUND THEN
 CLOSE get_emp;
 done_fetch := 1;
 EXIT;
```

```
 ELSE
 found := found + 1;
 END IF;
 END LOOP;
 END get_employees;
 END emp_actions;
END-EXEC;
```

You specify the `REPLACE` clause in the `CREATE` statement to redefine an existing package without having to drop the package, re-create it, and regrant privileges on it. For the full syntax of the `CREATE` statement see the *Oracle Database SQL Language Reference*.

If an embedded `CREATE {FUNCTION|PROCEDURE|PACKAGE}` statement fails, Oracle generates a warning, not an error.

Calling a Stored Subprogram

To invoke (call) a stored subprogram from your host program, you must use an anonymous PL/SQL block. In the following example, you call a 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 array, 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.

In the Pro*C example, three of the formal parameters are PL/SQL tables, and the corresponding actual parameters are host arrays. The program calls the stored procedure *get_employees* repeatedly, displaying each batch of employee data, until no more data is found.

```
#include <stdio.h>
#include <string.h>
typedef char asciz;
EXEC SQL BEGIN DECLARE SECTION;
 /* Define type for null-terminated strings */
 EXEC SQL TYPE asciz IS STRING(20);
 asciz username[20];
 asciz password[20];
 int dept_no; /* which department to query */
 char emp_name[10][21];
 char job[10][21];
 float salary[10];
```

```
 int done_flag;
 int array_size;
 int num_ret; /* number of rows returned */
 int SQLCODE;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE sqlca;
int print_rows(); /* produces program output */
int sql_error(); /* handles NOLOGGING errors */
main()
{
 int i;
 /* Connect to Oracle. */
 strcpy(username, "SCOTT");
 strcpy(password, "TIGER");
 EXEC SQL WHENEVER SQLERROR DO sql_error();
 EXEC SQL CONNECT :username IDENTIFIED BY :password;
 printf("\nConnected to Oracle as user: %s\n", username);
 printf("enter department number: ");
 scanf("%d", &dept_no);
 fflush(stdin);
 /* Set the array size. */
 array_size = 10;
 done_flag = 0;
 num_ret = 0;
 /* Array fetch loop - ends when done_flag is true. */
 for (;;)
 {
 EXEC SQL EXECUTE
 BEGIN emp_actions.get_employees
 (:dept_no, :array_size, :num_ret,
 :done_flag, :emp_name, :job, :salary);
 END;
 END-EXEC;
 print_rows(num_ret);
 if (done_flag)
 break;
 }
 /* Disconnect from the database. */
 EXEC SQL COMMIT WORK RELEASE;
 exit(0);
}
print_rows(n)
int n;
{
 int i;
 if (n == 0)
 {
 printf("No rows retrieved.\n");
 return;
 }
 printf("\n\nGot %d row%c\n", n, n == 1 ? '\0' : 's');
 printf("%-20.20s%-20.20s%s\n", "Ename", "Job", "Salary");
 for (i = 0; i < n; i++)
 printf("%20.20s%20.20s%6.2f\n",
 emp_name[i], job[i], salary[i]);
}
sql_error()
{
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 printf("\nOracle error detected:");
```

```
 printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
 EXEC SQL ROLLBACK WORK RELEASE;
 exit(1);
}
```

Remember, the datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Also, before a stored subprogram exits, all OUT formal parameters must be assigned values. Otherwise, the values of corresponding actual parameters are indeterminate.

Remote Access

PL/SQL lets you access remote databases through *database links*. Typically, database links are established by your DBA and stored in the Oracle data dictionary. A database link tells Oracle where the remote database is located, the path to it, and what Oracle username and password to use. In the following example, you use the database link *dallas* to call the *raise_salary* procedure:

```
EXEC SQL EXECUTE
 BEGIN
 raise_salary@dallas(:emp_id, :increase);
 END;
END-EXEC;
```

You can create synonyms to provide location transparency for remote subprograms, as the following example shows:

```
CREATE PUBLIC SYNONYM raise_salary FOR raise_salary@dallas;
```

Getting Information about Stored Subprograms

In Chapter 3, "Meeting Program Requirements", you learned how to embed OCI calls in your host program. After calling the library routine SQLLDA to set up the LDA, you can use the OCI call ODESSP to get useful information about a stored subprogram. When you call ODESSP, you must pass it a valid LDA and the name of the subprogram. For packaged subprograms, you must also pass the name of the package. ODESSP returns information about each subprogram parameter such as its datatype, size, position, and so on.

You can also use the procedure *describe_procedure* in package DBMS_DESCRIBE, which is supplied with Oracle.

Using Dynamic PL/SQL

Recall that the Oracle Precompilers treat an entire PL/SQL block like a single SQL statement. Therefore, you can store a PL/SQL block in a string host variable. Then, if the block contains no host variables, you can use dynamic SQL Method 1 to execute the PL/SQL string. Or, if the block contains a known number of host variables, you can use dynamic SQL Method 2 to prepare and execute the PL/SQL string. If the block contains an unknown number of host variables, you must use dynamic SQL Method 4. For more information, refer to Chapter 10, "Using Dynamic SQL".

Restriction

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

6 Running the Oracle Precompilers

This chapter contains the following:

- The Precompiler Command
- What Occurs during Precompilation?
- Precompiler Options
- Entering Options
- Scope of Options
- Quick Reference
- Using the Precompiler Options
- Conditional Precompilations
- Separate Precompilations
- Compiling and Linking

This chapter details the requirements for running the Oracle Precompilers. You learn what occurs during precompilation, how to issue the precompiler command, how to specify the many useful precompiler options, and how to do conditional and separate precompilations.

The Precompiler Command

To run an Oracle Precompiler, you issue one of the language-specific commands shown in Table 6–1.

Table 6–1 Precompiler Run Commands

Host Language	Precompiler Command
COBOL	`procob`
FORTRAN	`profor`

The location of the precompiler differs from system to system. Typically, your system manager or DBA defines environment variables, logicals, or aliases or uses other operating system-specific means to make the precompiler executable accessible.

The INAME option specifies the source file to be precompiled. For example, the Pro*COBOL command

```
procob INAME=test
```

precompiles the file *test.pco* in the current directory, since the precompiler 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, the precompiler 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 Pro*FORTRAN command

```
profor MODE=ANSI myfile.pfo DBMS=V7 myfile.f
```

is equivalent to

```
profor MODE=ANSI INAME=myfile.pfo DBMS=V7 ONAME=myfile.f
```

> **Note:** Option names and option values that do not name specific operating system objects, such as filenames, are not case-sensitive. In the examples in this guide, option names are written in upper case, and option values are usually in lowercase. Filenames, including the name of the precompiler executable itself, always follow the case conventions used by the operating system on which it is executed.

What Occurs during Precompilation?

During precompilation, an Oracle Precompiler generates host-language 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, and other information required by the Oracle run-time library, SQLLIB. The generated code also contains the calls to SQLLIB routines that perform the embedded SQL operations.

The generated code also includes calls to the SQLLIB routines that perform embedded SQL operations. Note that the precompiler does *not* generate calls to Oracle Call Interface (OCI) routines.

The precompiler does *not* generate calls to Oracle Call Interface (OCI) routines.

The precompiler can issue warnings and error messages. These messages have the prefix PCC-, and are described in *Oracle Database Error Messages*.

Precompiler Options

Many useful options are available at precompile time. They let you control how resources are used, how errors are reported, how input and output are formatted, and how cursors are managed. To specify a precompiler option, use the following syntax:

```
<option_name>=<value>
```

The *value* of an option is a string literal, which represents text or numeric values. For example, for the option

```
... INAME=my_test
```

the value is a string literal that specifies a filename, but for the option

```
... MAXOPENCURSORS=20
```

the value is numeric.

Some options take Boolean values, which you can represent with the strings YES or NO, TRUE or FALSE, or with the integer literals 1 or 0, respectively. For example, the option

```
... SELECT_ERROR=YES
```

is equivalent to

```
... SELECT_ERROR=TRUE
```

or

```
... SELECT_ERROR=1
```

The option value is always separated from the option name by an equal sign, leave no whitespace around the equal sign, because spaces delimit individual options. For example, you might specify the option `AUTO_CONNECT` on the command line as follows:

```
... AUTO_CONNECT=YES
```

You can abbreviate the names of options if the abbreviation is unambiguous. For example, you cannot use the abbreviation MAX because it might stand for MAXLITERAL or MAXOPENCURSORS.

A handy reference to the precompiler options is available online. To see the online display, enter the precompiler command with no arguments at your operating system prompt. The display gives the name, syntax, default value, and purpose of each option. Options marked with an asterisk (*) can be specified inline and on the command line.

Default Values

Many of the options have default values, which are determined by:

- A value built in to the precompiler
- A value set in the *system* configuration file
- A value set in a *user* configuration file
- A value set in an inline specification

For example, the option MAXOPENCURSORS specifies the maximum number of cached open cursors. The built-in precompiler default value for this option is 10. However, if MAXOPENCURSORS=32 is specified in the system configuration file, the default becomes 32. The user configuration file could set it to yet another value, which then overrides the system configuration value.

Then, if this option is set on the command line, the new command-line value takes precedence. Finally, an inline specification takes precedence over all preceding defaults. For more information, refer to Configuration Files".

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 Pro*COBOL command

```
procob ?
```

the complete option set, along with current values, is displayed on your terminal. In this case, the values are those built into the precompiler, 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 the precompiler or values specified in the system configuration file.

You can also determine the current value of a single option by simply specifying the option name followed by "=?" as in

```
procob MAXOPENCURSORS=?
```

> **Note:** : With some operating systems, the "?" may need to be preceded by an "escape" character, such as a back-slash (\). For example, instead of "procob ?," you might need to use "procob \?" to list the Pro*COBOL option settings.

Case Sensitivity

In general, you can use either uppercase or lowercase for command-line option names and values. However, if your operating system is case-sensitive, like UNIX, you must specify filename values, including the name of the precompiler executable, using the correct combination of upper and lowercase letters.

Configuration Files

A configuration file is a text file that contains precompiler options. Each record (line) in the file contains one option, with its associated value or values. For example, a configuration file might contain the lines

```
FIPS=YES
MODE=ANSI
```

to set defaults for the `FIPS` and `MODE` options.

There is a single system configuration file for each system. The name of the system configuration file is precompiler-specific and is shown in Table 6–2.

Table 6–2 System Configuration Files

Precompiler	Configuration File
Pro*COBOL	`pcccob.cfg`
Pro*FORTRAN	`pccfor.cfg`

The location of the file is operating system-specific. On most UNIX systems, the Pro*COBOL configuration file is usually located in the *$ORACLE_HOME/procob* directory, and the Pro*FORTRAN equivalent is in the *$ORACLE_HOME/profor* directory, where *$ORACLE_HOME* is the environment variable for the database software.

Each precompiler 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, refer to Determining Current Values.

> **Note:** You cannot nest configuration files. CONFIG is not a valid option inside a configuration file.

Entering Options

All the precompiler options can be entered on the command line or (except `CONFIG`) from a configuration file. Many options can also be entered inline. During a given run, the precompiler can accept options from all three sources.

On the Command Line

You enter precompiler options on the command line using the following syntax:

```
... [option_name=value] [option_name=value] ...
```

Separate each option with one or more spaces. For example, you might enter the following options:

```
... ERRORS=no LTYPE=short
```

Inline

You enter options inline by coding `EXEC ORACLE` statements, using the following syntax:

```
EXEC ORACLE OPTION (option_name=value);
```

For example, you might code the following statement:

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES);
```

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 C shows you how to use inline options to optimize run-time performance.

Specifying options inline is also helpful if your operating system limits the number of characters you can enter on the command line, and you can store inline options in configuration files, which are discussed in the next section.

Scope of EXEC ORACLE

An EXEC ORACLE statement stays in effect until textually superseded by another EXEC ORACLE statement specifying the same option. In the following example, `HOLD_CURSOR=NO` stays in effect until superseded by `HOLD_CURSOR=YES`:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name CHARACTER(20);
 emp_number INTEGER;
 salary REAL;
 dept_number INTEGER;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
EXEC ORACLE OPTION (HOLD_CURSOR=NO);
```

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT EMPNO, DEPTNO FROM EMP;
EXEC SQL OPEN emp_cursor;
display 'Employee Number Dept';
display '--------------- ----';
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_number, :dept_number;
 display emp_number, dept_number;
ENDLOOP;
no_more:
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
 LOOP
 display 'Employee number? ';
 read emp_number;
 IF emp_number = 0 THEN
 exit loop;
 EXEC ORACLE OPTION (HOLD_CURSOR=YES);
 EXEC SQL SELECT ENAME, SAL
 INTO :emp_name, :salary
 FROM EMP
 WHERE EMPNO = :emp_number;
 display 'Salary for ', emp_name, ' is ', salary;
 ENDLOOP;
...
```

From a Configuration File

The Oracle Precompilers can use a configuration file containing preset command-line options. By default, a text file called the *system configuration file* is used. However, you can specify any of several alternative files, called *user configuration files*, on the command line.

Advantages

Configuration files offer several advantages. The system configuration file lets you standardize a set of options for all projects. User configuration files let you customize a set of options for each project. With configuration files, you need not enter long strings of options on the command line. Also, if your system limits the length of a command line, configuration files let you specify more options than the command line can hold.

Using Configuration Files

Each record (line) in a configuration file holds one command-line option. For example, a configuration file might contain the following lines, which set defaults for the `FIPS`, `MODE`, and SQLCHECK options:

```
FIPS=YES
MODE=ANSI
SQLCHECK=SEMANTICS
```

Each Oracle Precompiler can have its own system configuration file. The name and location of the file are language- and system-specific. If the file is not found, you get a warning but the precompiler continues processing.

There is only one system configuration file for a given language, but you can create any number of user configuration files. You use the new command-line option CONFIG to specify the name and location of a particular user configuration file, as follows:

```
... CONFIG=<filename>
```

You cannot nest configuration files. Therefore, you cannot specify the CONFIG option in a configuration file. Also, you cannot specify CONFIG inline.

Setting Option Values

Many precompiler run-time options have built-in default values, which can be reset in a configuration file or on the command line. Command-line settings override user configuration file settings, which override system configuration file settings.

Scope of Options

A precompilation unit is a file containing host-language 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 6–3 is a quick reference to the precompiler options. The options marked with an asterisk can be entered inline.

Another handy reference is available online. To see the online display, just enter the precompiler command without options at your operating system prompt. The display provides the name, syntax, default value, and purpose of each option.

There are some platform-specific options. For example, on byte-swapped platforms that use MicroFocus COBOL, the option COMP5 governs the use of certain COMPUTATIONAL items. Check your system-specific Oracle manuals.

Table 6–3 Precompiler Options Quick Reference

Syntax	Default	Specifies ...
ASACC={YES | NO}	NO	carriage control for listing
ASSUME_SQLCODE={YES | NO}	NO	precompiler presumes that SQLCODE is declared
AUTO_CONNECT={YES | NO}	NO	automatic logon
CHAR_MAP={VARCHAR2 | CHARZ | STRING | CHARF} *	CHARZ	mapping of character arrays and strings
CHARSET_PICN={NCHAR_ CHARSET | DB_CHARSET }	NCHAR _ CHARSE T	the character set form used by PIC N variables
CHARSET_PICX={NCHAR_ CHARSET | DB_CHARSET }	DB_ CHARSE T	the character set form used by PIC X variables

Table 6–3 Precompiler Options Quick Reference

Syntax	Default	Specifies ...
CINCR	1	CINCR value for connection pool. Allows the application to set the next increment for physical connections to be opened to the database, if the current number of physical connections is less than CMAX
CLOSE_ON_COMMIT={YES | NO}	NO	close all cursors on COMMIT
CMAX	100	specifies the maximum number of physical connections that can be opened for the database
CMIN	2	specifies the minimum number of physical connections in the connection pool.
CNOWAIT	0 (which means not set)	determines if the application must repeatedly try for a physical connection when all other physical connections in the pool are busy, and the total number of physical connections has already reached its maximum. CNOWAIT Value for connection pool
CODE={ANSI_C | KR_C | CPP}	KR_C	type of C code to be generated
COMMON_NAME=*block_name* *		name of FORTRAN COMMON blocks
COMMON_PARSER	NO	parse using Common SQL Front End
COMP5	YES	generate COMP-5 rather than COMP variables
COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}	MULTI_ BYTE	the character set type the C/C++ compiler supports.
CONFIG=*filename*		name of user configuration file
CPOOL	NO	support connection pooling. Based on this option, the precompiler generates the appropriate code that directs SQLLIB to enable or disable the connection pool feature
CPP_SUFFIX=extension	*none*	override the default C++ filename extension
CTIMEOUT	0	physical connections that are idle for more than the specified time (in seconds) are terminated to maintain an optimum number of open physical connections
DB2_ARRAY={YES |NO}	NO	support DB2 array insert/select syntax. Based on this option, the precompiler activates the additional array insert/select syntax
DBMS={NATIVE|V7|V8}	NATIVE	version-specific behavior of Oracle at precompile time
DECLARE_SECTION	NO	if YES, DECLARE SECTION is required
DEF_SQLCODE={NO | YES}	NO	controls whether the Pro*C/C++ precompiler generates #define's for SQLCODE
DEFINE=*symbol* *		symbol used in conditional precompilation
DURATION={TRANSACTION | SESSION}	TRANSA CTION	set pin duration for objects in the cache
DYNAMIC={ANSI | ORACLE}	ORACLE	specify Oracle or ANSI SQL semantics.
END_OF_FETCH	1403	end-of-fetch SQLCODE value

Table 6–3 Precompiler Options Quick Reference

Syntax	Default	Specifies ...
ERRORS={YES I NO} *	YES	whether errors are sent to the terminal
ERRTYPE=*filename*	*none*	name of the list file for intype file errors
EVENTS	NO	support publish-subscribe event notifications
FILE_ID	0	unique numeric identifier for the generated COBOL file
FIPS={YES I NO}*	NO	whether ANSI/ISO extensions are flagged
FORMAT={ANSI I TERMINAL}	ANSI	format of COBOL or FORTRAN input line
Globalization Support_ LOCAL={YES I NO}	YES	blank-padding operations to be preformed by SQLLIB
HEADER=*extension*	*none*	name of the listing file for intype file error messages
HOLD_CURSOR={YES I NO}*	NO	how cursor cache handles SQL statements
HOST={COBOL I COB74}	COBOL	COBOL version of input file
IMPLICIT_SVPT	NO	implicit savepoint before buffered insert
[INAME=]*filename*		name of input file
INCLUDE=*path**		directory path for INCLUDEd files
INTYPE=*filename*	*none*	name of the input file for type information
IRECLEN=*integer*	80	record length of input file
LINES={YES I NO}	NO	whether #line directives are generated
LITDELIM={APOST I QUOTE} *	QUOTE	delimiter for COBOL strings
LNAME=*filename*		name of listing file
LRECLEN=*integer*	132	record length of listing file
LTYPE={LONG I SHORT I NONE}	LONG	type of listing
MAXLITERAL=*integer* *	platform-specific	maximum length of strings
MAXOPENCURSORS=*integer* *	10	maximum number of cursors cached
MAX_ROW_INSERT	0	maximum number of rows to buffer on insert
MODE={ORACLE I ANSI I ANSI14 I ANSI13}	ORACLE	compliance with the ANSI/ISO SQL standard
MULTISUBPROG={YES I NO}	YES	whether FORTRAN COMMON blocks are generated
NATIVE_TYPES	NO	support for native float/double
NESTED={YES I NO}	YES	if YES, nested programs are supported
NLS_CHAR=(*var1*, ..., *varn*)	*none*	specify multibyte character variables
NLS_LOCAL={YES I NO}	NO	if YES, use NCHAR semantics of previous Pro*COBOL releases
OBJECTS={YES I NO}	YES	Support of object types
OUTLINE	NO	category in which Outlines are created
OUTLNPREFIX	*none*	outline name prefix
[ONAME=]*filename*		name of output file

Table 6–3 Precompiler Options Quick Reference

Syntax	Default	Specifies ...
ORACA={YES|NO}*	NO	whether the ORACA is used
ORECLEN=*integer*	80	record length of output file
PAGELEN=*integer*	66	lines in each page in listing
PARSE={NONE | PARTIAL | FULL}	FULL	whether Pro*C/C++ parses (with a C parser) the.pc source
PICN_ENDIAN	BIG	endianness in PIC N host variables
PICX	CHARF	datatype of PIC X COBOL variables.
PREFETCH=0..65535	1	speed up queries by pre-fetching a given number of rows
RELEASE_CURSOR={YES|NO} *	NO	how cursor cache handles SQL statements
RUNOUTLINE	NO	create Outlines in the database
SELECT_ERROR={YES|NO}*	YES	how SELECT errors are handled
SQLCHECK={FULL|SYNTAX|LIMITED|NONE}*	SYNTAX	extent of syntactic and semantic checking
STMT_CACHE	0	size of statement cache
SYS_INCLUDE=pathname	none	directory where system header files, such as iostream.h, are found
THREADS={YES | NO}	NO	indicates a shared server application
TYPE_CODE={ORACLE | ANSI}	ORACLE	use of Oracle or ANSI type codes for dynamic SQL
UNSAFE_NULL={YES|NO}	NO	disables the ORA-01405 message
USERID=*username/password*		valid Oracle username and password
UTF16_CHARSET={NCHAR_CHARSET | DB_CHARSET}	NCHAR_CHARSET	specify the character set form used by UNICODE(UTF16)
VARCHAR={YES|NO}	NO	recognize implicit VARCHAR group items in COBOL
VERSION={ANY | LATEST | RECENT} *	RECENT	Which version of an object is to be returned
XREF={YES|NO}*	YES	cross reference section in listing

Using the Precompiler Options

This section is organized for easy reference. It lists the precompiler options alphabetically, and for each option provides 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 the Oracle Precompiler to presume that SQLCODE is declared irrespective of whether it is declared in the Declare Section or of the proper type. ASSUME_SQLCODE=YES causes Releases 1.6 and later of the Oracle Precompilers to behave similarly to Release 1.5 in this respect.

Syntax

```
ASSUME_SQLCODE={YES|NO}
```

Default

NO

Usage Notes

Cannot be entered inline.

When `ASSUME_SQLCODE=NO`, `SQLCODE` is recognized as a status variable if and only if at least one of the following criteria is satisfied:

- It is declared in a Declare Section with *exactly* the right datatype.
- The precompiler finds no other status variable.

If the precompiler finds a SQLSTATE declaration (of *exactly* the right type of course) in a Declare Section or finds an `INCLUDE` of the SQLCA, it will *not* presume SQLCODE is declared.

When `ASSUME_SQLCODE=YES`, and when SQLSTATE and SQLCA (Pro*FORTRAN only) are declared as status variables, the precompiler presumes SQLCODE is declared whether it is declared in a Declare Section or of the proper type. This causes Releases 1.6.7 and later to behave like Release 1.5 in this regard.

AUTO_CONNECT

Purpose

Specifies whether your program connects automatically to the default user account.

Syntax

AUTO_CONNECT={YES|NO}

Default

NO

Usage Note

Cannot be entered inline.

When AUTO_CONNECT=YES, as soon as the precompiler 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 null) 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 log on to Oracle.

CHAR_MAP

Purpose

Specifies the default mapping of C host variables of type char or char[n], and pointers to them, into SQL.

Syntax

CHAR_MAP={VARCHAR2 | CHARZ | STRING | CHARF}

Default

CHARZ

Usage Note

In earlier releases, you had to declare char or char[n] host variables as CHAR, using the SQL DECLARE statement. The external datatypes VARCHAR2 and CHARZ were the default character mappings of Oracle Database version 7.

CINCR

Purpose

Allows the application to set the next increment for physical connections to be opened to the database.

Syntax

CINCR = Range is 1 to (CMAX-CMIN).

Default

1

Usage Notes

Initially, all physical connections as specified through CMIN are opened to the server. Subsequently, physical connections are opened only when necessary. Users should set CMIN to the total number of planned or expected concurrent statements to be run by the application to get optimum performance. The default value is set to 2.

CLOSE_ON_COMMIT

Purpose

Specifies whether to close cursors on a commit statement.

Syntax

CLOSE_ON_COMMIT={YES | NO}

Default

NO

Usage Notes

Can be used only on the command line or in a configuration file.

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 will be closed on commit.

When CLOSE_ON_COMMIT=NO (when MODE=ORACLE), issuing a COMMIT or ROLLBACK will close only cursors that are declared using the FOR UPDATE clause or are referenced in a CURRENT OF clause. Other cursors that are not affected by the COMMIT or ROLLBACK statement, remain open, if they are open already. However, when CLOSE_ON_COMMIT=YES (when MODE=ANSI), issuing a COMMIT or ROLLBACK closes all cursors.

CMAX

Purpose

Specifies the maximum number of physical connections that can be opened for the database.

Syntax

CINCR = Range is 1 to 65535

Default

100

Usage Notes

CMAX value must be at least CMIN+CINCR.After this value is reached, more physical connections cannot be opened.In a typical application, running 100 concurrent database operations is more than sufficient. The user can set an appropriate value.

CMIN

Purpose

Specifies the minimum number of physical connections that can be opened for the database.

Syntax

CINCR = Range is 1 to (CMAX-CINCR).

Default

2

Usage Notes

CMAX value must be at least CMIN+CINCR.After this value is reached, more physical connections cannot be opened.In a typical application, running 100 concurrent database operations is more than sufficient. The user can set an appropriate value.

CNOWAIT

Purpose

This attribute determines if the application must repeatedly try for a physical connection when all other physical connections in the pool are busy, and the total number of physical connections has already reached its maximum.

Syntax

CNOWAIT = Range is 1 to 65535.

Default

0 which means not set.

Usage Notes

If physical connections are not available and no more physical connections can be opened, an error is thrown when this attribute is set. Otherwise, the call waits until it acquires another connection. By default, CNOWAIT is not to be set so a thread will wait until it can acquire a free connection, instead of returning an error.

CODE

Purpose

Specifies the format of C function prototypes generated by the Pro*C/C++ precompiler. (A *function prototype* declares a function and the datatypes of its arguments.) The precompiler generates function prototypes for SQL library routines, so that your C compiler can resolve external references. The CODE option lets you control the prototyping.

Syntax

CODE={ANSI_C | KR_C | CPP}

Default

KR_C

Usage Notes

Can be entered on the command line, but not inline.

ANSI C standard X3.159-1989 provides for function prototyping. When CODE=ANSI_C, Pro*C/C++ generates full function prototypes, which conform to the ANSI C standard. An example follows:

```
extern void sqlora(long *, void *);
```

The precompiler can also generate other ANSI-approved constructs such as the **const** type qualifier.

When CODE=KR_C (the default), the precompiler comments out the argument lists of generated function prototypes, as shown here:

```
extern void sqlora(/*_ long *, void * _*/);
```

Specify CODE=KR_C if your C compiler is not compliant with the X3.159 standard.

When CODE=CPP, the precompiler generates C++ compatible code.

COMMON_NAME

Purpose

For Pro*FORTRAN only, the `COMMON_NAME` option specifies a prefix used to name internal FORTRAN `COMMON` blocks. Your host program does not access the `COMMON` blocks directly. But, they allow two or more program units in the same precompilation unit to contain SQL statements.

Syntax

`COMMON_NAME=`*`blockname`*

Default

First five characters in name of input file

Usage Notes

The Pro*FORTRAN Precompiler uses a special program file called a *block data subprogram* to establish `COMMON` blocks for all the SQL variables in an input file. The block data subprogram defines two COMMON blocks -- one for CHARACTER variables, the other for non-CHARACTER variables -- and uses DATA statements to initialize the variables.

The format of a block data subprogram follows:

```
BLOCK DATA <subprogram_name>
variable declarations
COMMON statements
DATA statements
END
```

Your host program does not access the `COMMON` blocks directly. But, they allow two or more program files in the same precompilation file to contain SQL statements.

To name the COMMON blocks, the precompiler uses the name of the input file and the suffixes C, D, and I. At most, the first five characters of the filename are used. For example, if the name of the input file is ACCTSPAY, the precompiler names the COMMON blocks ACCTSC, ACCTSD, and ACCTSI.

The precompiler, however, can give COMMON blocks defined in different output files the same name, as the following schematic shows:

```
ACCTSPAY.PFO ===> ACCTSC, ACCTSD, ACCTSI in ACCTSPAY.FOR
ACCTSREC.PFO ===> ACCTSC, ACCTSD, ACCTSI in ACCTSREC.FOR
```

If you were to link ACCTSPAY and ACCTSREC into an executable program, the linker would see three, not six, COMMON blocks.

To solve the problem, you can rename the input files, or you can override the default COMMON block names by specifying COMMON_NAME inline or on the command line as follows:

```
COMMON_NAME=<block_name>
```

where *block_name* is a legal COMMON block name. For example, if you specify COMMON_NAME=PAY, the precompiler names its COMMON blocks PAYC and PAYI. At most, the first five characters in *block_name* are used.

For example, if you specify COMMON_NAME=PAY, the precompiler names its COMMON blocks PAYC and PAYI. At most, the first 5 characters in *block_name* are used.

If you specify COMMON_NAME inline, its EXEC ORACLE OPTION statement must precede the FORTRAN PROGRAM, SUBROUTINE, or FUNCTION statement.

You might want to override the default COMMON block names if they conflict with your user-defined COMMON block names. However, the preferred practice is to rename the user-defined COMMON blocks.

COMMON_NAME is not needed if you specify MULTISUBPROG.

COMMON_PARSER

Purpose

Specifies that the SQL99 syntax for SELECT, INSERT, DELETE, UPDATE and body of the cursor in a DECLARE CURSOR statement will be supported.

Syntax

COMMON_PARSER={YES | NO}

Default

NO

Usage Notes

Can be entered in the command line.

COMP_CHARSET

Purpose

Indicates to the Pro*C/C++ Precompiler whether multibyte character sets are (or are not) supported by the compiler to be used. It is intended for use by developers

working in a multibyte client-side environment (for example, when NLS_LANG is set to a multibyte character set).

Syntax

COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}

Default

MULTI_BYTE

Usage Notes

Can be entered only on the command line.

With COMP_CHARSET=MULTI_BYTE (default), Pro*C/C++ generates C code that is to be compiled by a compiler that supports multibyte character sets.

With COMP_CHARSET=SINGLE_BYTE, Pro*C/C++ generates C code for single-byte compilers that addresses a complication that *may* arise from the ASCII equivalent of a backslash (\) character in the second byte of a double-byte character in a multibyte string. In this case, the backslash (\) character is "escaped" with another backslash character preceding it.

> **Note:** The need for this feature is common when developing in a Shift-JIS environment with older C compilers.

This option has no effect when NLS_LANG is set to a single-byte character set.

COMP_CHARSET

Purpose

Indicates to the Pro*C/C++ Precompiler whether multibyte character sets are (or are not) supported by the compiler to be used. It is intended for use by developers working in a multibyte client-side environment (for example, when NLS_LANG is set to a multibyte character set).

Syntax

COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}

Default

MULTI_BYTE

Usage Notes

Can be entered only on the command line.

With COMP_CHARSET=MULTI_BYTE (default), Pro*C/C++ generates C code that is to be compiled by a compiler that supports multibyte character sets.

With COMP_CHARSET=SINGLE_BYTE, Pro*C/C++ generates C code for single-byte compilers that addresses a complication that *may* arise from the ASCII equivalent of a backslash (\) character in the second byte of a double-byte character in a multibyte string. In this case, the backslash (\) character is "escaped" with another backslash character preceding it.

Note: The need for this feature is common when developing in a Shift-JIS environment with older C compilers.

This option has no effect when NLS_LANG is set to a single-byte character set.

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.

The Oracle Precompilers can use a configuration file containing preset command-line options. By default, a text file called the *system configuration file* is used. However, you can specify any of several alternative files, called *user configuration files*.

You cannot nest configuration files. Therefore, you cannot specify the option `CONFIG` in a configuration file.

CPOOL

Purpose
Based on this option, the precompiler generates the appropriate code that directs SQLLIB to enable or disable the connection pool feature.

Syntax
CPOOL = {YES|NO}

Default
NO

Usage Notes
If this option is set to NO, other connection pooling options will be ignored by the precompiler.

CPP_SUFFIX

Purpose
The CPP_SUFFIX option provides the ability to specify the filename extension that the precompiler appends to the C++ output file generated when the CODE=CPP option is specified.

Syntax

CPP_SUFFIX=*filename_extension*

Default

System-specific.

Usage Notes

Most C compilers expect a default extension of ".c" for their input files. Different C++ compilers, however, can expect different filename extensions. The CPP_SUFFIX option provides the ability to specify the filename extension that the precompiler generates. The value of this option is a string, without the quotes or the period. For example, CPP_SUFFIX=cc, or CPP_SUFFIX=C.

CTIMEOUT

Purpose

Physical connections that are idle for more than the specified time (in seconds) are terminated to maintain an optimum number of open physical connections

Syntax

CTIMEOUT = Range is 1 to 65535.

Default

0 which means not set.

Usage Notes

Physical connections will not be closed until the connection pool is terminated.Creating a new physical connection will cost a round trip to the server.

DB2_ARRAY

Purpose

Based on this option, the precompiler activates the additional array insert/select syntax.

Syntax

DB2_ARRAY={YES |NO}

Default

NO

Usage Notes

If this option is set to NO, the Oracle precompiler syntax is supported, otherwise the DB2 insert/select array syntax is supported.

DBMS

Purpose

Specifies whether Oracle follows the semantic and syntactic rules of Oracle9*i*, Oracle8*i*, Oracle8, Oracle7, or the native version of Oracle (that is, the version to which your application is connected).

Syntax

```
DBMS={NATIVE|V7|V8}
```

Default

```
NATIVE
```

Usage Notes

Cannot be entered inline.

Using the DBMS option, you can 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.

When DBMS=V8, or DBMS=V7, Oracle follows the respective rules for Oracle9*i* (which remain the same as for Oracle7, Oracle8, and Oracle8*i*).

Table 6–4 shows how the compatible `DBMS` and `MODE` settings interact. All other combinations are incompatible or not recommended.

Table 6–4 Compatible DBMS and MODE Settings

Situation	DBMS=V7/V8 MODE=ANSI	DBMS=V7/V8 MODE=ORACLE
"no data found" warning code	+100	+1403
fetch nulls without using indicator variables	error -1405	error -1405
fetch truncated values without using indicator variables	no error but SQLWARN(2) is set	no error but SQLWARN(2) is set
cursors closed by COMMIT or ROLLBACK	all explicit	CURRENT OF only
open an already OPENed cursor	error -2117	no error
close an already CLOSEd cursor	error -2114	no error
SQL group function ignores nulls	no warning	no warning
when SQL group function in multirow query is called	FETCH time	FETCH time
declare SQLCA structure	optional	required
declare SQLCODE or SQLSTATE status variable	required	optional but Oracle ignores
default external datatype of character host variables external datatype used for CHAR in TYPE and VAR statements	CHARF	VARCHAR2
default external datatype of string literals in SQL statements	CHARF	CHARF
default internal datatype of CHAR variables in SQL statements	CHAR	CHAR

Table 6–4 Compatible DBMS and MODE Settings

Situation	DBMS=V7/V8 MODE=ANSI	DBMS=V7/V8 MODE=ORACLE
default external datatype of CHAR variables in PL/SQL blocks	CHARF	CHARF
default external datatype of value function USER returns	CHARF	CHARF
external datatype code DESCRIBE returns (dynamic SQL Method 4)	96	96
integrity constraints	enabled	enabled
PCTINCREASE for rollback segments	not allowed	not allowed
MAXEXTENTS storage parameters	not allowed	not allowed

DEF_SQLCODE

Purpose

Controls whether the Pro*C/C++ precompiler generates **#define**'s for SQLCODE.

Syntax

DEF_SQLCODE={NO | YES}

Default

NO

Usage Notes

Can be used only on the command line or in a configuration file.

When DEF_SQLCODE=YES, the precompiler defines SQLCODE in the generated source code as follows:

```
#define SQLCODE sqlca.sqlcode
```

You can then use SQLCODE to check the results of executable SQL statement. The DEF_SQLCODE option is supplied for compliance with standards that require the use of SQLCODE.

In addition, you must also include the SQLCA using one of the following entries in your source code:

```
#include <sqlca.h>
```

or

```
EXEC SQL INCLUDE SQLCA;
```

If the SQLCA is not included, using this option causes a precompile time error.

DEFINE

Purpose

Specifies a user-defined symbol that is used to include or exclude portions of source code during a conditional precompilation.

Syntax

`DEFINE=`*`symbol`*

Default

`None`

Usage Notes

If you enter `DEFINE` inline, the EXEC ORACLE statement takes the following form:

```
EXEC ORACLE DEFINE <symbol>;
```

DURATION

Purpose

Sets the pin duration used by subsequent EXEC SQL OBJECT CREATE and EXEC SQL OBJECT DEREF statements. Objects in the cache are implicitly unpinned at the end of the duration.

Syntax

DURATION={TRANSACTION | SESSION}

Default

TRANSACTION

Usage Notes

Can be entered inline by use of the EXEC ORACLE OPTION statement.

TRANSACTION means that objects are implicitly unpinned when the transaction completes.

SESSION means that objects are implicitly unpinned when the connection is terminated.

DYNAMIC

Purpose

This micro option specifies the descriptor behavior in dynamic SQL Method 4. The setting of MODE determines the setting of DYNAMIC.

Syntax

DYNAMIC={ORACLE | ANSI}

Default

ORACLE

Usage Notes

Cannot be entered inline by use of the EXEC ORACLE OPTION statement.

ERRORS

Purpose

Specifies whether precompiler 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.

ERRTYPE

Purpose

Specifies an output file in which errors generated in processing type files are written. If omitted, errors are output to the screen.

Syntax

ERRTYPE=*filename*

Default

None

Usage Notes

Only one error file will be produced. If multiple values are entered, the last one is used by the precompiler.

EVENTS

Purpose

Specifies that the application is interested in registering for and receiving notifications.

Syntax

EVENTS={YES | NO}

Default

NO

Usage Notes

Can only be entered in the command line.

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 (Pro*COBOL only), NUMBER, RAW, LONG RAW, VARRAW, ROWID, and VARCHAR datatypes
- ORACLE OPTION statement for specifying run-time 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 or FORTRAN input lines.

Syntax

```
FORMAT={ANSI|TERMINAL}
```

Default

```
ANSI
```

Usage Notes

Cannot be entered inline.

The format of input lines is system-dependent. Check your system-specific Oracle manuals.

When `FORMAT=ANSI`, the format of input lines conforms as much as possible to the current ANSI standard.

Globalization Support_LOCAL

Purpose

For Pro*COBOL only, the Globalization Support_LOCAL option determines whether Globalization Support character conversions are performed by the precompiler run-time library or by the Oracle Server.

Syntax

```
Globalization Support_LOCAL={YES|NO}
```

Default

```
NO
```

Usage Notes

Cannot be entered inline.

When `Globalization Support_LOCAL=YES`, the run-time library (SQLLIB) locally performs blank-padding and blank-stripping for host variables that have multibyte Globalization Support datatypes.

When Globalization `Support_LOCAL=NO`, blank-padding and blank-stripping operations are *not* performed locally for host variables that have multibyte Globalization Support datatypes.

Oracle does not perform any blank-padding or blank-stripping of Globalization Support variables. When Globalization Support_LOCAL=NO, the Oracle Server returns an error upon executing a SQL statement that uses multibyte Globalization Support data.

HEADER

Purpose

Permits precompiled header files. Specifies the file extension for precompiled header files.

Syntax

HEADER=*extension*

Default

NONE

Usage Notes

When precompiling a header file, this option is required and is used to specify the file extension for the output file that is created by precompiling that header file.

When precompiling an ordinary Pro*C/C++ program this option is optional. When given, it enables the use of the precompiled header mechanism during the precompilation of that Pro*C/C++ program.

In both cases, this option also specifies the file extension to use when processing a #include directive. If an #include file exists with the specified extension, Pro*C/C++ assumes the file is a precompiled header file previously generated by Pro*C/C++. Pro*C/C++ will then instantiate the data from that file rather than process the #include directive and precompile the included header file.

This option is only allowed on the command line or in a configuration file. It is not allowed inline. When using this option, specify the file extension only. Do not include any file separators. For example, do not include a period '.' in the extension.

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, refer to Appendix C

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. `HOLD_CURSOR` controls what happens to the link between the cursor and cursor cache.

When `HOLD_CURSOR=NO`, after Oracle executes the SQL statement and the cursor is closed, the precompiler marks the link as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks.

When `HOLD_CURSOR=YES` and `RELEASE_CURSOR=NO`, the link is maintained; the precompiler does not reuse it. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set `HOLD_CURSOR` before executing the SQL statement. For inline use with explicit cursors, set `HOLD_CURSOR` before opening the cursor.

Note that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For information showing how these two options interact, refer to Table C–1.

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 1985 version. Other values might be available on your platform.

IMPLICIT_SVPT

Purpose

Controls whether an implicit savepoint is taken before the start of a new batched insert.

Syntax

implicit_svpt={YES|NO}

Default

NO

Usage Notes

If implict_svpt=yes, a savepoint is taken before the start of a new batch of rows. If an error occurs on the insert, an implicit "rollback to savepoint" is executed. This option exists for DB/2 compatibility, the obvious downside being the extra round-trip.

If implict_svpt=no, there is no implicit savepoint taken. If an error occurs on the buffered insert, then it is reported back to the application, but no rollback is executed.

INAME

Purpose

Specifies the name of the input file.

Syntax

INAME=*filename*

Default

None

Usage Notes

Cannot be entered inline.

When specifying the name of your input file on the command line, the keyword **INAME** is optional. For example, in Pro*COBOL, you can specify *myprog.pco* instead of INAME=myprog.pco.

The precompiler assumes the standard input file extension (refer to Table 6–5). So, you need not use a file extension when specifying INAME unless the extension is nonstandard. For example, in Pro*FORTRAN, you can specify *myprog* instead of *myprog.pfo*.

Table 6–5 Input File Extensions

Host Language	Standard File Extension
COBOL	pco
FORTRAN	pfo

For Pro*COBOL only, if you use a nonstandard input file extension when specifying INAME, you must also specify HOST.

INCLUDE

Purpose

Specifies a directory path for EXEC SQL INCLUDE files. It only applies to operating systems that use directories.

Syntax

INCLUDE=path

Default

Current directory

Usage Notes

Typically, you use INCLUDE to specify a directory path for the SQLCA and ORACA files. The precompiler searches first in the current directory, then in the directory specified by INCLUDE, and finally in a directory for standard INCLUDE files. Hence, you need not specify a directory path for standard files such as the SQLCA and ORACA.

You must still use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. You can specify more than one path on the command line, as follows:

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

The precompiler searches first in the current directory, then in the directory named by *path1*, then in the directory named by *path2*, and finally in the directory for standard INCLUDE files.

Remember, the precompiler searches for a file in the current directory first--even if you specify a directory path. So, if the file you want to `INCLUDE` resides in another directory, make sure no file with the same name resides in the current directory.

The syntax for specifying a directory path is system-specific. Follow the conventions of your operating system.

IRECLEN

Purpose
Specifies the record length of the input file.

Syntax
`IRECLEN=`*`integer`*

Default
`80`

Usage Notes
Cannot be entered inline.

The value you specify for IRECLEN should not exceed the value of ORECLEN. The maximum value allowed is system-dependent.

INTYPE

Purpose
Specifies one or more OTT-generated type files (only needed if Object types are used in the application).

Syntax
INTYPE=(*file_1,file_2,...,file_n*)

Default
None

Usage Notes
There will be one type file for each Object type in the Pro*C/C++ code.

LINES

Purpose
Specifies whether the Pro*C/C++ precompiler adds **#line** preprocessor directives to its output file.

Syntax
LINES={YES | NO}

Default
NO

Usage Notes

Can be entered only on the command line.

The LINES option helps with debugging.

When LINES=YES, the Pro*C/C++ precompiler adds **#line** preprocessor directives to its output file.

Normally, your C compiler increments its line count after each input line is processed. The **#line** directives force the compiler to reset its input line counter so that lines of precompiler-generated code are not counted. Moreover, when the name of the input file changes, the next **#line** directive specifies the new filename.

The C compiler uses the line numbers and filenames to show the location of errors. Thus, error messages issued by the C compiler always refer to your original source files, not the modified (precompiled) source file. This also enables stepping through the original source code using most debuggers.

When LINES=NO (the default), the precompiler adds no **#line** directives to its output file.

> **Note:** The Pro*C/C++ precompiler does not support the **#line** directive. You cannot directly code **#line** directives in the precompiler source. But you can still use the LINES= option to have the precompiler insert **#line** directives for you.

LITDELIM

Purpose

For Pro*COBOL only, the LITDELIM option specifies the delimiter for string constants and literals.

Syntax

`LITDELIM={APOST|QUOTE}`

Default

`QUOTE`

Usage Notes

When `LITDELIM=APOST`, the precompiler 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 delimiter is used in the Pro*COBOL source file, the precompiler generates the delimiter 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 255. If you specify a value the range, 80 is used instead. If you specify a value earlier the range, 255 is used instead. LRECLEN should exceed IRECLEN by at least 8 to allow for the insertion of line numbers.

LTYPE

Purpose
Specifies the listing type.

Syntax
`LTYPE={LONG|SHORT|NONE}`

Default
`LONG`

Usage Notes
Cannot be entered inline.

When `LTYPE=LONG`, input lines appear in the listing file. When `LTYPE=SHORT`, input lines do *not* appear in the listing file. When `LTYPE=NONE`, no listing file is created.

MAXLITERAL

Purpose

Specifies the maximum length of string literals generated by the precompiler 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 precompiler-specific as shown here:

Precompiler	Default
Pro*COBOL	256
Pro*FORTRAN	1000

Usage Notes

The maximum value of MAXLITERAL is compiler-dependent. The default value is language-dependent, but you might have to specify a lower value. For example, some COBOL compilers cannot handle string literals longer than 132 characters, so you would specify MAXLITERAL=132.

Strings that exceed the length specified by MAXLITERAL are divided during precompilation, then recombined (concatenated) at run time.

You can enter MAXLITERAL inline but your program can set its value just once, and the EXEC ORACLE statement must precede the first EXEC SQL statement. Otherwise, the precompiler 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 the precompiler 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, refer to Appendix C

When precompiling separately, use MAXOPENCURSORS as described in "Separate Precompilations".

MAXOPENCURSORS specifies the *initial* size of the SQLLIB cursor cache. If a new cursor is needed, and there are no free cache entries, Oracle tries to reuse an entry. Its success depends on the values of HOLD_CURSOR and RELEASE_CURSOR, and, for explicit cursors, on the status of the cursor itself. Oracle allocates an additional cache entry if it cannot find one to reuse. If necessary, Oracle keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN_CURSORS. To avoid a "maximum open cursors exceeded" Oracle error, MAXOPENCURSORS must be lower than OPEN_CURSORS by at least 6.

As your program's need for concurrently open cursors grows, you might want to re-specify MAXOPENCURSORS to match the need. A value of 45 to 50 is not uncommon, but remember that each cursor requires another private SQL area in the user process memory space. The default value of 10 is adequate for most programs.

MAX_ROW_INSERT

Purpose

Controls the number of rows that need to be buffered before executing the INSERT statement.

Syntax

max_row_insert={number of rows to be buffered}

Default

0

Usage Notes

Any number greater than zero enables buffered insert feature and buffers that many rows before executing the INSERT statement.

MODE

Purpose

Specifies whether your program observes Oracle practices or complies with the current ANSI SQL standard.

Syntax

`MODE={ANSI|ISO|ANSI14|ISO14|ANSI13|ISO13|ORACLE}`

Default

`ORACLE`

Usage Notes

Cannot be entered inline.

The following pairs of `MODE` values are equivalent: ANSI and ISO, ANSI14 and ISO14, ANSI13 and ISO13.

When `MODE=ORACLE` (the default), your embedded SQL program observes Oracle practices.

When `MODE={ANSI14|ANSI13}`, your program complies closely with the current ANSI SQL standard.

When `MODE=ANSI`, your program complies *fully* with the ANSI standard and the following changes go into effect:

- CHAR column values, `USER` pseudocolumn values, character host values, and quoted literals are treated like ANSI fixed-length character strings. And, ANSI-compliant blank-padding semantics are used when you assign, compare, `INSERT`, `UPDATE`, `SELECT`, or `FETCH` such values.
- Issuing a `COMMIT` or `ROLLBACK` closes all explicit cursors. (When `MODE={ANSI13|ORACLE}`, a commit or rollback closes only cursors referenced in a `CURRENT OF` clause.)
- 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 reparsing.)
- The "no data found" Oracle warning code returned to SQLCODE becomes +100 instead of +1403. The error message text does not change.
- 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 (SQLCOD in FORTRAN) or a 5-byte character variable named SQLSTATE (SQLSTA in FORTRAN) must be declared. For more information, refer to "Error Handling Alternatives".

Table 6–4 shows how the MODE and DBMS settings interact. Other combinations are incompatible or are not recommended.

MULTISUBPROG

Purpose

For Pro*FORTRAN only, the MULTISUBPROG option specifies whether the Pro*FORTRAN precompiler generates `COMMON` statements and `BLOCK DATA` subprograms.

> **Note:** This option allows Pro*FORTRAN release 1.3 applications to migrate to later releases. You can ignore the MUTISUBPROG option if you are not migrating Pro*FORTRAN release 1.3 source code.

Syntax

`MULTISUBPROG={YES|NO}`

Default

`YES`

Usage Notes

Cannot be entered inline.

When `MULTISUBPROG=YES`, the precompiler generates `COMMON` statements and `BLOCK DATA` subprograms. Your host program does not access the `COMMON` blocks directly, but it allows two or more program units in the same precompilation unit to contain SQL statements.

However, the precompiler can give `COMMON` blocks defined in different output files the same name. If you link the files into an executable program, you get a link-time or

run-time error. To solve this problem, you can rename the input files or override the default COMMON block names by specifying the option `COMMON_NAME`. To avoid the problem, specify `MULTISUBPROG=NO`.

Specify `MULTISUBPROG=NO` if your Pro*FORTRAN source code has only a single subprogram in each source file (this was the restriction in release 1.3). When `MULTISUBPROG=NO`, the `COMMON_BLOCK` option is ignored and the precompiler generates no `COMMON` statements or `BLOCK DATA` subprograms. Every program unit that contains executable SQL statements *must* have a Declare Section. Otherwise, you get a precompilation error. For input files that contain more than one embedded SQL program unit, the precompiler generates the same declarations in each unit.

NATIVE_TYPES

Purpose
Support for native float/double.

Syntax
NATIVE_TYPES = {YES | NO}

Default
NO

Usage Notes
The native float and native double datatypes represent the single-precision and double-precision floating point values. They are represented natively, that is, in the host system's floating point format.

NLS_CHAR

Purpose
Specifies which C host character variables are treated by the precompiler as multibyte character variables.

Syntax
NLS_CHAR=*varname* or NLS_CHAR=(*var_1*,*var_2*,...,*var_n*)

Default
None.

Usage Notes
Can be entered only on the command line, or in a configuration file.

This option provides the ability to specify at precompile time a list of the names of one or more host variables that the precompiler must treat as multibyte character variables. You can specify only C *char* variables or Pro*C/C++ VARCHARs using this option.

If you specify in the option list a variable that is not declared in your program, then the precompiler generates no error.

NLS_LOCAL

Purpose
Determines whether multibyte character set conversions are performed by the precompiler run-time library, SQLLIB, or by the database server.

Syntax
NLS_LOCAL={NO | YES}

Default
NO

Usage Notes
When set to YES, local multibyte support is provided by Pro*C/C++ and the SQLLIB library. The option NLS_CHAR must be used to indicate which C host variables are multibyte.

When set to NO, Pro*C/C++ will use the database server support for multibyte objects. Set NLS_LOCAL to NO for all new applications.

Environment variable NLS_NCHAR must be set to a valid fixed-width National Character Set. Variable-width National Character Sets are not supported.

Can be entered only on the command line, or in a configuration file.

OBJECTS

Purpose
Requests support for object types.

Syntax
OBJECTS={YES | NO}

Default
YES

Usage Notes
Can only be entered in the command line.

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.

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.

OUTLINE

Purpose
Indicates that the outline SQL file must be generated for the SQL statements.

Syntax
outline={yes | no | category_name}

Default
no

Usage Notes
The outline SQL file should be in the DEFAULT category if the value is `yes` and the generated outline format is

```
DEFAULT_<filename>_<filetype>_<sequence_no>
```

If the category name is mentioned, then the SQL file should be generated in the category mentioned. The generated outline format for this is

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

The outline SQL file is not generated if the value is `no`.

Semantic check should be full when this option is turned on, which means option sqlcheck=full/semantics. If sqlcheck=syntax/limited/none, then error will be generated.

OUTLNPREFIX

Purpose
Controls the generation of the outline names.

Syntax
outlnprefix={none | prefix_name}

Default
no

Usage Notes
If outlnprefix=prefix_name, then the outline format

```
<category_name>_<filename>_<filetype>
```

is replaced with `<prefix_name>` for the outline names.

If the length of the outline name exceeds 30 bytes, then this option is helpful for the user who can just specify the prefix name.

If outlnprefix=none, then the outline names are generated by the system. The generated format is

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

Semantic check should be full when this option is turned on, which means option sqlcheck=full/semantics. If sqlcheck=syntax/limited/none, or outline=false, or both, then an error will be generated.

PAGELEN

Purpose
Specifies the number of lines in 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.

PARSE

Purpose
Specifies the way that the Pro*C/C++ precompiler parses the source file.

Syntax
PARSE={FULL | PARTIAL | NONE}

Default
FULL

Usage Notes
To generate C++ compatible code, the PARSE option must be either NONE or PARTIAL.

If PARSE=NONE or PARSE=PARTIAL, all host variables must be declared inside a Declare Section.

The variable SQLCODE must also be declared inside a declare section, or it cannot be relied on to detect errors. Check the default value of PARSE for your platform.

If PARSE=FULL, the C parser is used, and it does not understand C++ constructs, such as classes, in your code.

With PARSE=FULL or PARSE=PARTIAL Pro*C/C++ fully supports C preprocessor directives, such as **#define**, **#ifdef**, and so on. However, with PARSE=NONE conditional preprocessing is supported by EXEC ORACLE statements.

> **Note:** Some platforms have the default value of PARSE as other than FULL. See your system-dependent documentation.

PREFETCH

Purpose

Use this option to speed up queries by pre-fetching several 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 inline it must placed before OPEN statements with explicit cursors. Then the number of rows pre-fetched when that OPEN is done is determined by the last inline PREFETCH option in effect.

The value range allowed is 0.. 65535.

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, refer to Appendix C.

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. RELEASE_CURSOR controls what happens to the link between the cursor cache and private SQL area.

When RELEASE_CURSOR=YES, after Oracle executes the SQL statement and the cursor is closed, the precompiler immediately removes the link. This frees memory allocated to the private SQL area and releases parse locks. To make sure that associated resources are freed when you CLOSE a cursor, you must specify RELEASE_CURSOR=YES.

When RELEASE_CURSOR=NO and HOLD_CURSOR=YES, the link is maintained. The precompiler does not reuse the link unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed

often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set RELEASE_CURSOR before executing the SQL statement. For inline use with explicit cursors, set RELEASE_CURSOR before opening the cursor.

Note that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For information showing how these two options interact, refer to Appendix C–1

RUNOUTLINE

Purpose

Provides the developer with the option of executing "create outline" statements either by using precompiler or by the developer manually at a later time.

Syntax

runoutline={yes | no}

Default

no

Usage Notes

If runoutline=yes, then the generated 'create outline' statements are executed by the precompiler/translator at the end of a successful precompilation.

The outline option should be set to true or category_name when runoutline is used. Semantic check should be full when this option is turned on, which means option sqlcheck=full/semantics. If sqlcheck=syntax/limited/none, then error will be generated.

SELECT_ERROR

Purpose

Specifies whether your program generates an error when a single-row SELECT statement returns more than one row or more rows than a host array can accommodate.

Syntax

SELECT_ERROR={YES|NO}

Default

YES

Usage Notes

When **SELECT_ERROR=YES**, an error is generated if a single-row select returns too many rows or an array select returns more rows than the host array can accommodate.

When **SELECT_ERROR=NO**, no error is generated when a single-row select returns too many rows or when an array select returns more rows than the host array can accommodate.

Whether you specify **YES** or **NO**, a random row is selected from the table. To ensure a specific ordering of rows, use the **ORDER BY** clause in your `SELECT` statement. When `SELECT_ERROR=NO` and you use `ORDER BY`, Oracle returns the first row, or the first *n* rows if you are selecting into an array. When `SELECT_ERROR=YES`, whether you use `ORDER BY`, an error is generated if too many rows are returned.

SQLCHECK

Purpose

Specifies the type and extent of syntactic and semantic checking.

Syntax

`SQLCHECK={SEMANTICS|FULL|SYNTAX|LIMITED|NONE}`

Default

`SYNTAX`

Usage Notes

The values `SEMANTICS` and `FULL` are equivalent, as are the values `SYNTAX` and `LIMITED`.

The Oracle Precompilers 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 on the command line, inline and on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line. For example, if you specify `SQLCHECK=NONE` on the command line, you cannot specify `SQLCHECK=SYNTAX` inline.

If `SQLCHECK=SYNTAX|SEMANTICS`, the precompiler 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`, the precompiler checks the syntax and semantics of

- Data manipulation statements such as `INSERT` and `UPDATE`
- PL/SQL blocks

However, the precompiler checks only the syntax of remote data manipulation statements (those using the AT *db_name* clause).

The precompiler gets the information for a semantic check from embedded `DECLARE TABLE` statements or, if you specify the option `USERID`, by connecting to 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, the precompiler checks the syntax of

- Data manipulation statements
- PL/SQL blocks

No semantic checking is done. DECLARE TABLE statements are ignored and PL/SQL blocks are not allowed. When checking data manipulation statements, the precompiler uses Oracle database version 7 syntax rules, which are downwardly compatible. Specify SQLCHECK=SYNTAX when migrating your precompiled programs.

When SQLCHECK=NONE, no syntactic or semantic checking is done. DECLARE TABLE statements are ignored and PL/SQL blocks are not allowed. Specify SQLCHECK=NONE if your program

- Contains non-Oracle SQL (for example, because it will connect to a non-Oracle server through Open Gateway)
- References tables not yet created and lacks DECLARE TABLE statements for them

Table 6–6 summarizes the checking done by SQLCHECK. For more information about syntactic and semantic checking, refer to Appendix D.

Table 6–6 SQLCHECK Checking

	SQLCHECK=SEMANTIC		SQLCHECK=SYNTAX		SQLCHECK=NONE	
	Syntax	Semantic	Syntax	Semantic	Syntax	Semantic
DML	Y	Y	Y			
Remote DML	Y		Y			
PL/SQL	Y	Y				

STMT_CACHE

Purpose

Denotes the Statement cache size for the dynamic SQL statements.

Syntax

STMT_CACHE = Range is 0 to 65535

Default

0

Usage Notes

The stmt_cache option can be set to hold the anticipated number of distinct dynamic SQL statements in the application.

THREADS

Purpose

When THREADS=YES, the precompiler searches for context declarations.

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.

TYPE_CODE

Purpose

This micro option specifies whether ANSI or Oracle datatype codes are used in 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.

UNSAFE_NULL

Purpose

Specifying `UNSAFE_NULL=YES` prevents generation of `ORA-01405` messages when fetching `NULLs` without using indicator variables.

Syntax

`UNSAFE_NULL={YES|NO}`

Default

`NO`

Usage Notes

Cannot be entered inline.

The `UNSAFE_NULL=YES` is allowed only when `MODE=ORACLE` and DBMS=V7.

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.

USERID

Purpose
Specifies an Oracle username and password.

Syntax
`USERID=`*`username/password`*

Default
`None`

Usage Notes
Cannot be entered inline.

Do not specify this option when using the automatic logon feature, which accepts your Oracle username prefixed with the value of the Oracle initialization parameter `OS_AUTHENT_PREFIX`.

When `SQLCHECK=SEMANTICS,` if you want the precompiler to get needed information by connecting to Oracle and accessing the data dictionary, you must also specify USERID.

UTF16_CHARSET

Purpose
Specify the character set form used by UNICODE(UTF16) variables.

Syntax
```
UTF16_CHARSET={NCHAR_CHARSET | DB_CHARSET}
```

Default
`NCHAR_CHARSET`

Usage Notes
Can be used only on the command line or in a configuration file, but not inline.

If UTF16_CHARSET=NCHAR_CHARSET (the default), the UNICODE(UTF16) bind / define buffer is converted according to the server side National Character Set. There may be a performance impact when the target column is CHAR.

If UTF16_CHAR=DB_CHARSET, the UNICODE(UTF16) bind / define buffer is converted according to the database character set.

Caution: There may be data loss when the target column is NCHAR.

VARCHAR

Purpose

For Pro*COBOL only, the `VARCHAR` option instructs the precompiler to treat the COBOL group item described in Chapter 1 of the *Pro*COBOL Programmer's Guide* 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 1 of the *Pro*COBOL Programmer's Guide* is accepted as an Oracle VARCHAR external datatype with a length field and a string field.

When `VARCHAR=NO`, the Pro*COBOL Precompiler does not accept the implicit group items as `VARCHAR` external datatypes.

VERSION

Purpose

Determines which version of the object will be returned by the EXEC SQL OBJECT DEREF statement.

Syntax

VERSION={RECENT | LATEST | ANY}

Default

RECENT

Usage Notes

Can be entered inline using the EXEC ORACLE OPTION statement.

RECENT means that if the object has been selected into the object cache in the current transaction, then that object is returned. For transactions running in serializable mode, this option has the same effect as LATEST without incurring as many network round trips. Most applications should use RECENT.

LATEST means that if the object does not reside in the object cache, it is retrieved from the database. If It does reside in the object cache, it is refreshed from the server. Use LATEST with caution because it incurs the greatest number of network round trips. Use LATEST only when it is imperative that the object cache is kept as coherent as possible with the server buffer cache

ANY means that if the object already resides in the object cache, return that object. If not, retrieve the object from the server. ANY incurs the fewest network round trips. Use in applications that access read-only objects or when a user will have exclusive access to the objects.

XREF

Purpose

Specifies whether a cross-reference section is included in the listing file.

Syntax

XREF={YES|NO}

Default

YES

Usage Notes

When XREF=YES, cross references are included for host variables, cursor names, and statement names. The cross references show where each object is defined and referenced in your program.

When XREF=NO, the cross-reference section is not included.

Conditional Precompilations

Conditional precompilation includes (or excludes) sections of code in your host 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 and EXEC SQL statements in these sections. The following statements let you exercise conditional control over precompilation:

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

All EXEC ORACLE statements must be terminated with the statement terminator for your host language. For example, in Pro*COBOL, a conditional statement must be terminated with "END-EXEC." and in Pro*FORTRAN it must be terminated by a return character.

An Example

In the following example, the SELECT statement is precompiled only when the symbol *site2* is defined:

```
EXEC ORACLE IFDEF site2;
 EXEC SQL SELECT DNAME
 INTO :dept_name
 FROM DEPT
 WHERE DEPTNO = :dept_number;
EXEC ORACLE ENDIF;
```

Blocks of conditions can be nested as shown in the following example:

```
EXEC ORACLE IFDEF outer;
```

```
 EXEC ORACLE IFDEF inner;
 ...
 EXEC ORACLE ENDIF;
EXEC ORACLE ENDIF;
```

You can "comment out" host-language or embedded SQL code by placing it between IFDEF and ENDIF and *not* defining the symbol.

Defining Symbols

You can define a symbol in two ways. Either include the statement

```
EXEC ORACLE DEFINE symbol;
```

in your host program or define the symbol on the command line using the syntax

```
... INAME=filename ... DEFINE=symbol
```

where *symbol* is not case-sensitive.

Some port-specific symbols are predefined for you when the Oracle Precompilers are installed on your system. For example, predefined operating system symbols include CMS, MVS, MS-DOS, UNIX, and VMS.

Separate Precompilations

With the Oracle Precompilers, you can precompile several host program modules separately, then link them into one executable program. This supports modular programming, which is required when the functional components of a program are written and debugged by different programmers. The individual program modules need not be written in the same language.

Guidelines

The following guidelines will help you avoid some common problems.

Referencing Cursors

Cursor names are SQL identifiers, whose scope is the precompilation unit. Hence, cursor operations cannot span precompilation units (files). That is, you cannot declare a cursor in one file and open or fetch from it in another file. So, when doing a separate precompilation, make sure all definitions and references to a given cursor are in one file.

Specifying MAXOPENCURSORS

When you precompile the program module that connects to Oracle, specify a value for MAXOPENCURSORS that is high enough for any of the program modules. If you use it for another program module, MAXOPENCURSORS is ignored. Only the value in effect for the connect is used at run time.

Using a Single SQLCA

If you want to use just one SQLCA, you must declare it globally in one of the program modules.

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. Refer to Using Embedded SQL 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 the precompiler, then link the resulting object module with any modules needed from SQLLIB and system-specific Oracle libraries. Also, if you are embedding OCI calls, make sure to link in the OCI run-time library (OCILIB).

The linker resolves symbolic references in the object modules. If these references conflict, the link fails. This can happen when you try to link third party software into a precompiled program. Not all third-party software is compatible with Oracle, so you might have problems. Check with Oracle Customer Services to see if the software is supported.

Compiling and linking are system-dependent. For instructions, see your system-specific Oracle manuals.

System-Dependent

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, refer to your system-specific Oracle documentation.

Multibyte Globalization Support Compatibility

When using multibyte Globalization Support features, you must link your object files to the current version of the SQLLIB run-time library. The multibyte Globalization Support features in this release are supported by the SQLLIB run-time library and *not* by the Oracle Server. The resulting application can then be executed with any release of the Oracle database.

7

Defining and Controlling Transactions

This chapter explains how to perform transaction processing. You learn the basic techniques that safeguard the consistency of your database, including how to control whether changes to Oracle data are made permanent or undone. The following topics are discussed:

- Some Terms You Should Know
- How Transactions Guard Your Database
- How to Begin and End Transactions
- Using the COMMIT Statement
- Using the ROLLBACK Statement
- Using the SAVEPOINT Statement
- Using the RELEASE Option
- Using the SET TRANSACTION Statement
- Overriding Default Locking
- Fetching Across Commits
- Handling Distributed Transactions
- Guidelines

Some Terms You Should Know

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

The jobs or tasks that Oracle 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 Oracle. Oracle allows user sessions to work "simultaneously" and share computer resources. To do this, Oracle must control *concurrency*, which means many user accessing the same data. Without adequate concurrency controls, there might be a loss of *data integrity*. That is, changes to data or structures might be made incorrectly.

Oracle 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 Oracle data and structures. However, you can request *data locks* on tables or rows when it is to your advantage to override default locking. You can choose from several *modes* of locking such as *row share* and *exclusive*.

A *deadlock* can occur when two or more users try to access the same database object. For example, two users updating the same table might wait if each tries to update a row currently locked by the other. Because each user is waiting for resources held by another user, neither can continue until Oracle breaks the deadlock. Oracle signals an error to the participating transaction that had completed the least amount of work, and the "deadlock detected while waiting for resource" Oracle error code is returned to SQLCODE in the SQLCA.

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

How Transactions Guard Your Database

Oracle is transaction oriented; that is, it uses transactions to ensure data integrity. A transaction is a series of one or more logically related SQL statements you define to accomplish some task. Oracle treats the series of SQL statements as a unit so that all the changes brought about by the statements are either *committed* (made permanent) or *rolled back* (undone) at the same time. If your application program fails in the middle of a transaction, the database is automatically restored to its former (pre-transaction) state.

The subsequent sections show you how to define and control transactions. Specifically, you learn how to

- Begin and end transactions
- Use the `COMMIT` statement to make transactions permanent
- Use the `SAVEPOINT` statement with the `ROLLBACK TO` statement to undo parts of transactions
- Use the `ROLLBACK` statement to undo whole transactions
- Specify the `RELEASE` option to free resources and log off the database
- Use the `SET TRANSACTION` statement to set read-only transactions
- Use the `FOR UPDATE` clause or `LOCK TABLE` statement to override default locking

For details about the SQL statements discussed in this chapter, see the *Oracle Database SQL Language Reference*.

How to Begin and End Transactions

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

You end a transaction in one of the following ways:

- Code a `COMMIT` or `ROLLBACK` statement, with or without the `RELEASE` option. This *explicitly* makes permanent or undoes changes to the database.

- Code a data definition statement (ALTER, CREATE, or GRANT, for example) 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 interruption. Oracle rolls back the transaction.

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

Using the COMMIT Statement

You use the COMMIT statement to make changes to the database permanent. Until changes are committed, other users cannot access the changed data; they see it as it was before your transaction began. The COMMIT statement has no effect on the values of host variables or on the flow of control in your program. Specifically, the COMMIT statement

- Makes permanent all changes made to the database during the current transaction
- Makes these changes visible to other users
- Erases all savepoints (refer to Using the ROLLBACK Statement)
- Releases all row and table locks, but not parse locks
- Closes cursors referenced in a CURRENT OF clause or, when MODE={ANSI13|ORACLE}, closes *all* explicit cursors
- Ends the transaction

When MODE={ANSI13|ORACLE}, explicit cursors not referenced in a CURRENT OF clause remain open across commits. This can boost performance. For an example, refer to "Fetching Across Commits".

Because they are part of normal processing, COMMIT statements should be placed inline, on the main path through your program. Before your program terminates, it must explicitly commit pending changes. Otherwise, Oracle rolls them back. In the following example, you commit your transaction and disconnect from Oracle:

```
EXEC SQL COMMIT WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all Oracle resources (locks and cursors) held by your program and logs off the database.

You need not follow a data definition statement with a COMMIT statement because data definition statements issue an automatic commit before *and* after executing. So, whether they succeed or fail, the prior transaction is committed.

Using the ROLLBACK Statement

You use the ROLLBACK statement to undo pending changes made to the database. For example, if you make a mistake such as deleting the wrong row from a table, you can use ROLLBACK to restore the original data. The ROLLBACK statement has no effect on the values of host variables or on the flow of control in your program. Specifically, the ROLLBACK statement

- Undoes all changes made to the database during the current transaction
- Erases all savepoints
- Ends The Transaction
- Releases All Row And Table Locks, But Not Parse Locks
- Closes cursors referenced in a CURRENT OF clause or, when `MODE={ANSI|ANSI14}`, closes *all* explicit cursors

When `MODE={ANSI13|ORACLE}`, explicit cursors not referenced in a `CURRENT OF` clause remain open across rollbacks.

Because they are part of exception processing, `ROLLBACK` statements should be placed in error handling routines, off the main path through your program. In the following example, you roll back your transaction and disconnect from Oracle:

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

The optional keyword `WORK` provides ANSI compatibility. The `RELEASE` option frees all resources held by your program and logs off the database.

If a `WHENEVER` SQLERROR `GOTO` statement branches to an error handling routine that includes a `ROLLBACK` statement, your program might enter an infinite loop if the rollback fails with an error. You can avoid this by coding `WHENEVER` SQLERROR `CONTINUE` before the `ROLLBACK` statement.

For example, consider the following:

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
FOR EACH new employee
 display 'Employee number? ';
 read emp_number;
 display 'Employee name? ';
 read emp_name;
 EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
 VALUES (:emp_number, :emp_name);
ENDFOR;
...
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
```

Oracle rolls back transactions if your program terminates abnormally.

Statement-Level Rollbacks

Before executing any SQL statement, Oracle marks an implicit savepoint (not available to you). Then, if the statement fails, Oracle 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 that before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. Errors detected

while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

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

Using the SAVEPOINT Statement

You use the `SAVEPOINT` statement to mark and name the current point in the processing of a transaction. Each marked point is called a *savepoint*. For example, the following statement marks a savepoint named *start_delete*:

```
EXEC SQL SAVEPOINT start_delete;
```

Savepoints let you divide long transactions, giving you more control over complex procedures. For example, if a transaction performs several functions, you can mark a savepoint before each function. Then, if a function fails, you can easily restore the Oracle data to its former state, recover, then reexecute the function.

To undo part of a transaction, you use savepoints with the `ROLLBACK` statement and its `TO SAVEPOINT` clause. The `TO SAVEPOINT` clause lets you roll back to an intermediate statement in the current transaction, so you do not have to undo all your changes. Specifically, the `ROLLBACK TO SAVEPOINT` statement

- Undoes changes made to the database since the specified savepoint was marked
- Erases all savepoints marked after the specified savepoint
- Releases all row and table locks acquired since the specified savepoint was marked

In the example, 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 'Customer number? ';
 read cust_number;
 display 'Customer name? ';
 read cust_name;
EXEC SQL INSERT INTO MAIL_LIST (CUSTNO, CNAME, STAT)
 VALUES (:cust_number, :cust_name, 'ACTIVE');
ENDFOR;
FOR EACH revised status
 display 'Customer number? ';
 read cust_number;
 display 'New status? ';
 read new_status;
 EXEC SQL UPDATE MAIL_LIST
 SET STAT = :new_status WHERE CUSTNO = :cust_number;
ENDFOR;
-- mark savepoint
EXEC SQL SAVEPOINT start_delete;
EXEC SQL DELETE FROM MAIL_LIST WHERE STAT = 'INACTIVE';
IF sqlca.sqlerrd(3) < 25 THEN -- check number of rows deleted
 display 'Number of rows deleted is ', sqlca.sqlerrd(3);
ELSE
 display 'Undoing deletion of ', sqlca.sqlerrd(3), ' rows';
 EXEC SQL WHENEVER SQLERROR GOTO sql_error;
 EXEC SQL ROLLBACK TO SAVEPOINT start_delete;
```

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

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.

By default, the number of active savepoints in each user session is limited to 5. An *active* savepoint is one that you marked since the last commit or rollback. Your Database Administrator (DBA) can raise the limit by increasing the value of the Oracle initialization parameter SAVEPOINTS. If you give two savepoints the same name, the earlier savepoint is erased.

Using the RELEASE Option

Oracle rolls back changes automatically if your program terminates abnormally. Abnormal termination occurs when your program does not explicitly commit or roll back work and disconnect from Oracle using the RELEASE option.

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

```
EXEC SQL COMMIT RELEASE;
```

or

```
EXEC SQL ROLLBACK RELEASE;
```

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

Using the SET TRANSACTION Statement

You use the SET TRANSACTION statement to begin a read-only 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;
```

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, as a store manager, you check sales activity for the day, the past week, and the past month by using a read-only transaction to generate a summary report. The report is unaffected by other users updating the database during the transaction.

```
EXEC SQL SET TRANSACTION READ ONLY;
 EXEC SQL SELECT SUM(SALEAMT) INTO :daily FROM SALES
 WHERE SALEDATE = SYSDATE;
 EXEC SQL SELECT SUM(SALEAMT) INTO :weekly FROM SALES
 WHERE SALEDATE > SYSDATE - 7;
 EXEC SQL SELECT SUM(SALEAMT) INTO :monthly FROM SALES
 WHERE SALEDATE > SYSDATE - 30;
 EXEC SQL COMMIT WORK;
-- simply ends the transaction since there are no changes
-- to make permanent
-- format and print report
```

Overriding Default Locking

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

With the SELECT FOR UPDATE OF statement, you can explicitly lock specific rows of a table to make sure they do not change before an update or delete is executed. However, Oracle automatically obtains row-level locks at update or delete time. So, use the FOR UPDATE OF clause only if you want to lock the rows *before* the update or delete.

You can explicitly lock entire tables using the LOCK TABLE statement.

Using the FOR UPDATE OF Clause

When you DECLARE a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you use the FOR UPDATE OF clause to acquire exclusive row locks. SELECT FOR UPDATE OF identifies the rows that will be updated or deleted, then locks each row in the active set. (All rows are locked at the open, not as they are fetched.) This is useful, for example, when you want to base an update on the existing values in a row. You must make sure the row is not changed by another user before your update.

The FOR UPDATE OF clause is optional. For instance, instead of

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20
 FOR UPDATE OF SAL;
```

you can drop the FOR UPDATE OF clause and simply code

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

The `CURRENT OF` clause signals the precompiler to add a FOR UPDATE clause if necessary. You use the `CURRENT OF` clause to refer to the latest row fetched from a cursor.

Restrictions

If you use the `FOR UPDATE OF` clause, you cannot reference multiple tables. Also, an explicit `FOR UPDATE OF` or an implicit `FOR UPDATE` acquires exclusive row locks. Row locks are released when you commit or rollback (except when you rollback to a savepoint). If you try to fetch from a `FOR UPDATE` cursor after a commit, Oracle generates the following error:

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

Using the LOCK TABLE Statement

You use the `LOCK TABLE` statement to lock one or more tables in a specified lock mode. For example, the statement locks the EMP table in *row share* mode. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use.

```
EXEC SQL LOCK TABLE EMP IN ROW SHARE MODE NOWAIT;
```

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an *exclusive* lock. While one user has an exclusive lock on a table, no other users can insert, update, or delete rows in that table. For more information about lock modes, see the *Oracle Database Advanced Application Developer's Guide*.

The optional keyword NOWAIT tells Oracle not to wait for a table if it has been locked by another user. Control is immediately returned to your program, so it can do other work before trying again to acquire the lock. (You can check SQLCODE in the SQLCA to see if the table lock failed.) If you omit `NOWAIT`, Oracle waits until the table is available; the wait has no set limit.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. So, a query never blocks another query or an update, and an update never blocks a query. Only if two different transactions try to update the same row will one transaction wait for the other to complete. Table locks are released when your transaction issues a commit or rollback.

Fetching Across Commits

If you want to intermix commits and fetches, do not use the `CURRENT OF` clause. Instead, select the rowid of each row, then use that value to identify the current row during the update or delete. Consider the following example:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL, ROWID FROM EMP WHERE JOB = 'CLERK';
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary, :row_id;
 ...
 EXEC SQL UPDATE EMP SET SAL = :new_salary
 WHERE ROWID = :row_id;
 EXEC SQL COMMIT;
```

```
ENDLOOP;
```

Note: The fetched rows are *not* locked. So, you might get inconsistent results if another user modifies a row after you read it but before you update or delete it.

Handling Distributed Transactions

A *distributed database* is a single logical database comprising multiple physical databases at different nodes. A *distributed statement* is any SQL statement that accesses a remote node using a database link. A *distributed transaction* includes at least one distributed statement that updates data at multiple nodes of a distributed database. If the update affects only one node, the transaction is non-distributed.

When you issue a commit, changes to each database affected by the distributed transaction are made permanent. If you issue a rollback instead, all the changes are undone. However, if a network or computer 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';
...
EXEC SQL ROLLBACK FORCE '25.33.86';
```

FORCE commits or rolls back only the specified transaction and does not affect your current transaction. Note that you cannot manually roll back in-doubt transactions to a savepoint.

The COMMENT clause in the COMMIT statement lets you specify a comment to be associated with a distributed transaction. If ever the transaction is in doubt, Oracle stores the text specified by COMMENT in the data dictionary view DBA_2PC_PENDING along with the transaction ID. The text must be a quoted literal <= 50 characters in length. An example follows:

```
EXEC SQL COMMIT COMMENT 'In-doubt trans; notify Order Entry';
```

For more information about distributed transactions, see *Oracle Database Concepts*.

Guidelines

The following guidelines will help you avoid some common problems.

Designing Applications

When designing your application, group logically related actions together in one transaction. A well-designed transaction includes all the steps necessary to accomplish a given task -- no more and no less.

Data in the tables you reference must be left in a consistent state. So, the SQL statements in a transaction should change the data in a consistent way. For example, a transfer of funds between two bank accounts should include a debit to one account and a credit to another. Both updates should either succeed or fail together. An

unrelated update, such as a new deposit to one account, should not be included in the transaction.

Obtaining Locks

If your application programs include SQL locking statements, make sure the Oracle users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other users can lock tables they own or tables for which they have a privilege, such as `ALTER`, `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

Using PL/SQL

If a PL/SQL block is part of a transaction, commits and rollbacks inside the block affect the whole transaction. In the following example, the rollback undoes changes made by the update *and* the insert:

```
EXEC SQL INSERT INTO EMP ...
EXEC SQL EXECUTE
 BEGIN UPDATE emp
...
 ...
 EXCEPTION
 WHEN DUP_VAL_ON_INDEX THEN
 ROLLBACK;
 END;
END-EXEC;
...
```

8

Error Handling and Diagnostics

- The Need for Error Handling
- Error Handling Alternatives
- Using Status Variables when MODE={ANSI | ANSI14}
- Using the SQL Communications Area
- Using the Oracle Communications Area

An application program must anticipate run-time errors and attempt to recover from them. This chapter provides an in-depth discussion of error reporting and recovery. You learn how to handle warnings and errors using the status variables SQLCODE, SQLSTATE, SQLCA (SQL Communications Area), and the `WHENEVER` statement. You also learn how to diagnose problems by using the status variable ORACA (Oracle Communications Area). The following topics are discussed:

- The need for error handling
- Error handling alternatives
- Using status variables when MODE={ANSI | ANSI14}
- Using the SQL Communications Area
- Using the Oracle Communications Area

The Need for Error Handling

A significant part of every application program must be devoted to error handling. The main benefit of error handling is that it allows your program to continue operating in the presence of errors. Errors arise from design faults, coding mistakes, hardware failures, invalid user input, and many other sources

You cannot anticipate all possible errors, but you can plan to handle certain kinds of errors meaningful to your program. For the Oracle Precompilers, error handling means detecting and recovering from SQL statement execution errors.

You can also prepare to handle warnings such as "value truncated" and status changes such as "end of data." It is especially important to check for error and warning conditions after every data manipulation statement, because an `INSERT`, `UPDATE`, or `DELETE` statement might fail before processing all eligible rows in a table.

Error Handling Alternatives

The Oracle Precompilers provide four status variables that serve as error handling mechanisms:

- SQLCODE (SQLCOD in Pro*FORTRAN)
- SQLSTATE (SQLSTA in Pro*FORTRAN)
- SQLCA (using the WHENEVER statement)
- ORACA

The MODE option (described) governs ANSI/ISO compliance. The availability of the SQLCODE, SQLSTATE, and SQLCA variables depends on the MODE setting. You can declare and use the ORACA variable regardless of the MODE setting. For more information, refer to Using the Oracle Communications Area .

When `MODE={ORACLE|ANSI13}`, you must declare the SQLCA status variable. SQLCODE and SQLSTATE declarations are accepted (not recommended) but are not recognized as status variables. For more information, refer to Using the SQL Communications Area.

When `MODE={ANSI|ANSI14}`, you can use any one, two, or all three of the SQLCODE, SQLSTATE, and SQLCA variables. To determine which variable (or variable combination) is best for your application, refer to Using Status Variables when MODE={ANSI | ANSI14}" .

SQLCODE and SQLSTATE

With Release 1.5 of the Oracle Precompilers, the SQLCODE status variable was introduced as the SQL89 standard ANSI/ISO error reporting mechanism. The SQL92 standard listed SQLCODE as a deprecated feature and defined a new status variable, SQLSTATE (introduced with Release 1.6 of the Oracle Precompilers), as the preferred ANSI/ISO error reporting mechanism.

SQLCODE stores error codes and the "not found" condition. It is retained only for compatibility with SQL89 and is likely to be removed from future versions of the standard.

Unlike SQLCODE, SQLSTATE stores error and warning codes and uses a standardized coding scheme. After executing a SQL statement, the Oracle server returns a status code to the SQLSTATE variable currently in scope. The status code indicates whether a SQL statement executed successfully or raised an exception (error or warning condition). To promote *interoperability* (the ability of systems to exchange information easily), SQL92 predefines all the common SQL exceptions.

SQLCA

The SQLCA is a record-like, host-language data structure. Oracle updates the SQLCA after every *executable* SQL statement. (SQLCA values are undefined after a declarative statement.) By checking Oracle return codes stored in the SQLCA, your program can determine the outcome of a SQL statement. This can be done in two ways:

- Implicit checking with the WHENEVER statement
- Explicit checking of SQLCA variables

You can use `WHENEVER` statements, code explicit checks on SQLCA variables, or do both. Generally, using `WHENEVER` statements is preferable because it is easier, more portable, and ANSI-compliant.

ORACA

When more information is needed about run-time errors than the SQLCA provides, you can use the ORACA, which contains cursor statistics, SQL statement data, option settings, and system statistics.

The ORACA is optional and can be declared regardless of the MODE setting. For more information about the ORACA status variable, refer to "Using the Oracle Communications Area.

Using Status Variables when MODE={ANSI|ANSI14}

When `MODE={ANSI|ANSI14}`, you must declare at least one -- you may declare two or all three -- of the following status variables:

- SQLCODE
- SQLSTATE
- SQLCA

In Pro*COBOL, you cannot declare SQLCODE if SQLCA is declared. Likewise, you cannot declare SQLCA if SQLCODE is declared. The field in the SQLCA data structure that stores the error code for Pro*COBOL is also called SQLCODE, so errors will occur if both status variables are declared.

Your program can get the outcome of the most recent executable SQL statement by checking SQLCODE and SQLSTATE, SQLCODE or SQLSTATE explicitly with your own code after executable SQL and PL/SQL statements. Your program can also check SQLCA implicitly (with the `WHENEVER` SQLERROR and `WHENEVER` SQLWARNING statements) or it can check the SQLCA variables explicitly.

> **Note:** When `MODE={ORACLE|ANSI13}`, you must declare the SQLCA status variable. For more information, refer to Using the SQL Communications Area".

Some Historical Information

The treatment of status variables and variable combinations by the Oracle Precompilers has evolved beginning with Release 1.5.

Release 1.5

The Oracle Precompiler, Release 1.5, presumed there was a status variable SQLCODE whether it was declared in a Declare Section; in fact, the precompiler never bothered to note whether there was a declaration for SQLCODE or not -- it just presumed that the declaration exists. SQLCA would be used as a status variable only if there was an `INCLUDE` of the SQLCA.

Release 1.6

Beginning with Oracle Precompilers, Release 1.6, the precompilers no longer presume that there is a SQLCODE status variable and it is not required. The precompiler requires that *at least* one of SQLCA, SQLCODE, or SQLSTATE be declared.

SQLCODE is recognized as a status variable if and only if at least one of the following criteria is satisfied:

- It is declared in a Declare Section with *exactly* the correct datatype.

- The precompiler finds no other status variable.

If the precompiler finds a SQLSTATE declaration (of *exactly* the correct type of course) in a Declare Section or finds an `INCLUDE` of the SQLCA, it will *not* presume SQLCODE is declared.

Release 1.7

Because Release 1.5 of the Oracle Precompilers allowed the SQLCODE variable to be declared outside a Declare Section while declaring SQLCA at the same time, precompilers Release 1.6 and greater are presented with a compatibility problem. A new option, `ASSUME_SQLCODE={YES|NO}` (default NO), was added to fix this in Release 1.6.7 and is documented as a new feature in Release 1.7.

When `ASSUME_SQLCODE=YES`, and when SQLSTATE or SQLCA (Pro*FORTRAN only) are declared as status variables, the precompiler presumes SQLCODE is declared irrepsective of whether it is declared in a Declare Section or of the proper type. This causes Releases 1.6.7 and later to act like Release 1.5 in this regard. For information about the precompiler option ASSUME_SQLCODE, refer to "ASSUME_SQLCODE" .

Declaring Status Variables

This section describes how to declare SQLCODE and SQLSTATE. For information about declaring the SQLCA status variable, refer to "Declaring the SQLCA".

Declaring SQLCODE

SQLCODE (SQLCOD in Pro*FORTRAN) must be declared as a 4-byte integer variable either *inside* or *outside* the Declare Section, as shown in Table 8–1.

Table 8–1 SQLCODE Declarations

Language	SQLCODE Declaration
COBOL	SQLCODE PIC S9(9) COMP.
FORTRAN	INTEGER*4 SQLCOD

If declared outside the Declare Section, SQLCODE is recognized as a status variable only if ASSUME_SQLCODE=YES. SQLCODE declarations are ignored when `MODE={ORACLE|ANSI13}`.

> **Warning:** In Pro*COBOL source files, *donot* declare SQLCODE if SQLCA is declared. Likewise, *donot* 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.

By using host languages that allow both local and global declarations, you can declare more than one SQLCODE variable. Access to a local SQLCODE is limited by its scope within your program. After every SQL operation, Oracle returns a status code to the SQLCODE currently in scope. So, your program can learn the outcome of the most recent SQL operation by checking SQLCODE explicitly, or implicitly with the `WHENEVER` statement.

When you declare SQLCODE instead of the SQLCA in a particular compilation unit, the precompiler allocates an internal SQLCA for that unit. Your host program cannot

access the internal SQLCA. If you declare the SQLCA *and* SQLCODE (not supported in Pro*COBOL), Oracle returns the same status code to both after every SQL operation.

Declaring SQLSTATE

SQLSTATE (SQLSTA in Pro*FORTRAN) must be declared as a five-character alphanumeric string inside the Declare Section, as shown inTable 8–2. Declaring the SQLCA is optional.

Table 8–2 SQLSTATE Declarations

Language	SQLSTATE Declaration
COBOL	SQLSTATE PIC X(5).
FORTRAN	CHARACTER*5 SQLSTA

When MODE={ORACLE | ANSI13}, declarations of the SQLSTATE variable are ignored.

Status Variable Combinations

When `MODE={ANSI|ANSI14}`, the behavior of the status variables depends on the following:

- Which variables are declared
- Declaration placement (*inside* or *outside* the Declare Section)
- ASSUME_SQLCODE setting

Table 8–3 and Table 8–4 describe the resulting behavior of each status variable combination when ASSUME_SQLCODE=NO and when ASSUME_SQLCODE=YES, respectively.

Table 8–3 Status Variable Combinations - SQLCODE = NO

Declare Section (IN/OUT/ --)			Behavior
SQLCODE	**SQLSTATE**	**SQLCA**	
OUT	--	--	SQLCODE is declared and is presumed to be a status variable.
OUT	--	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCA is declared as a status variable, and SQLCODE is declared but is not recognized as a status variable.
OUT	--	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, 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	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCA is declared as a status variable, and SQLCODE and SQLSTATE are declared but are not recognized as status variables.
OUT	OUT	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, this status variable configuration is not supported.

Table 8–3 Status Variable Combinations - SQLCODE = NO

Declare Section (IN/OUT/ --)			Behavior
OUT	IN	--	SQLSTATE is declared as a status variable, and SQLCODE is declared but is not recognized as a status variable.
OUT	IN	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLSTATE and SQLCA are declared as status variables, and SQLCODE is declared but is not recognized as a status variable.
OUT	IN	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, this status variable configuration is not supported.
IN	--	--	SQLCODE is declared as a status variable.
IN	--	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCODE and SQLCA are declared as a status variables.
IN	--	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, 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	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCODE and SQLCA are declared as a status variables, and SQLSTATE is declared but is not recognized as a status variable.
IN	OUT	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, this status variable configuration is not supported.
IN	IN	--	SQLCODE and SQLSTATE are declared as a status variables.
IN	IN	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCODE, SQLSTATE, and SQLCA are declared as a status variables.
IN	IN	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, this status variable configuration is not supported.
--	--	--	This status variable configuration is not supported.
--	--	OUT	SQLCA is declared as a status variable.
--	--	IN	In Pro*COBOL, SQLCA is declared as a status host variable. In Pro*FORTRAN, this status variable configuration is not supported.
--	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	In Pro*COBOL, SQLCA is declared as a status host variable, and SQLSTATE is declared but is not recognized as a status variable. In Pro*FORTRAN, this status variable configuration is not supported.
--	IN	--	SQLSTATE is declared as a status variable.
--	IN	OUT	SQLSTATE and SQLCA are declared as status variables.

Table 8–3 Status Variable Combinations - SQLCODE = NO

Declare Section (IN/OUT/ --)			Behavior
--	IN	IN	In Pro*COBOL, SQLSTATE and SQLCA are declared as status host variables. In Pro*FORTRAN, this status variable configuration is not supported.

Table 8–4 Status Variable Combinations - SQLCODE = YES

Declare Section (IN/OUT/ --)			Behavior
SQLCODE	**SQLSTATE**	**SQLCA**	
OUT	--	--	SQLCODE is declared and is presumed to be a status variable.
OUT	--	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCA is declared as a status variable, and SQLCODE is declared and is presumed to be a status variable.
OUT	--	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, 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	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCA is declared as a status variable, SQLCODE is declared and is presumed to be a status variable, and SQLSTATE is declared but is not recognized as status variable.
OUT	OUT	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, 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	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLSTATE and SQLCA are declared as status variables, and SQLCODE is declared and is presumed to be a status variable.
OUT	IN	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, this status variable configuration is not supported.
IN	--	--	SQLCODE is declared as a status variable.
IN	--	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCODE and SQLCA are declared as a status variables.
IN	--	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, 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.

Table 8–4 Status Variable Combinations - SQLCODE = YES

Declare Section (IN/OUT/ --)			Behavior
IN	OUT	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCODE and SQLCA are declared as a status variables, and SQLSTATE is declared but is not recognized as a status variable.
IN	OUT	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, this status variable configuration is not supported.
IN	IN	--	SQLCODE and SQLSTATE are declared as a status variables.
IN	IN	OUT	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, SQLCODE, SQLSTATE, and SQLCA are declared as a status variables.
IN	IN	IN	In Pro*COBOL, this status variable configuration is not supported. In Pro*FORTRAN, this status variable configuration is not supported.
-- -- -- -- -- -- -- -- --	-- -- -- OUT OUT OUT IN IN IN	-- OUT IN -- OUT IN -- OUT IN	These status variable configurations are not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.

Status Variable Values

This section describes the values for the SQLCODE and SQLSTATE status variables. For information about the SQLCA status variable, refer to "Key Components of Error Reporting".

SQLCODE Values

After every SQL operation, Oracle returns a status code to the SQLCODE variable currently in scope. The status code, which indicates the outcome of the SQL operation, can be any of the following numbers:

0

Oracle executed the SQL statement without detecting an error or exception.

> 0

Oracle executed the statement but detected an exception. This occurs when Oracle cannot find a row that meets the condition in your `WHERE` clause or when a `SELECT INTO` or `FETCH` returns no rows.

When `MODE={ANSI|ANSI14|ANSI13}, +100` is returned to SQLCODE after an `INSERT` of no rows. This can happen when a subquery returns no rows to process.

< 0

Oracle did not execute the statement because of a database, system, network, or application error. Such errors are irrecoverable. When they occur, the current transaction should, in most cases, be rolled back. Negative return codes correspond to error codes listed in *Oracle Database Error Messages*.

You can learn the outcome of the most recent SQL operation by checking SQLCODE explicitly with your own code or implicitly with the `WHENEVER` statement.

When you declare SQLCODE instead of the SQLCA in a particular precompilation unit, the precompiler allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA. If you declare the SQLCA *and* SQLCODE (Pro*FORTRAN only), Oracle returns the same status code to both after every SQL operation.

> **Note:** When `MODE={ORACLE|ANSI13}`, declarations of SQLCODE are ignored.

SQLSTATE Values

SQLSTATE status codes consist of a two-character *class code* followed by a three-character *subclass code*. Aside from class code 00 (successful completion), the class code denotes a category of exceptions. Aside from subclass code 000 (not applicable), the subclass code denotes a specific exception within that category. For example, the SQLSTATE value `22012' consists of class code 22 (data exception) and subclass code 012 (division by zero).

Each of the five characters in a SQLSTATE value is a digit (0..9) or an uppercase Latin letter (A..Z). Class codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined conditions (those defined in SQL92). All other class codes are reserved for implementation-defined conditions. Within predefined classes, subclass codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined subconditions. All other subclass codes are reserved for implementation-defined subconditions. Figure 8–1 shows the coding scheme.

Figure 8–1 SQLSTATE Coding Scheme

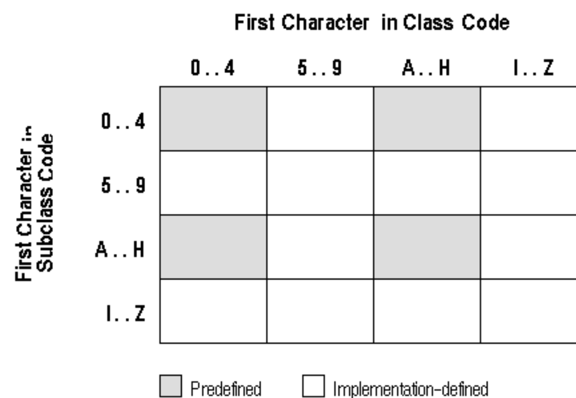


Table 8–5 shows the classes predefined by SQL92.

Table 8–5 Predefined SQL92 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

Table 8–5 Predefined SQL92 Classes

Class	Condition
22	data exception
23	integrity constraint violation
24	invalid cursor state
25	invalid transaction state
26	invalid SQL statement name
27	triggered data change violation
28	invalid authorization specification
2A	direct SQL syntax error or access rule violation
2B	dependent privilege descriptors still exist
2C	invalid character set name
2D	invalid transaction termination
2E	invalid connection name
33	invalid SQL descriptor name
34	invalid cursor name
35	invalid condition number
37	dynamic SQL syntax error or access rule violation
3C	ambiguous cursor name
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 8–6 shows how Oracle errors map to SQLSTATE status codes. In some cases, several Oracle errors map to the status code. In other cases, no Oracle error maps to the status code (so the last column is empty). Status codes in the range 60000 .. 99999 are implementation-defined.

Table 8–6 Oracle Error Mapping to SQLSTATE Status

Code	Condition	Oracle Error
00000	successful completion	ORA-00000
01000	warning	
01001	cursor operation conflict	
01002	disconnect error	
01003	null value eliminated in set function	

Table 8–6 Oracle Error Mapping to SQLSTATE Status

Code	Condition	Oracle Error
01004	string data - right truncation	
01005	insufficient item descriptor areas	SQL-02142
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 ORA-0100
07000	dynamic SQL error	SQL-02137 SQL-02139
07001	using clause does not match parameter specs	
07002	using clause does not match target specs	
07003	cursor specification cannot be executed	
07004	using clause required for dynamic parameters	
07005	prepared statement not a cursor specification	
07006	restricted datatype attribute violation	
07007	using clause required for result fields	
07008	invalid descriptor count	SQL-02126 SQL-02141
07009	invalid descriptor index	SQL-02140
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 ORA-01422
22000	data exception	
22001	string data - right truncation	ORA-01401 ORA-01406 ORA-12899
22002	null value - no indicator parameter	ORA-01405 SQL-02124

Table 8–6 Oracle Error Mapping to SQLSTATE Status

Code	Condition	Oracle Error
22003	numeric value out of range	ORA-01426 ORA-01438 ORA-01455 ORA-01457
22005	error in assignment	
22007	invalid datetime format	
22008	datetime field overflow	ORA-01800 .. 01899
22009	invalid time zone displacement value	
22011	substring error	
22012	division by zero	ORA-01476
22015	interval field overflow	
22018	invalid character value for cast	
22019	invalid escape character	ORA-00911 ORA-01425
22021	character not in repertoire	
22022	indicator overflow	ORA-01411
22023	invalid parameter value	ORA-01025 ORA-01488 ORA-04000 .. 04019
22024	unterminated C string	ORA-01479 .. 01480
22025	invalid escape sequence	ORA-01424
22026	string data - length mismatch	
22027	trim error	
23000	integrity constraint violation	ORA-00001 ORA-01400 ORA-02290 .. 02299
24000	invalid cursor state	ORA-01001 .. 01003 ORA-01410 ORA-06511 ORA-08006 SQL-02114 SQL-02117 SQL-02118 SQL-02122
25000	invalid transaction state	
26000	invalid SQL statement name	
27000	triggered data change violation	
28000	invalid authorization specification	
2A000	direct SQL syntax error or access rule violation	
2B000	dependent privilege descriptors still exist	
2C000	invalid character set name	
2D000	invalid transaction termination	
2E000	invalid connection name	

Table 8–6 Oracle Error Mapping to SQLSTATE Status

Code	Condition	Oracle Error
33000	invalid SQL descriptor name	SQL-02138
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
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	shared server and detached process errors	ORA-00101 .. 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; archival and media recovery errors	ORA-00200 .. 00369 ORA-01100 .. 01250
65000	PL/SQL errors	ORA-06500 .. 06599

Table 8–6 Oracle Error Mapping to SQLSTATE Status

Code	Condition	Oracle Error
66000	SQL*Net driver errors	ORA-06000 .. 06149 ORA-06250 .. 06429 ORA-06600 .. 06999 ORA-12100 .. 12299 ORA-12500 .. 12599
67000	licensing errors	ORA-00430 .. 00439
69000	SQL*Connect errors	ORA-00570 .. 00599 ORA-07000 .. 07199
72000	SQL execute phase errors	ORA-01000 .. 01099 ORA-01400 .. 01489 ORA-01495 .. 01499 ORA-01500 .. 01699 ORA-02400 .. 02419 ORA-02425 .. 02449 ORA-04060 .. 04069 ORA-08000 .. 08190 ORA-12000 .. 12019 ORA-12300 .. 12499 ORA-12700 .. 21999
82100	out of memory (could not allocate)	SQL-02100
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 Oracle cursor number	SQL-02106
82107	program too old for run-time library	SQL-02107
82108	invalid descriptor passed to run-time library	SQL-02108
82109	inconsistent host cache: host reference is out of range	SQL-02109
82110	inconsistent host cache: invalid host cache entry type	SQL-02110
82111	heap consistency error	SQL-02111
82112	unable to open message file	SQL-02113
82113	code generation internal consistency failed	SQL-02115
82114	reentrant code generator gave invalid context	SQL-02116
82115	invalid hstdef argument	SQL-02119
82116	first and second arguments to sqlrcn both null	SQL-02120
82117	invalid OPEN or PREPARE for this connection	
82118	application context not found	SQL-02123

Table 8–6 Oracle Error Mapping to SQLSTATE Status

Code	Condition	Oracle Error
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	run-time context in use	SQL-02131
82124	unable to allocate run-time context	ORA-01422 SQL-02132
82125	unable to initialize process for use with threads	SQL-02133
82126	invalid run-time context	SQL-02134
90000	debug events	ORA-10000 .. 10999
99999	catch all	all others
HZ000	remote database access	

Using the SQL Communications Area

The SQL Communications area (SQLCA) is a record-like data structure. Its fields contain error, warning, and status information updated by Oracle whenever a SQL statement is executed. Thus, the SQLCA always reflects the outcome of the most recent SQL operation. To determine the outcome, you can check variables in the SQLCA.

In host languages that allow both local and global declarations, your program can have more than one SQLCA. For example, it might have one global SQLCA and several local ones. Access to a local SQLCA is limited by its scope within the program. Oracle returns information only to the "active" SQLCA.

Also note that, when your application uses SQL*Net to access a combination of local and remote databases concurrently, all the databases write to one SQLCA. There is *no* different SQLCA for each database. For more information, refer to "Concurrent Logons" .

When `MODE={ORACLE|ANSI13}`, the SQLCA is required; if the SQLCA is not declared, compile-time errors will occur. The SQLCA is optional when `MODE={ANSI|ANSI14}`, but you cannot use the `WHENEVER` SQLWARNING statement without declaring SQLCA. So, if you want to use the `WHENEVER` SQLWARNING statement, you must declare the SQLCA.

If you declare SQLCODE instead of the SQLCA in a particular compilation unit, the precompiler allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA. If you declare the SQLCA *and* SQLCODE (Pro*FORTRAN only), Oracle returns the same status code to both after every SQL operation.

When `MODE={ANSI|ANSI14}`, you must declare either SQLSTATE (refer to SQLCODE and SQLSTATE The SQLSTATE status variable supports the SQLSTATE status variable specified by the SQL92 standard. You can use the SQLSTATE status variable with or without SQLCODE. refer to Table 8–3 and Table 8–4for more information.

Declaring the SQLCA

To declare the SQLCA, simply include it (using an EXEC SQL INCLUDE statement) in your host-language source file as follows:

```
* Include the Oracle Communications Area (ORACA).
 EXEC SQL INCLUDE ORACA
EXEC SQL INCLUDE SQLCA;
```

The SQLCA is used if and only if there is an INCLUDE of the SQLCA.

When you precompile your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow Oracle to communicate with the program.

Declaring the SQLCA in Pro*COBOL

In Pro*COBOL, it makes no difference whether the INCLUDE is *inside* or *outside* of a Declare Section. For more information about declaring the SQLCA in Pro*COBOL, refer to *Pro*COBOL Programmer's Guide*.

Declaring the SQLCA in Pro*FORTRAN

In Pro*FORTRAN, the SQLCA must be declared *outside* the Declare Section, because it is a COMMON block. Furthermore, the SQLCA must come before the CONNECT statement and the first executable FORTRAN statement.

You must declare the SQLCA in each subroutine and function that contains SQL statements. Every time a SQL statement in one of the subroutines or functions is executed, Oracle updates the SQLCA held in the COMMON block.

Ordinarily, only the order and datatypes of variables in a COMMON-list matter, not their names. However, you cannot rename the SQLCA variables because the precompiler generates code that refers to them. Thus, all declarations of the SQLCA must be identical. For more information about declaring the SQLCA in Pro*FORTRAN, refer to *Pro*FORTRAN Supplement to the Oracle Precompilers Guide*.

What's in the SQLCA?

The SQLCA contains the following run-time information about the outcome of SQL statements:

- Oracle error codes
- Warning flags
- Event information
- Rows-processed count
- Diagnostics

Figure 8–2 shows all the variables in the SQLCA. To see the SQLCA structure and variable names in a particular host language, refer to your supplement to this Guide.

Figure 8–2 SQLCA Variables

Variable	Description
SQLCAID	Character string "SQLCA"
SQLCABC	Length of SQLCA data structure in bytes
SQLCODE	Oracle error message code
SQLERRM	Subrecord for storing error message
SQLERRML	Length of error message
SQLERRMC	Text of error message
SQLERRP	Reserved for future use
SQLERRD	Array of six integer status codes
SQLERRD(1)	Reserved for future use
SQLERRD(2)	Reserved for future use
SQLERRD(3)	Number of rows processed
SQLERRD(4)	Reserved for future use
SQLERRD(5)	Parse error offset
SQLERRD(6)	Reserved for future use
SQLWARN	Array of eight warning flags
SQLWARN(0)	Another warning flag set
SQLWARN(1)	Character string truncated
SQLWARN(2)	No longer in use
SQLWARN(3)	SELECT list not equal to INTO list
SQLWARN(4)	DELETE or UPDATE without WHERE clause
SQLWARN(5)	Reserved for future use
SQLWARN(6)	No longer in use
SQLWARN(7)	No longer in use
SQLEXT	Reserved for future use

Key Components of Error Reporting

Error reporting depends on variables in the SQLCA. This section highlights the key components of error reporting. The next section takes a close look at the SQLCA.

Status Codes

Every executable SQL statement returns a status code to the SQLCA variable SQLCODE, which you can check implicitly with the `WHENEVER` statement or explicitly with your own code.

Status codes can be zero, less than zero, or greater than zero. Refer to Declaring SQLCODE" for complete SQLCODE descriptions.

Warning Flags

Warning flags are returned in the SQLCA variables SQLWARN(0) through SQLWARN(7), which you can check implicitly or explicitly. These warning flags are useful for run-time conditions not considered errors by Oracle.

Rows-Processed Count

The number of rows processed by the most recently executed SQL statement is returned in the SQLCA variable SQLERRD(3), which you can check explicitly.

Speaking strictly, this variable is not for error reporting, but it can help you avoid mistakes. For example, suppose you expect to delete about ten rows from a table. After the deletion, you check SQLERRD(3) and find that 75 rows were processed. To be safe, you might want to roll back the deletion and examine the search condition in your `WHERE` clause.

Parse Error Offset

Before executing a SQL statement, Oracle must *parse* it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. If Oracle finds an error, an offset is stored in the SQLCA variable SQLERRD(5), which you can check explicitly.

The offset specifies the character position in the SQL statement at which the parse error begins. The first character occupies position zero. For example, if the offset is 9, the parse error begins at the tenth character.

By default, static SQL statements are checked for syntactic errors at precompile time. So, SQLERRD(5) is most useful for debugging dynamic SQL statements, which your program accepts or builds at run time.

Parse errors arise from missing, misplaced, or misspelled keywords, invalid options, nonexistent tables, and the like. For example, the dynamic SQL statement

```
UPDATE EMP SET JIB = :job_title WHERE EMPNO = :emp_number
```

causes the parse error

```
ORA-00904: invalid column name
```

because the column name JOB is misspelled. The value of SQLERRD(5) is 15 because the erroneous column name JIB begins at the sixteenth character.

If your SQL statement does not cause a parse error, Oracle sets SQLERRD(5) to zero. Oracle also sets SQLERRD(5) to zero if a parse error begins at the first character (which occupies position zero). So, check SQLERRD(5) only if SQLCODE is negative, which means that an error has occurred.

Error Message Text

The error code and message for Oracle errors are available in the SQLCA variable SQLERRMC. At most, the first 70 characters of text are stored. To get the full text of messages longer than 70 characters, you use the SQLGLM function. Refer to "Getting the Full Text of Error Messages".

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

Oracle executed the statement without detecting an error or exception.

> 0

Oracle executed the statement but detected an exception. This occurs when Oracle cannot find a row that meets your `WHERE`-clause search condition or when a `SELECT INTO` or `FETCH` returns no rows.

< 0

When `MODE={ANSI|ANSI14|ANSI13}, +100` is returned to SQLCODE after an INSERT of no rows. This can happen when a subquery returns no rows to process.

Oracle did not execute the statement because of a database, system, network, or application error. Such errors are irrecoverable. When they occur, the current transaction should, in most cases, be rolled back.

Negative return codes correspond to error codes listed in *Oracle Database Error Messages*.

SQLERRM

This subrecord contains the following two fields:

SQLERRML

This integer field holds the length of the message text stored in SQLERRMC.

SQLERRMC

This string field holds the message text 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 array of binary integers has six elements. Descriptions of the fields in SQLERRD (called SQLERD in FORTRAN) follow:

SQLERRD(1)

This field is reserved for future use.

SQLERRD(2)

This field is reserved for future use.

SQLERRD(3)

This field holds the number of rows processed by the most recently executed SQL statement. However, if the SQL statement failed, the value of SQLERRD(3) is undefined, with one exception. If the error occurred during an array operation, processing stops at the row that caused the error, so SQLERRD(3) gives the number of rows processed successfully.

The rows-processed count is zeroed after an `OPEN` statement and incremented after a `FETCH` statement. For the `EXECUTE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT INTO` statements, the count reflects the number of rows processed successfully. The count does *not* include rows processed by an update or delete cascade. For example, if 20 rows are deleted because they meet WHERE-clause criteria, and 5 more rows are deleted because they now (after the primary delete) violate column constraints, the count is 20 not 25.

SQLERRD(4)

This field is reserved for future use.

SQLERRD(5)

This field holds an offset that specifies the character position at which a parse error begins in the most recently executed SQL statement. The first character occupies position zero.

SQLERRD(6)

This field is reserved for future use.

SQLWARN

This array of single characters has eight elements. They are used as warning flags. Oracle sets a flag by assigning it a "W" (for warning) character value. The flags warn of exceptional conditions.

For example, a warning flag is set when Oracle assigns a truncated column value to an output host variable.

Also note that, while Figure 8–2 illustrates SQLWARN as an array, it is implemented in Pro*COBOL as a group item with elementary PIC X items named SQLWARN0 through SQLWARN7. The Pro*FORTRAN implementation is composed of the LOGICAL variables, SQLWN0 through SQLWN7.

Descriptions of the fields in SQLWARN follow:

SQLWARN(0)

This flag is set if another warning flag is set.

SQLWARN(1)

This flag is set if a truncated column value was assigned to an output host variable. This applies only to character data. Oracle truncates certain numeric data without setting a warning or returning a negative SQLCODE value.

To find out if a column value was truncated and by how much, check the indicator variable associated with the output host variable. The (positive) integer returned by an indicator variable is the original length of the column value. You can increase the length of the host variable accordingly.

SQLWARN(2)

This flag is set if one or more nulls were ignored in the evaluation of a SQL group function such as AVG, COUNT, or MAX. This behavior is expected because, except for COUNT(*), all group functions ignore nulls. If necessary, you can use the SQL function NVL to temporarily assign values (zeros, for example) to the null column entries.

SQLWARN(3)

This flag is set if the number of columns in a query select list does not equal the number of host variables in the `INTO` clause of the `SELECT` or `FETCH` statement. The number of items returned is the lesser of the two.

SQLWARN(4)

This flag is set if every row in a table was processed by an `UPDATE` or `DELETE` statement without a WHERE clause. An update or deletion is called *unconditional* if no search condition restricts the number of rows processed. Such updates and deletions are unusual, so Oracle sets this warning flag. That way, you can roll back the transaction if necessary

SQLWARN(5)

This flag is set when an EXEC SQL CREATE {PROCEDURE | FUNCTION | PACKAGE | PACKAGE BODY} statement fails because of a PL/SQL compilation error.

SQLWARN(6)

This flag is no longer in use.

SQLWARN(7)

This flag is no longer in use.

SQLEXT

This string field is reserved for future use.

PL/SQL Considerations

When your precompiler program executes an embedded PL/SQL block, not all fields in the SQLCA are set. For example, if the block fetches several rows, the rows-processed count, SQLERRD(3), is set to 1, *not* the actual number of rows fetched. So, you should rely only on the SQLCODE and SQLERRM fields in the SQLCA after executing a PL/SQL block.

Getting the Full Text of Error Messages

The SQLCA can accommodate error messages up to 70 characters long. To get the full text of longer (or nested) error messages, you need the SQLGLM function. If connected to Oracle, you can call SQLGLM using the syntax

```
SQLGLM(message_buffer, buffer_size, message_length);
```

where:

message_buffer

is the text buffer in which you want Oracle to store the error message (Oracle blank-pads to the end of this buffer).

buffer_size

is an integer variable that specifies the maximum size of the buffer in bytes.

message_length

is an integer variable in which Oracle stores the actual length of the error message.

The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names. The maximum length of an error message returned by SQLGLM depends on the value you specify for *buffer_size*.

In the following example, you call SQLGLM to get an error message of up to 100 characters in length:

```
-- declare variables for function call
msg_buffer CHARACTER(100);
buf_size INTEGER;
msg_length INTEGER;
set buf_size = 100;
EXEC SQL WHENEVER SQLERROR DO sql_error;
-- other statements
```

```
ROUTINE sql_error
BEGIN
 -- get full text of error message
 SQLGLM(msg_buffer, buf_size, msg_length);
 display contents of msg_buffer;
 exit program with an error
END sql_error;
```

Notice that SQLGLM is called only when a SQL error has occurred. Always make sure SQLCODE is negative *before* calling SQLGLM. If you call SQLGLM when SQLCODE is zero, you get the message text associated with a prior SQL statement.

Using the WHENEVER Statement

By default, precompiled programs ignore Oracle error and warning conditions and continue processing if possible. To perform automatic condition checking and error handling, use the `WHENEVER` statement.

With the `WHENEVER` statement, you can specify actions to be taken when Oracle detects an error, warning condition, or "not found" condition. These actions include continuing with the next statement, calling a routine, branching to a labeled statement, or stopping.

You code the `WHENEVER` statement by using the following syntax:

```
EXEC SQL WHENEVER <condition> <action>;
```

You can have Oracle automatically check the SQLCA for any of the following conditions.

SQLWARNING

SQLWARN(0) is set because Oracle 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 Oracle assigns a truncated column value to an output host variable.

Declaring the SQLCA is optional when MODE={ANSI|ANSI14}. To use WHENEVER SQLWARNING, however, you *must* declare the SQLCA.

SQLERROR

SQLCODE has a negative value because Oracle returned an error.

NOT FOUND

SQLCODE has a value of +1403 (+100 when `MODE={ANSI|ANSI14| ANSI13}`), because Oracle could not find a row that meets the search condition of a `WHERE` clause, or a `SELECT INTO` or `FETCH` returned no rows. When `MODE={ANSI|ANSI14|ANSI13}, +100` is returned to SQLCODE after an `INSERT` of no rows.

When Oracle detects one of the preceding *conditions*, you can have your program take any of the following actions.

CONTINUE

Your program continues to run with the next statement if possible. This is the default action, equivalent to not using the WHENEVER statement. You can use it to "turn off" condition checking.

DO

Your program transfers control to an internal routine. When the end of the routine is reached, control transfers to the statement that follows the failed SQL statement.

A *routine* is any functional program unit that can be invoked, such as a COBOL paragraph or FORTRAN subroutine. In this context, separately compiled programs, such as COBOL subroutines, are *not* routines.

The usual rules for entering and exiting a routine apply. However, passing parameters to the routine is *not* allowed. Furthermore, the routine must *not* return a value.

The parameter *routine_call* is a host language invocation, as in

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

or

```
EXEC SQL -- FORTRAN
 WHENEVER <condition> DO CALL <subroutine_name> -- FORTRAN
```

GOTO

Your program branches to a labeled statement.

STOP

Your program stops running and uncommitted work is rolled back.

Be careful. The STOP action displays no messages before logging off Oracle. In Pascal, the STOP action is illegal because Pascal has no equivalent command.

Some Examples

If you want your program to

- go to *close_cursor* if a "no data found" condition occurs,
- continue with the next statement if a warning occurs, and
- go to *error_handler* if an error occurs

simply code the following WHENEVER statements before the first executable SQL statement:

```
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO error_handler;
```

The following Pro*C example uses WHENEVER...DO statements to handle specific errors:

```
EXEC SQL WHENEVER SQLERROR DO handle_insert_error;
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
 VALUES (:emp_number, :emp_name, :dept_number);
```

```
EXEC SQL WHENEVER SQLERROR DO handle_delete_error;
EXEC SQL DELETE FROM DEPT WHERE DEPTNO = :dept_number;
...
ROUTINE handle_insert_error;
 BEGIN
 IF sqlca.sqlcode = -1 THEN -- duplicate key value
 ...
 ELSEIF sqlca.sqlcode = -1401 THEN -- value too large
 ...
 ENDIF;
 ...
 END;
ROUTINE handle_delete_error;
 BEGIN
 IF sqlca.sqlerrd(3) = 0 THEN -- no rows deleted
 ...
 ELSE
 ...
 ENDIF;
 ...
 END;
...
```

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

Scope

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. Therefore, you should code the WHENEVER statement before the first executable SQL statement you want to test.

A WHENEVER statement stays in effect until superseded by another WHENEVER statement checking for the same condition.

In the example , the first WHENEVER SQLERROR statement is superseded by a second, and so applies only to the CONNECT statement. The second WHENEVER SQLERROR statement applies to both the UPDATE and DROP statements, despite the flow of control from *step1* to *step3*.

```
step1:
EXEC SQL WHENEVER SQLERROR STOP;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
 ...
 GOTO step3;
step2:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL UPDATE EMP SET SAL = SAL * 1.10;
 ...
step3:
 EXEC SQL DROP INDEX EMP_INDEX;
 ...
```

Guidelines

The following guidelines will help you avoid some common pitfalls.

Placing the Statements. In general, code a WHENEVER statement before the first executable SQL statement in your program. This ensures that all ensuing errors are trapped because WHENEVER statements stay in effect to the end of a file.

Handling End-of-Data Conditions. Your program should be prepared to handle an end-of-data condition when using a cursor to fetch rows. If a FETCH returns no data, the program should branch to a labeled section of code where a CLOSE command is issued, as follows:

```
SQL WHENEVER NOT FOUND GOTO no_more;
...
no_more:
 ...
 EXEC SQL CLOSE my_cursor;
 ...
```

Avoiding Infinite Loops. If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes an executable SQL statement, your program might enter an infinite loop if the SQL statement fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the SQL statement, as shown in the following example:

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 ...
```

Without the WHENEVER SQLERROR CONTINUE statement, a ROLLBACK error would invoke the routine again, starting an infinite loop.

Careless use of WHENEVER can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets NOT FOUND because no rows meet the search condition:

```
-- improper use of WHENEVER
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 ...
ENDLOOP;
no_more:
 EXEC SQL DELETE FROM EMP WHERE EMPNO = :emp_number;
 ...
```

In the next example, you handle the NOT FOUND condition properly by resetting the GOTO target:

```
-- proper use of WHENEVER
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 ...
ENDLOOP;
no_more:
 EXEC SQL WHENEVER NOT FOUND GOTO no_match;
 EXEC SQL DELETE FROM EMP WHERE EMPNO = :emp_number;
 ...
```

```
no_match:
 ...
```

Maintaining Addressability. With host languages that allow local and global identifiers, make sure all SQL statements governed by a `WHENEVER GOTO` statement can branch to the `GOTO` label. The following code results in a compile-time error because *labelA* in FUNC1 is not within the scope of the INSERT statement in FUNC2:

```
FUNC1
 BEGIN
 EXEC SQL WHENEVER SQLERROR GOTO labelA;
 EXEC SQL DELETE FROM EMP WHERE DEPTNO = :dept_number;
 ...
 labelA:
 ...
 END;
FUNC2
 BEGIN
 EXEC SQL INSERT INTO EMP (JOB) VALUES (:job_title);
 ...
 END;
```

The label to which a `WHENEVER GOTO` statement branches must be in the same precompilation file as the statement.

Returning after an Error. If your program must return after handling an error, use the `DO` *routine_call* action. Alternatively, you can test the value of SQLCODE, as shown in the following example:

```
EXEC SQL UPDATE EMP SET SAL = SAL * 1.10;
IF sqlca.sqlcode < 0 THEN
 -- handle error
EXEC SQL DROP INDEX EMP_INDEX;
...
```

Just make sure no `WHENEVER GOTO` or `WHENEVER STOP` statement is active.

Getting the Text of SQL Statements

In many precompiler 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 run-time library, returns the following information:

- The text of the most recently parsed SQL statement
- The Length of the statement
- A Function code (refer to Table 8–8 for the SQL command used in the statement

You can call SQLGLS after issuing a static SQL statement. With dynamic SQL Method 1, you can call SQLGLS after the SQL statement is executed. With dynamic SQL Method 2, 3, or 4, you can call SQLGLS after the statement is prepared.

To call SQLGLS, you use the following syntax:

```
SQLGLS(SQLSTM, STMLEN, SQLFC)
```

Table 8–7 shows the host-language datatypes available for the parameters in the SQLGLS argument list.

Table 8–7 SQLGLS Parameter Datatypes

Parameter	Language	Datatype
SQLSTM	COBOL	PIC X(*n*)
	FORTRAN	CHARACTER*n
STMLEN, SQLFC	COBOL	PIC S9(9) COMP
	FORTRAN	INTEGER*4

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

The parameter SQLSTM is a blank-padded (not null-terminated) character buffer that holds the returned text of the SQL statement. Your program must statically declare the buffer or dynamically allocate memory for it.

The length parameter STMLEN is a four-byte integer. Before calling SQLGLS, set this parameter to the actual size (in bytes) of the SQLSTM buffer. When SQLGLS returns, the SQLSTM buffer contains the SQL statement text blank padded to the length of the buffer. STMLEN returns the actual number of bytes in the returned statement text, not counting the blank padding. However, STMLEN returns a zero if an error occurred.

Some possible errors follow:

- No SQL statement was parsed.
- You passed an invalid parameter (for example, a negative length value).
- An internal exception occurred in SQLLIB.

The parameter SQLFC is a four-byte integer that returns the SQL function code for the SQL command in the statement. Table 8–8 shows the function code for each SQL command.

SQLGLS does not return statements that contain the following commands:

- CONNECT
- COMMIT
- ROLLBACK
- RELEASE
- FETCH

There are no SQL function codes for these statements.

Table 8–8 SQL Command Function Codes

Code	SQL Function	Code	SQL Function
01	CREATE TABLE	39	AUDIT
02	SET ROLE	40	NOAUDIT
03	INSERT	41	ALTER INDEX
04	SELECT	42	CREATE EXTERNAL DATABASE
05	UPDATE	43	DROP EXTERNAL DATABASE
06	DROP ROLE	44	CREATE DATABASE
07	DROP VIEW	45	ALTER DATABASE

Table 8–8 SQL Command Function Codes

Code	SQL Function	Code	SQL Function
08	DROP TABLE	46	CREATE ROLLBACK SEGMENT
09	DELETE	47	ALTER ROLLBACK SEGMENT
10	CREATE VIEW	48	DROP ROLLBACK SEGMENT
11	DROP USER	49	CREATE TABLESPACE
12	CREATE ROLE	50	ALTER TABLESPACE
13	CREATE SEQUENCE	51	DROP TABLESPACE
14	ALTER SEQUENCE	52	ALTER SESSION
15	(not used)	53	ALTER USER
16	DROP SEQUENCE	54	COMMIT
17	CREATE SCHEMA	55	ROLLBACK
18	CREATE CLUSTER	56	SAVEPOINT
19	CREATE USER	57	CREATE CONTROL FILE
20	CREATE INDEX	58	ALTER TRACING
21	DROP INDEX	59	CREATE TRIGGER
22	DROP CLUSTER	60	ALTER TRIGGER
23	VALIDATE INDEX	61	DROP TRIGGER
24	CREATE PROCEDURE	62	ANALYZE TABLE
25	ALTER PROCEDURE	63	ANALYZE INDEX
26	ALTER TABLE	64	ANALYZE CLUSTER
27	EXPLAIN	65	CREATE PROFILE
28	GRANT	66	DROP PROFILE
29	REVOKE	67	ALTER PROFILE
30	CREATE SYNONYM	68	DROP PROCEDURE
31	DROP SYNONYM	69	(not used)
32	ALTER SYSTEM SWITCH LOG	70	ALTER RESOURCE COST
33	SET TRANSACTION	71	CREATE SNAPSHOT LOG
34	PL/SQL EXECUTE	72	ALTER SNAPSHOT LOG
35	LOCK TABLE	73	DROP SNAPSHOT LOG
36	(not used)	74	CREATE SNAPSHOT
37	RENAME	75	ALTER SNAPSHOT
38	COMMENT	76	DROP SNAPSHOT

Using the Oracle Communications Area

In the same way the SQLCA handles standard SQL communications; the Oracle Communications Area (ORACA) handles Oracle communications. When you need more information about run-time errors and status changes than the SQLCA provides,

use the ORACA. It contains an extended set of diagnostic tools. However, use of the ORACA is optional because it adds to run-time overhead.

Besides helping you to diagnose problems, the ORACA lets you monitor your program's use of Oracle resources such as the SQL Statement Executor and the cursor cache.

In host languages that allow local and global declarations, your program can have more than one ORACA. For example, it might have one global ORACA and several local ones. Access to a local ORACA is limited by its scope within the program. Oracle returns information only to the "active" ORACA. The information is available only after a commit or rollback.

Declaring the ORACA

To declare the ORACA, simply include it (using an EXEC SQL `INCLUDE` statement) in your host-language source file as follows:

```
* Include the Oracle Communications Area (ORACA).
 EXEC SQL INCLUDE ORACA
```

The ORACA must be declared *outside* the Declare Section.

When you precompile your program, the `INCLUDE` ORACA statement is replaced by several program variable declarations. These declarations allow Oracle to communicate with your program.

Enabling the ORACA

To enable the ORACA, you must specify the ORACA option, either on the command line with

```
ORACA=YES
```

or inline with

```
EXEC ORACLE OPTION (ORACA=YES);
```

Then, you must choose appropriate run-time options by setting flags in the ORACA.

What's in the ORACA?

The ORACA contains option settings, system statistics, and extended diagnostics such as

- SQL statement text (you can specify when to save the text)
- name of the file in which an error occurred
- location of the error in a file
- cursor cache errors and statistics

Figure 8–3 shows all the variables in the ORACA. To see the ORACA structure and variable names in a particular host language, refer to your supplement to this Guide.

Figure 8–3 ORACA Variables

ORACAID	Character string "ORACA"
ORACABC	Length of ORACA data structure in bytes
ORACCHF	Cursor cache conssistency flag
ORADBGF	Master debug flag
ORAHCHF	Heap consistency flag
ORASTXTF	Save SQL statement flag
ORASTXT	Subrecord for storing SQL statement
ORASTXTL	Length of current SQL statement
ORASTXTC	Text of current SQL statement
ORASFNM	Subrecord for storing filename
ORASFNML	Length of filename
ORASFNMC	Name of file containing current SQL statement
ORASLNR	Line in file at or near current SQL statement
ORAHOC	Highest MAXOPENCURSORS requested
ORAMOC	Maximum open cursors required
ORACOC	Current number of cursors used
ORANOR	Number of cursor cache reassignments
ORANPR	Number of SQL statement parses
ORANEX	Number of SQL statement executions

Choosing Run-time Options

The ORACA includes several option flags. Setting these flags by assigning them nonzero values enables

- 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 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 Oracle run-time library does the consistency checking and might issue error messages, which are listed in *Oracle Database Error Messages*. They are returned to the SQLCA just like Oracle error messages.

This flag has the following settings:

0

Disable cache consistency checking (the default).

1

Enable cache consistency checking.

ORADBGF

This master flag lets you choose all the DEBUG options. It has the following settings:

0

Disable all DEBUG operations (the default).

1

Enable all DEBUG operations.

ORAHCHF

If the master DEBUG flag (ORADBGF) is set, this flag tells the Oracle run-time library to check the heap for consistency every time the precompiler dynamically allocates or frees memory. This is useful for detecting program bugs that upset memory.

This flag must be set before the `CONNECT` command is issued and, once set, cannot be cleared; subsequent change requests are ignored. It has the following settings:

0

Disable heap consistency checking (the default).

1

Enable heap consistency checking.

ORASTXTF

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

0

Never save the SQL statement text (the default).

1

Save the SQL statement text on SQLERROR only.

2

Save the SQL statement text on SQLERROR or SQLWARNING.

3

Always save the SQL statement text.

The SQL statement text is saved in the ORACA subrecord named ORASTXT.

Diagnostics

The ORACA provides an enhanced set of diagnostics; the following variables help you to locate errors quickly.

ORASTXT

This subrecord helps you find faulty SQL statements. It lets you save the text of the last SQL statement parsed by Oracle. It contains the following two fields:

ORASTXTL

This integer field holds the length of the current SQL statement.

ORASTXTC

This string field holds the text of the current SQL statement. At most, the first 70 characters of text are saved.

Statements parsed by the precompiler, such as `CONNECT`, `FETCH`, and `COMMIT`, are *not* saved in the ORACA.

ORASFNM

This subrecord identifies the file containing the current SQL statement and so helps you find errors when multiple files are precompiled for one application. It contains the following two fields:

ORASFNML

This integer field holds the length of the filename stored in ORASFNMC.

ORASFNMC

This string field holds the filename. At most, the first 70 characters are stored.

ORASLNR

This integer field identifies the line at (or near) which the current SQL statement can be found.

Cursor Cache Statistics

The variables 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 Oracle cursors required by your program. This number can be higher than ORAHOC if `MAXOPENCURSORS` was set too low, which forced the precompiler to extend the cursor cache.

ORACOC

This integer field records the current number of open Oracle cursors required by your program.

ORANOR

This integer field records the number of cursor cache reassignments required by your program. This number shows the degree of "thrashing" in the cursor cache and should be kept as low as possible.

ORANPR

This integer field records the number of SQL statement parses required by your program.

ORANEX

This integer field records the number of SQL statement executions required by your program. The ratio of this number to the ORANPR number should be kept as high as possible. In other words, avoid unnecessary reparsing. For help, refer to Appendix C, "Performance Tuning".

An Example

The following program prompts for a department number, inserts the name and salary of each employee in that department into one of two tables, then displays diagnostic information from the ORACA:

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 emp_name INTEGER;
 dept_number INTEGER;
 salary REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE ORACA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR DO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
EXEC ORACLE OPTION (ORACA=YES);
-- set flags in the ORACA
set oraca.oradbgf = 1; -- enable debug operations
set oraca.oracchf = 1; -- enable cursor cache consistency check
set oraca.orastxtf = 3; -- always save the SQL statement
display 'Department number? ';
read dept_number;
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL + NVL(COMM,0)
 FROM EMP
 WHERE DEPTNO = :dept_number;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO no_more;
rLOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 IF salary < 2500 THEN
 EXEC SQL INSERT INTO PAY1 VALUES (:emp_name, :salary);
 ELSE
 EXEC SQL INSERT INTO PAY2 VALUES (:emp_name, :salary);
```

```
 ENDIF;
ENDLOOP;
ROUTINE no_more
BEGIN
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL COMMIT WORK RELEASE;
 display 'Last SQL statement: ', oraca.orastxt.orastxtc;
 display '... at or near line number: ', oraca.oraslnr;
 display
 display ' Cursor Cache Statistics';
 display '-------------------------------------------';
 display 'Maximum value of MAXOPENCURSORS ', oraca.orahoc;
 display 'Maximum open cursors required: ', oraca.oramoc;
 display 'Current number of open cursors: ', oraca.oracoc;
 display 'Number of cache reassignments: ', oraca.oranor;
 display 'Number of SQL statement parses: ', oraca.oranpr;
 display 'Number of SQL statement executions: ', oraca.oranex;
 exit program;
END no_more;
ROUTINE sql_error
BEGIN
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Last SQL statement: ', oraca.orastxt.orastxtc;
 display '... at or near line number: ', oraca.oraslnr;
 display
 display ' Cursor Cache Statistics';
 display '-------------------------------------------';
 display 'Maximum value of MAXOPENCURSORS ', oraca.orahoc;
 display 'Maximum open cursors required: ', oraca.oramoc;
 display 'Current number of open cursors: ', oraca.oracoc;
 display 'Number of cache reassignments: ', oraca.oranor;
 display 'Number of SQL statement parses: ', oraca.oranpr;
 display 'Number of SQL statement executions: ', oraca.oranex;
 exit program with an error;
END sql_error;
```

9

Using Host Arrays

This chapter describes the following:

- What Is a Host Array?
- Why Use Arrays?
- Declaring Host Arrays
- Using Arrays in SQL Statements
- Selecting into Arrays
- Using Indicator Arrays
- Using the FOR Clause
- Using the WHERE Clause
- Mimicking the CURRENT OF Clause
- Using SQLERRD(3)

This chapter looks at using arrays to simplify coding and improve program performance. You learn how to manipulate Oracle data using arrays, how to operate on all the elements of an array with a single SQL statement, and how to limit the number of array elements processed. The following questions are answered:

- What is a host array?
- Why use arrays?
- How are host arrays declared?
- How are arrays used in SQL statements?

What Is a Host Array?

An *array* is a collection of related data items, called *elements*, associated with a single variable name. When declared as a host variable, the array is called a *host array*. Likewise, an indicator variable declared as an array is called an *indicator array*. An indicator array can be associated with any host array.

Why Use Arrays?

Arrays can ease programming and offer improved performance. When writing an application, you are usually faced with the problem of storing and manipulating large collections of data. Arrays simplify the task of naming and referencing the individual items in each collection.

Using arrays can boost the performance of your application. Arrays let you manipulate an entire collection of data items with a single SQL statement. Thus, Oracle communication overhead is reduced markedly, especially in a networked environment. For example, suppose you want to insert information about 300 employees into the EMP table. Without arrays, your program must do 300 individual INSERTs--one for each employee. With arrays, only one INSERT need be done.

Declaring Host Arrays

You declare host arrays in the Declare Section like simple host variables. You also *dimension* (set the size of) host arrays in the Declare Section. In the following example, you declare three host arrays and dimension them with 50 elements:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (50) CHARACTER(20);
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
```

Dimensioning Arrays

The maximum dimension of a host array is 32,767 elements. If you use a host array that exceeds the maximum, you get a "parameter out of range" run-time error. If you use multiple host arrays in a single SQL statement, their dimensions should be the same. Otherwise, an "array size mismatch" warning message is issued at precompile time. If you ignore this warning, the precompiler uses the *smallest* dimension for the SQL operation.

Restrictions

You cannot declare host arrays of pointers. Also, host arrays that might be referenced in a SQL statement are limited to one dimension. So, the two-dimensional array declared in the following example is *invalid*:

```
EXEC SQL BEGIN DECLARE SECTION;
 hi_lo_scores (25, 25) INTEGER; -- not allowed
EXEC SQL END DECLARE SECTION;
```

Using Arrays in SQL Statements

The Oracle Precompilers allow the use of host arrays in data manipulation statements. You can use host arrays as input variables in the INSERT, UPDATE, and DELETE statements and as output variables in the INTO clause of SELECT and FETCH statements.

Note that when MODE=ANSI14, array operations are *not* allowed. In other words, you can reference host arrays in a SQL statement only when MODE={ANSI|ANSI13|ORACLE}.

The syntax used for host arrays and simple host variables is nearly the same. One difference is the optional FOR clause, which lets you control array processing. Also, there are restrictions on mixing host arrays and simple host variables in a SQL statement.

The following sections illustrate the use of host arrays in data manipulation statements.

Selecting into Arrays

You can use host arrays as output variables in the SELECT statement. If you know the maximum number of rows the select will return, simply dimension the host arrays with that number of elements. In the following example, you select directly into three host arrays. Knowing the select will return no more than 50 rows, you dimension the arrays with 50 elements:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (50) CHARACTER(20);
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT ENAME, EMPNO, SAL
 INTO :emp_name, :emp_number, :salary
 FROM EMP
 WHERE SAL > 1000;
```

In this example, the SELECT statement returns up to 50 rows. If there are fewer than 50 eligible rows or you want to retrieve only 50 rows, this method will suffice. However, if there are more than 50 eligible rows, you cannot retrieve all of them this way. If you reexecute the SELECT statement, it just returns the first 50 rows again, even if more are eligible. You must either dimension a larger array or declare a cursor for use with the FETCH statement.

If a SELECT INTO statement returns more rows than the number of elements you dimensioned, Oracle issues the error message

```
SQL-02112: SELECT...INTO returns too many rows
```

unless you specify SELECT_ERROR=NO. For more information about the option SELECT_ERROR, refer to "SELECT_ERROR" on page 6-41

Batch Fetches

If you do not know the maximum number of rows a select will return, you can declare and open a cursor_name fetch from it in "batches." Batch fetches within a loop let you retrieve a large number of rows with ease. Each fetch returns the next batch of rows from the current active set. In the following example, you fetch in 20-row batches:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number (20) INTEGER;
 salary (20) REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT EMPNO, SAL FROM EMP;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_number, :salary;
 -- process batch of rows
ENDLOOP;
```

Number of Rows Fetched

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

- The end of the active set is reached. The "no data found" Oracle warning code is returned to SQLCODE in the SQLCA. For example, this happens if you fetch into an array of dimension 100 but only 20 rows are returned.
- Fewer than a full batch of rows remain to be fetched. For example, this happens if you fetch 70 rows into an array of dimension 20 because after the third fetch, only 10 rows remain to be fetched.
- An error is detected while processing a row. The fetch fails and the applicable Oracle error code is returned to SQLCODE.

The cumulative number of rows returned can be found in the third element of SQLERRD in the SQLCA, called SQLERRD(3) in this guide. This applies to each open cursor. In the following example, notice how the status of each cursor is maintained separately:

```
EXEC SQL OPEN cursor1;
EXEC SQL OPEN cursor2;
EXEC SQL FETCH cursor1 INTO :array_of_20;
 -- now running total in SQLERRD(3) is 20
EXEC SQL FETCH cursor2 INTO :array_of_30;
 -- now running total in SQLERRD(3) is 30, not 50
EXEC SQL FETCH cursor1 INTO :array_of_20;
 -- now running total in SQLERRD(3) is 40 (20 + 20)
EXEC SQL FETCH cursor2 INTO :array_of_30;
 -- now running total in SQLERRD(3) is 60 (30 + 30)
```

Restrictions

Using host arrays in the `WHERE` clause of a `SELECT` statement is allowed only in a subquery. (For example, refer to "Using the WHERE Clause".) Also, you cannot mix simple host variables with host arrays in the INTO clause of a `SELECT` or `FETCH` statement; if any of the host variables is an array, all must be arrays. Table 9–1 shows which uses of host arrays are valid in a `SELECT INTO` statement.

Table 9–1 Valid Host Arrays for SELECT INTO

INTO Clause	WHERE Clause	Valid?
array	array	no
scalar	scalar	yes
array	scalar	yes
scalar	array	no

Fetching Nulls

If you select or fetch a null into a host array that lacks an indicator array, Oracle stops processing, sets SQLERRD(3) to the number of rows processed, and issues the following error message:

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

To learn how to find nulls and truncated values, refer to "Using Indicator Variables".

Fetching Truncated Values

When DBMS=V7, if you select or fetch a truncated column value into a host array that lacks an indicator array, Oracle stops processing, sets SQLERRD(3) to the number of rows processed, and issues the following error message:

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

You can check SQLERRD(3) for the number of rows processed before the truncation occurred. The rows-processed count includes the row that caused the truncation error.

When MODE=ANSI, truncation is not considered an error, so Oracle continues processing.

Again, when doing array selects and fetches, always use indicator arrays. That way, if Oracle assigns one or more truncated column values to an output host array, you can find the original lengths of the column values in the associated indicator array.

Inserting with Arrays

You can use host arrays as input variables in an `INSERT` statement. Just make sure your program populates the arrays with data before executing the `INSERT` statement. If some elements in the arrays are irrelevant, you can use the `FOR` clause to control the number of rows inserted. Refer to Using the FOR Clause.

An example of inserting with host arrays follows:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (50) CHARACTER(20);
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
 VALUES (:emp_name, :emp_number, :salary);
```

The cumulative number of rows inserted can be found in SQLERRD(3).

Although functionally equivalent to the following statement, the `INSERT` statement in the last example is much more efficient because it issues only one call to Oracle:

```
FOR i = 1 TO array_dimension
 EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
 VALUES (:emp_name[i], :emp_number[i], :salary[i]);
ENDFOR;
```

In this imaginary example (imaginary because host variables *cannot* be subscripted in a SQL statement), you use a FOR loop to access all array elements in sequential order.

Restrictions

You cannot use an array of pointers in the VALUES clause of an `INSERT` statement; all array elements must be data items. Also, mixing simple host variables with host arrays in the VALUES clause of an `INSERT` statement is *not* allowed; if any of the host variables is an array, all must be arrays.

Updating with Arrays

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

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

```
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
EXEC SQL UPDATE EMP SET SAL = :salary WHERE EMPNO = :emp_number;
```

The cumulative number of rows updated can be found in SQLERRD(3). The number does *not* include rows processed by an update cascade.

If some elements in the arrays are irrelevant, you can use the FOR clause to limit the number of rows updated.

The last example showed a typical update using a unique key (*emp_number*). Each array element qualified just one row for updating. In the following example, each array element qualifies multiple rows:

```
EXEC SQL BEGIN DECLARE SECTION;
 job_title (10) CHARACTER(10);
 commission (50) REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
EXEC SQL UPDATE EMP SET COMM = :commission WHERE JOB = :job_title;
```

Restrictions: Mixing simple host variables with host arrays in the `SET` or `WHERE` clause of an `UPDATE` statement is *not* allowed. If any of the host variables is an array, all must be arrays. Furthermore, if you use a host array in the SET clause, you *must* use one in the `WHERE` clause. However, their dimensions and datatypes need not match.

You cannot use host arrays with the `CURRENT` OF clause in an UPDATE statement. For an alternative, refer to Mimicking the CURRENT OF Clause.

Table 9–2 shows which uses of host arrays are valid in an `UPDATE` statement:

Table 9–2 Valid Host Arrays for UPDATE

SET Clause	WHERE Clause	Valid?
array	array	yes
scalar	scalar	yes
array	scalar	no
scalar	array	no

Deleting with Arrays

You can also use host arrays as input variables in a `DELETE` statement. It is like executing the `DELETE` statement repeatedly using successive elements of the host array in the `WHERE` clause. Thus, each execution might delete zero, one, or more rows from the table. An example of deleting with host arrays follows:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 emp_number (50) INTEGER;
 EXEC SQL END DECLARE SECTION;
-- populate the host array
EXEC SQL DELETE FROM EMP WHERE EMPNO = :emp_number;
```

The cumulative number of rows deleted can be found in SQLERRD(3). 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 array element qualified just one row for deletion. In the following example, each array element qualifies multiple rows:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 job_title (10) CHARACTER(10);
EXEC SQL END DECLARE SECTION;
-- populate the host array
EXEC SQL DELETE FROM EMP WHERE JOB = :job_title;
```

Restrictions

Mixing simple host variables with host arrays in the `WHERE` clause of a `DELETE` statement is *not* allowed; if any of the host variables is an array, all must be arrays. Also, you cannot use host arrays with the `CURRENT OF` clause in a `DELETE` statement. For an alternative, refer to "Mimicking the CURRENT OF Clause".

Using Indicator Arrays

You use indicator arrays to assign nulls to input host arrays and to detect null or truncated values in output host arrays. The following example shows how to insert with indicator arrays:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number (50) INTEGER;
 dept_number (50) INTEGER;
 commission (50) REAL;
 ind_comm (50) SMALLINT; -- indicator array
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
-- populate the indicator array; to insert a null into
-- the COMM column, assign -1 to the appropriate element in
-- the indicator array
EXEC SQL INSERT INTO EMP (EMPNO, DEPTNO, COMM)
 VALUES (:emp_number, :dept_number, :commission:ind_comm);
```

The dimension of the indicator array cannot be smaller than the dimension of the host array.

Using the FOR Clause

You can use the optional FOR clause to set the number of array elements processed by any of the following SQL statements:

- `DELETE`
- `EXECUTE`
- `FETCH`
- `INSERT`
- `OPEN`
- `UPDATE`

The `FOR` clause is especially useful in `UPDATE`, `INSERT`, and `DELETE` statements. With these statements, you might not want to use the entire array. The `FOR` clause lets you limit the elements used to just the number you need, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (100) CHARACTER(20);
 salary (100) REAL;
 rows_to_insert INTEGER;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
set rows_to_insert = 25; -- set FOR-clause variable
EXEC SQL FOR :rows_to_insert -- will process only 25 rows
 INSERT INTO EMP (ENAME, SAL)
 VALUES (:emp_name, :salary);
```

The `FOR` clause must use an integer host variable to count array elements. For example, the following statement is illegal:

```
EXEC SQL FOR 25 -- illegal
 INSERT INTO EMP (ENAME, EMPNO, SAL)
 VALUES (:emp_name, :emp_number, :salary);
```

The `FOR`-clause variable specifies the number of array elements to be processed. Make sure the number does not exceed the smallest array dimension. Also, the number must be positive. If it is negative or zero, no rows are processed.

Restrictions

Two restrictions keep `FOR` clause semantics clear.: You cannot use the FOR clause in a `SELECT` statement or with the `CURRENT OF` clause.

In a SELECT Statement

If you use the `FOR` clause in a `SELECT` statement, you get the following error message:

```
PCC-E-0056: FOR clause not allowed on SELECT statement at ...
```

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

```
EXEC SQL FOR :limit FETCH emp_cursor INTO ...
```

With the CURRENT OF Clause

You can use the `CURRENT OF` clause in an `UPDATE` or `DELETE` statement to refer to the latest row returned by a `FETCH` statement, as the following example shows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL FROM EMP WHERE EMPNO = :emp_number;
...
EXEC SQL OPEN emp_cursor;
...
EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
...
EXEC SQL UPDATE EMP SET SAL = :new_salary
 WHERE CURRENT OF emp_cursor;
```

However, you cannot use the FOR clause with the `CURRENT OF` clause. The following statements are invalid because the only logical value of *limit* is 1 (you can only update or delete the current row once):

```
EXEC SQL FOR :limit UPDATE EMP SET SAL = :new_salary
 WHERE CURRENT OF emp_cursor;
...
EXEC SQL FOR :limit DELETE FROM EMP
 WHERE CURRENT OF emp_cursor;
```

Using the WHERE Clause

Oracle treats a SQL statement containing host arrays of dimension *n* like the same SQL statement executed *n* times with *n* different scalar variables (the individual array elements). The precompiler issues the following error message only when such treatment is ambiguous:

```
PCC-S-0055: Array <name> not allowed as bind variable at ...
```

For example, assuming the declarations

```
EXEC SQL BEGIN DECLARE SECTION;
 mgr_number (50) INTEGER;
 job_title (50) CHARACTER(20);
EXEC SQL END DECLARE SECTION;
```

it would be ambiguous if the statement

```
EXEC SQL SELECT MGR INTO :mgr_number FROM EMP
 WHERE JOB = :job_title;
```

were treated like the imaginary statement

```
FOR i = 1 TO 50
 SELECT MGR INTO :mgr_number[i] FROM EMP
 WHERE JOB = :job_title[i];
ENDFOR;
```

because multiple rows might meet the `WHERE`-clause search condition, but only one output variable is available to receive data. Therefore, an error message is issued.

However, it would not be ambiguous if the statement

```
EXEC SQL UPDATE EMP SET MGR = :mgr_number
 WHERE EMPNO IN (SELECT EMPNO FROM EMP WHERE JOB = :job_title);
```

were treated like the imaginary statement

```
FOR i = 1 TO 50
 UPDATE EMP SET MGR = :mgr_number[i]
 WHERE EMPNO IN
 (SELECT EMPNO FROM EMP WHERE JOB = :job_title[i]);
ENDFOR;
```

because there is a *mgr_number* in the `SET` clause for each row matching *job_title* in the `WHERE` clause, even if each *job_title* matches multiple rows. All rows matching each *job_title* can be `SET` to the same *mgr_number*. So, no error message is issued.

Mimicking the CURRENT OF Clause

You use the `CURRENT OF` *cursor* clause in a `DELETE` or `UPDATE` statement to refer to the latest row fetched from the cursor. However, you cannot use `CURRENT OF` with host arrays. Instead, select the `ROWID` of each row, then use that value to identify the current row during the update or delete. An example follows:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (25) CHARACTER(20);
 job_title (25) CHARACTER(15);
 old_title (25) CHARACTER(15);
 row_id (25) CHARACTER(18);
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, JOB, ROWID FROM EMP;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
...
LOOP
 EXEC SQL FETCH emp_cursor
 INTO :emp_name, :job_title, :row_id;
 ...
 EXEC SQL DELETE FROM EMP
 WHERE JOB = :old_title AND ROWID = :row_id;
 EXEC SQL COMMIT WORK;
ENDLOOP;
```

However, the fetched rows are *not* locked because no `FOR UPDATE OF` clause is used. So, you might get inconsistent results if another user changes a row after you read it but before you delete it.

Using SQLERRD(3)

For `INSERT`, `UPDATE`, `DELETE`, and `SELECT INTO` statements, SQLERRD(3) records the number of rows processed. For `FETCH` statements, it records the cumulative sum of rows processed.

When using host arrays with `FETCH`, to find the number of rows returned by the most recent iteration, subtract the current value of SQLERRD(3) from its previous value (stored in another variable). In the following example, you determine the number of rows returned by the most recent fetch:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number (100) INTEGER;
 emp_name (100) CHARACTER(20);
EXEC SQL END DECLARE SECTION;
...
 rows_to fetch INTEGER;
 rows_before INTEGER;
 rows_this_time INTEGER;
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT EMPNO, ENAME
 FROM EMP
 WHERE DEPTNO = 30;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...
-- initialize loop variables
set rows_to_fetch = 20; -- number of rows in each "batch"
set rows_before = 0; -- previous value of sqlerrd(3)
set rows_this_time = 20;
WHILE rows_this_time = rows_to_fetch
```

```
 LOOP
 EXEC SQL FOR :rows_to_fetch
 FETCH emp_cursor
 INTO :emp_number, :emp_name;
 set rows_this_time = sqlca.sqlerrd(3) - rows_before;
 set rows_before = sqlca.sqlerrd(3);
 ENDLOOP;
ENDWHILE;
```

SQLERRD(3) is also useful when an error occurs during an array operation. Processing stops at the row that caused the error, so SQLERRD(3) gives the number of rows processed successfully.

10

Using Dynamic SQL

This chapter describes the following sections:

- What Is Dynamic SQL?
- Advantages and Disadvantages of Dynamic SQL
- When to Use Dynamic SQL
- Requirements for Dynamic SQL Statements
- How Dynamic SQL Statements Are Processed
- Methods for Using Dynamic SQL
- Using Method 1
- Using Method 2
- Using Method 3
- Using Method 4
- Using the DECLARE STATEMENT Statement
- Using PL/SQL

This chapter shows you how to use dynamic SQL, an advanced programming technique that adds flexibility and functionality to your applications. After weighing the advantages and disadvantages of dynamic SQL, you learn four methods from simple to complex for writing programs that accept and process SQL statements "on the fly" at run time. You learn the requirements and limitations of each method and how to choose the right method for a given job.

What Is Dynamic SQL?

Most database applications do a specific job. For example, a simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables. In this case, you know the makeup of the `UPDATE` statement at 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 run-time 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 SQL statement, but *not* contain the EXEC SQL clause, host-language delimiters or statement terminator, or any of the following embedded SQL commands:

- `CLOSE`
- `DECLARE`
- `DESCRIBE`
- `EXECUTE`
- `FETCH`
- `INCLUDE`
- `OPEN`
- `PREPARE`
- `WHENEVER`

In most cases, the character string can contain *dummy* host variables. They hold places in the SQL statement for actual host variables. Because dummy host variables are just placeholders, you do not declare them and can name them anything you like. For example, Oracle makes no distinction between the following two strings:

```
'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'
```

How Dynamic SQL Statements Are Processed

Typically, an application program prompts the user for the text of a SQL statement and the values of host variables used in the statement. Then Oracle *parses* the SQL statement. That is, Oracle examines the SQL statement to make sure it follows syntax rules and refers to valid database objects. Parsing also involves checking database access rights, reserving needed resources, and finding the optimal access path.

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

Then Oracle *executes* the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.

The SQL statement can be executed repeatedly using new values for the host variables.

Methods for Using Dynamic SQL

This section introduces four methods you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, then offers guidelines for choosing the right method. Later sections describe how to use the methods. In addition, you can find sample host-language programs in your supplement to this Guide.

The four methods are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and so on. However, each method is most useful for handling a certain kind of SQL statement, as Table 10–1 shows.

Table 10–1 Dynamic SQL Method Applicability

Method	Kind of SQL Statement
1	nonquery without input host variables
2	nonquery with known number of input host variables
3	query with known number of select-list items and input host variables
4	query with unknown number of select-list items or input host variables

The term *select-list item* includes column names and expressions.

Method 1

This method lets your program accept or build a dynamic SQL statement, then immediately execute it using the `EXECUTE IMMEDIATE` command. The SQL statement must not be a query (`SELECT` statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

```
'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO scott'
```

With Method 1, the SQL statement is parsed every time it is executed (unless you specify HOLD_CURSOR=YES).

Method 2

This method lets your program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands .The SQL statement must not be a query. The number of placeholders 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 (:emp_name, :job_title)'
'DELETE FROM EMP WHERE EMPNO = :emp_number'
```

With Method 2, the SQL statement is parsed just once (unless you specify RELEASE_CURSOR=YES), but it can be executed many times with different values for the host variables. SQL data definition statements such as CREATE are executed when they are PREPAREd.

Method 3

This method lets your program accept or build a dynamic query, then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands. The number of select-list items, the number of placeholders 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 = :dept_number'
```

Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors (discussed in "Using Method 4"). The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'
'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables.

Guidelines

With all four methods, you must store the dynamic SQL statement in a character string, which must be a host variable or quoted literal. When you store the SQL statement in the string, omit the keywords EXEC SQL and the statement terminator.

With Methods 2 and 3, the number of placeholders for input host variables and the datatypes of the input host variables must be known at 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 is to be executed repeatedly by Method 1, use Method 2 instead to avoid reparsing for each execution.

Method 4 provides maximum flexibility, but requires complex coding and a full understanding of dynamic SQL concepts. In general, use Method 4 only if you cannot use Methods 1, 2, or 3. The decision logic in Figure 10–1 will help you choose the correct method.

Avoiding Common Errors

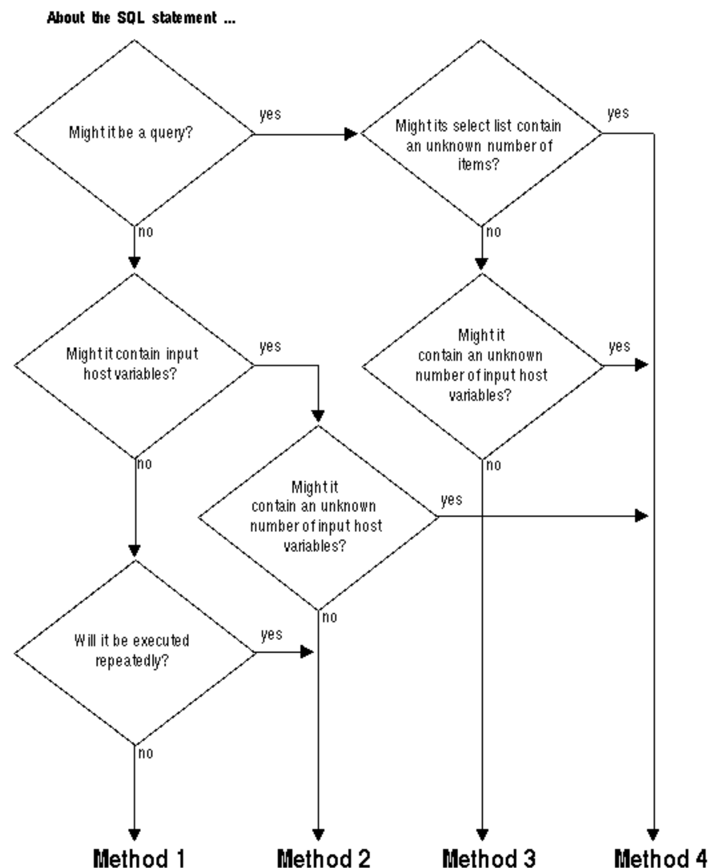
If you use a character array to store the dynamic SQL statement, blank-pad the array before storing the SQL statement. That way, you clear extraneous characters. This is especially important when you reuse the array for different SQL statements. As a rule, always initialize (or reinitialize) the host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

Do not null-terminate the host string. Oracle does not recognize the null terminator as an end-of-string sentinel. Instead, Oracle treats it as part of the SQL statement.

If you use a `VARCHAR` variable to store the dynamic SQL statement, make sure the length of the `VARCHAR` is set (or reset) correctly before you execute the PREPARE or `EXECUTE IMMEDIATE` statement.

`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 10–1 Choosing the Right Method



Using Method 1

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

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

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 };
```

In the following example, you use the host variable *sql_stmt* to store SQL statements input by the user:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 sql_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
...
LOOP
 display 'Enter SQL statement: ';
 read sql_stmt;
 IF sql_stmt is empty THEN
 exit loop;
 ENDIF;
 -- sql_stmt now contains the text of a SQL statement
 EXEC SQL EXECUTE IMMEDIATE :sql_stmt;
ENDLOOP;
```

You can also use string literals, as the following example shows:

```
EXEC SQL EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER';
```

Because EXECUTE IMMEDIATE parses the input SQL statement before every execution, Method 1 is best for statements that are executed only once. Data definition statements usually fall into this category.

An Example

The following program prompts the user for a search condition to be used in the `WHERE` clause of an `UPDATE` statement, then executes the statement using Method 1:

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 update_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
 search_cond CHARACTER(40);
EXEC SQL INCLUDE SQLCA;
```

```
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
set update_stmt = 'UPDATE EMP SET COMM = 500 WHERE ';
display 'Enter a search condition for the following statement:';
display update_stmt;
read search_cond;
concatenate update_stmt, search_cond;
EXEC SQL EXECUTE IMMEDIATE :update_stmt;
EXEC SQL COMMIT WORK RELEASE;
exit program;
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
```

Using Method 2

What Method 1 does in one step, Method 2 does in two. The dynamic SQL statement, which cannot be a query, is first PREPAREd (named and parsed), then executed.

With Method 2, the SQL statement can contain placeholders for input host variables and indicator variables. You can PREPARE the SQL statement once, then EXECUTE it repeatedly using different values of the host variables. Also, you need *not* rePREPARE the SQL statement after a COMMIT or ROLLBACK (unless you log off and reconnect).

Note that you can use EXECUTE for nonqueries with Method 4.

The syntax of the PREPARE statement follows:

```
EXEC SQL PREPARE statement_name
 FROM { :host_string | string_literal };
```

PREPARE parses the SQL statement and gives it a name.

The *statement_name* is an identifier used by the precompiler, *not* a host or program variable, and should not be declared in the Declare Section. It simply designates the PREPAREd statement you want to EXECUTE.

The syntax of the EXECUTE statement is

```
EXEC SQL EXECUTE statement_name [USING host_variable_list];
```

where *host_variable_list* stands for the following syntax:

```
:host_variable1[:indicator1] [, host_variable2[: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 placeholder *n*:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 emp_number INTEGER;
 delete_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
```

```
 search_cond CHARACTER(40);
...
set delete_stmt = 'DELETE FROM EMP WHERE EMPNO = :n AND ';
display 'Complete the following statement's search condition:';
display delete_stmt;
read search_cond;
concatenate delete_stmt, search_cond;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
LOOP
 display 'Enter employee number: ';
 read emp_number;
 IF emp_number = 0 THEN
 exit loop;
 EXEC SQL EXECUTE sql_stmt USING :emp_number;
ENDLOOP;
```

With Method 2, you must know the datatypes of input host variables at precompile time. In the last example, *emp_number* was declared as type `INTEGER`. It could also have been declared as type `CHARACTER` or `REAL`, because Oracle supports all these datatype conversions to the `NUMBER` datatype.

The USING Clause

When the SQL statement is `EXECUTEd`, input host variables in the `USING` clause replace corresponding placeholders in the `PREPAREd` dynamic SQL statement.

Every placeholder in the PREPAREd dynamic SQL statement must correspond to a host variable in the `USING` clause. So, if the same placeholder appears two or more times in the `PREPAREd` statement, each appearance must correspond to a host variable in the `USING` clause. If one of the host variables in the `USING` clause is an array, all must be arrays.

The names of the placeholders need not match the names of the host variables. However, the order of the placeholders in the `PREPAREd` dynamic SQL statement must match the order of corresponding host variables in the `USING` clause.

To specify nulls, you can associate indicator variables with host variables in the `USING` clause. For more information, refer to "Using Indicator Variables".

An Example

The following program prompts the user for a search condition to be used in the `WHERE` clause of an `UPDATE` statement, then prepares and executes the statement using Method 2. Notice that the `SET` clause of the `UPDATE` statement contains a placeholder (*c*).

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 sql_stmt CHARACTER(80);
 empno INTEGER VALUE 1234;
 deptno1 INTEGER VALUE 97;
 deptno2 INTEGER VALUE 99;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC ORACLE OPTION (ORACA=YES);
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
display 'Username? ';
read username;
display 'Password? ';
```

```
read password;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
set sql_stmt =
 'INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:v1, :v2)';
display "V1 = ", empno, "V2 = ", deptno1;
EXEC SQL PREPARE S FROM :sql_stmt;
EXEC SQL EXECUTE S USING :empno, :deptno1;
set empno = empno + 1;
display "V1 = ", empno, "V2 = ", deptno2;
EXEC SQL EXECUTE S USING :empno, :deptno2;
set sql_stmt =
 'DELETE FROM EMP WHERE DEPTNO = :v1 OR DEPTNO = :v2")';
display "V1 = ", deptno1, "V2 = ", deptno2;
EXEC SQL PREPARE S FROM :sql_stmt;
EXEC SQL EXECUTE S USING :deptno1, :deptno2;
EXEC SQL COMMIT WORK RELEASE;
exit program;
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 display 'Processing error';
 EXEC SQL ROLLBACK WORK RELEASE;
 exit program with an error;
```

Using Method 3

Method 3 is similar to Method 2 but combines the `PREPARE` statement with the statements needed to define and manipulate a cursor. This allows your program to accept and process queries. In fact, if the dynamic SQL statement is a query, you *must* use Method 3 or 4.

For Method 3, the number of columns in the query select list and the number of placeholders for input host variables must be known at 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:

```
PREPARE statement_name FROM { :host_string | string_literal };
DECLARE cursor_name CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list];
FETCH cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

Now let us look at what each statement does.

PREPARE

`PREPARE` parses the dynamic SQL statement and gives it a name. In the following example, `PREPARE` parses the query stored in the character string *select_stmt* and gives it the name *sql_stmt*:

```
set select_stmt = 'SELECT MGR, JOB FROM EMP WHERE SAL < :salary';
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

Commonly, the query `WHERE` clause is input from a terminal at run time or is generated by the application.

The identifier *sql_stmt* is *not* a host or program variable, but must be unique. It designates a particular dynamic SQL statement.

DECLARE

DECLARE defines a cursor by giving it a name and associating it with a specific query. The cursor declaration is local to its precompilation unit. Continuing our example, DECLARE defines a cursor named *emp_cursor* and associates it with *sql_stmt*, as follows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

The identifiers *sql_stmt* and *emp_cursor* are *not* host or program variables, but must be unique. If you declare two cursors using the same statement name, the precompiler considers the two cursor names synonymous. For example, if you execute the statements

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
EXEC SQL DECLARE dept_cursor FOR sql_stmt;
```

when you OPEN *emp_cursor*, you will process the dynamic SQL statement stored in *delete_stmt*, not the one stored in *select_stmt*.

OPEN

OPEN allocates an Oracle cursor, binds input host variables, and executes the query, identifying its active set. OPEN also positions the cursor on the first row in the active set and zeroes the rows-processed count kept by the third element of SQLERRD in the SQLCA. Input host variables in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

In our example, OPEN allocates *emp_cursor* and assigns the host variable *salary* to the WHERE clause, as follows:

```
EXEC SQL OPEN emp_cursor USING :salary;
```

FETCH

FETCH returns a row from the active set, assigns column values in the select list to corresponding host variables in the INTO clause, and advances the cursor to the next row. When no more rows are found, FETCH returns the "no data found" Oracle error code to SQLCODE in the SQLCA.

In our example, FETCH returns a row from the active set and assigns the values of columns MGR and JOB to host variables *mgr_number* and *job_title*, as follows:

```
EXEC SQL FETCH emp_cursor INTO :mgr_number, :job_title;
```

CLOSE

CLOSE disables the cursor. After you CLOSE a cursor, you can no longer FETCH from it. In our example, the CLOSE statement disables *emp_cursor*, as follows:

```
EXEC SQL CLOSE emp_cursor;
```

An Example

The following program prompts the user for a search condition to be used in the WHERE clause of a query, then prepares and executes the query using Method 3.

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 dept_number INTEGER;
 emp_name CHARACTER(10);
 salary REAL;
 select_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
 search_cond CHARACTER(40);
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
set select_stmt = 'SELECT ENAME,SAL FROM EMP WHERE ';
display 'Enter a search condition for the following statement:';
display select_stmt;
read search_cond;
concatenate select_stmt, search_cond;
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
display 'Employee Salary';
display '-------- ------';
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 display emp_name, salary;
ENDLOOP;
no_more:
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL COMMIT WORK RELEASE;
 exit program;
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 exit program with an error;
```

Using Method 4

The implementation of Method 4 is very language-dependent. Therefore, this section only gives an overview. For details, see your host-language supplement.

There is a kind of dynamic SQL statement that your program cannot process using Method 3. When the number of select-list items or placeholders for input host variables is unknown until run time, your program must use a descriptor. A *descriptor* is an area of memory used by your program and Oracle to hold a complete description of the variables in a dynamic SQL statement.

Recall that for a multirow query, you FETCH selected column values INTO a list of declared output host variables. If the select list is unknown, the host-variable list

cannot be established at precompile time by the INTO clause. For example, you know the following query returns two column values:

```
SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept_number;
```

However, if you let the user define the select list, you might not know how many column values the query will return.

Need for the SQLDA

To process this kind of dynamic query, your program must issue the DESCRIBE SELECT LIST command and declare a data structure called the SQL Descriptor Area (SQLDA). Because it holds descriptions of columns in the query select list, this structure is also called a *select descriptor*.

Likewise, if a dynamic SQL statement contains an unknown number of placeholders for input host variables, the host-variable list cannot be established at 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 placeholders for the input host variables. (Input host variables are also called *bind variables*.)

If your program has more than one active SQL statement (it might have OPENed two or more cursors, for example), each statement must have its own SQLDA(s). However, non-concurrent cursors can reuse SQLDAs. There is no set limit on the number of SQLDAs in a program.

The DESCRIBE Statement

DESCRIBE initializes a descriptor to hold descriptions of select-list items or input host variables.

If you supply a select descriptor, the DESCRIBE SELECT LIST statement examines each select-list item in a PREPAREd dynamic query to determine its name, datatype, constraints, length, scale, and precision. It then stores this information in the select descriptor.

If you supply a bind descriptor, the DESCRIBE BIND VARIABLES statement examines each placeholder in a PREPAREd dynamic SQL statement to determine its name, length, and the datatype of its associated input host variable. It then stores this information in the bind descriptor for your use. For example, you might use placeholder names to prompt the user for the values of input host variables.

What Is a SQLDA?

A SQLDA is a host-program data structure that holds descriptions of select-list items or input host variables.

SQLDA variables are *not* defined in the Declare Section.

Though SQLDAs differ among host languages, a generic select SQLDA contains the following information about a query select list:

- Maximum number of columns that can be described
- Actual number of columns found by describe
- Addresses of buffers to store column values

- Lengths of column values
- Datatypes of column values
- addresses of indicator-variable values
- Addresses of buffers to store column names
- Sizes of buffers to store column names
- Current lengths of column names

A generic bind SQLDA contains the following information about the input host variables in a SQL statement:

- Maximum number of placeholders that can be described
- Actual number of placeholders found by describe
- Addresses of input host variables
- Lengths of input host variables
- Datatypes of input host variables
- Addresses of indicator variables
- Addresses of buffers to store placeholder names
- Sizes of buffers to store placeholder names
- Current lengths of placeholder names
- Addresses of buffers to store indicator-variable names
- Sizes of buffers to store indicator-variable names
- Current lengths of indicator-variable names

To see the SQLDA structure and variable names in a particular host language, refer to your host-language supplement.

Implementing Method 4

With Method 4, you generally use the following sequence of embedded SQL statements:

```
EXEC SQL PREPARE statement_name
 FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
 INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
 [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
 INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
 USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

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 placeholders for input host variables is unknown, you can use the Method 4 `OPEN` statement with the following Method 3 `FETCH` statement:

```
EXEC SQL FETCH emp_cursor INTO host_variable_list;
```

Conversely, if the number of placeholders for input host variables is known, but the number of columns in the select list is unknown, you can use the following Method 3 OPEN statement with the Method 4 FETCH statement:

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

Note that EXECUTE can be used for nonqueries with Method 4.

To learn how these statements allow your program to process dynamic SQL statements using descriptors, see your host-language supplement.

Using the DECLARE STATEMENT Statement

With Methods 2, 3, and 4, you might need to use the statement

```
EXEC SQL [AT db_name] DECLARE statement_name STATEMENT;
```

where *db_name* and *statement_name* are identifiers used by the precompiler, *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 nondefault database. An example using Method 2 follows:

```
EXEC SQL AT remote_db DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL EXECUTE sql_stmt;
```

In the example, *remote_db* tells Oracle 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 sql_stmt STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
```

The usual sequence of statements is

```
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

Usage of Host Arrays

Usage of host arrays in static and dynamic SQL is similar. For example, to use input host arrays with dynamic SQL Method 2, use the syntax

```
EXEC SQL EXECUTE statement_name USING host_array_list;
```

where *host_array_list* contains one or more host arrays. With Method 3, use the following syntax:

```
OPEN cursor_name USING host_array_list;
```

To use output host arrays with Method 3, use the following syntax:

```
FETCH cursor_name INTO host_array_list;
```

With Method 4, you must use the optional FOR clause to tell Oracle the size of your input or output host array. To learn how this is done, see your host-language supplement.

Using PL/SQL

The Oracle Precompilers treat a PL/SQL block like a single SQL statement. So, like a SQL statement, a PL/SQL block can be stored in a string host variable or literal. When you store the PL/SQL block in the string, omit the keywords EXEC SQL `EXECUTE`, the keyword END-EXEC, and the statement terminator.

However, there are two differences in the way the precompiler handles SQL and PL/SQL:

- The precompiler treats all PL/SQL host variables as *input* host variables whether they serve as input or output host variables (or both) inside the PL/SQL block.
- You cannot FETCH from a PL/SQL block because it might contain any number of SQL statements.

With Method 1

If the PL/SQL block contains no host variables, you can use Method 1 to `EXECUTE` the PL/SQL string in the usual way.

With Method 2

If the PL/SQL block contains a known number of input and output host variables, you can use Method 2 to `PREPARE` and `EXECUTE` the PL/SQL string in the usual way.

You must put *all* host variables in the USING clause. When the PL/SQL string is `EXECUTEd`, host variables in the USING clause replace corresponding placeholders in the `PREPAREd` string. Though the precompiler treats all PL/SQL host variables as input host variables, values are assigned correctly. Input (program) values are assigned to input host variables, and output (column) values are assigned to output host variables.

Every placeholder in the `PREPAREd` PL/SQL string must correspond to a host variable in the `USING` clause. So, if the same placeholder appears two or more times in the `PREPAREd` string, each appearance must correspond to a host variable in the USING clause.

With Method 3

Methods 2 and 3 are the same except that Method 3 allows `FETCHing`. Since you cannot `FETCH` from a PL/SQL block, 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 the precompiler treats all PL/SQL host variables as input host variables, executing `DESCRIBE SELECT LIST` has no effect.

The use of bind descriptors with Method 4 is detailed in your host-language supplement.

> **Note:** 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 (/* ... */).

11

Writing User Exits

This chapter contains the following:

- What Is a User Exit?
- Why Write a User Exit?
- Developing a User Exit
- Writing a User Exit
- Calling a User Exit
- Passing Parameters to a User Exit
- Returning Values to a Form
- An Example
- Precompiling and Compiling a User Exit
- Using the GENXTB Utility
- Linking a User Exit into SQL*Forms
- Guidelines for SQL*Forms User Exits
- EXEC TOOLS Statements

This chapter focuses on writing user exits for your SQL*Forms and Oracle Forms applications. First, you learn the EXEC IAF statements that allow a SQL*Forms application to interface with user exits. Then, you learn how to write and link a SQL*Forms user exit. You also learn how to use EXEC TOOLS statements with Oracle Forms. (SQL*Forms does not support EXEC TOOLS.) That way, you can use EXEC IAF statements to enhance your existing applications and EXEC TOOLS statements to build new applications. The following topics are covered:

- Common uses for user exits
- Writing a user exit
- Passing values between SQL*Forms and a user exit
- Implementing a user exit
- Calling a user exit
- Guidelines for SQL*Forms user exits
- Using EXEC TOOLS statements with Oracle Forms

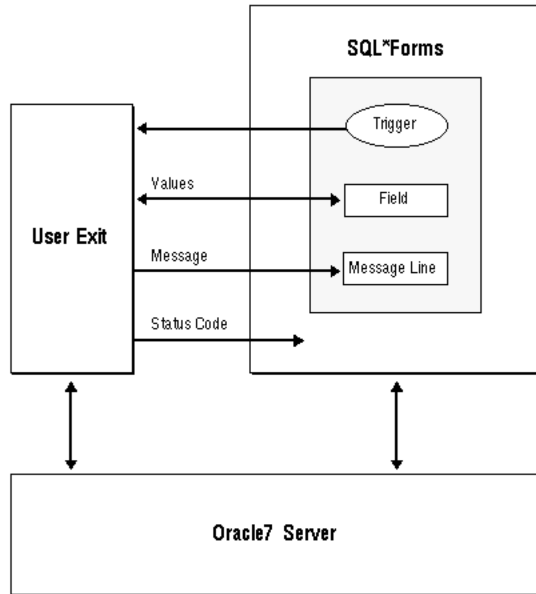
This chapter is supplemental. For more information about user exits, see the *SQL*Forms Designer's Reference*, the *Oracle Forms Reference Manual, Vol. 2*, and your system-specific Oracle manuals.

What Is a User Exit?

A *user exit* is a host-language subroutine written by you and called by SQL*Forms to do special-purpose processing. You can embed SQL commands and PL/SQL blocks in your user exit, then precompile it as you would a host program.

When called by a SQL*Forms trigger, the user exit runs, then returns a status code to SQL*Forms (refer to Figure 11–1). Your user exit can display messages on the SQL*Forms status line, get and put field values, manipulate Oracle data, do high-speed computations and table lookups -- even log on to different databases.

Figure 11–1 SQL*Forms



Why Write a User Exit?

SQL*Forms Version 3 enables use PL/SQL blocks in triggers. So, in most cases, instead of calling a user exit, you can use the procedural power of PL/SQL. If the need arises, you can call user exits from a PL/SQL block with the `USER_EXIT` function.

User exits are harder to write and implement than SQL, PL/SQL, or SQL*Forms commands. So, you will probably use them only to do processing that is beyond the scope of SQL, PL/SQL, and SQL*Forms. Some common uses follow:

- Operations more quickly or easily performed in third generation languages like C and FORTRAN (for example, numeric integration)
- Controlling real time devices or processes (for example, issuing a sequence of instructions to a printer or graphics device)
- Data manipulations that need extended procedural capabilities (for example, recursive sorting)
- Special file I/O operations

Developing a User Exit

This section outlines the way to develop a SQL*Forms user exit; later sections go into more detail. For information about EXEC TOOLS statements, which are available with

Oracle Forms, see EXEC TOOLS Statements"

To incorporate a user exit into a form, you take the following steps:

1. Write the user exit in a supported host language.

2. Precompile the source code.

3. Compile the modified source code.

4. Use the GENXTB utility to create a database table, IAPXTB.

5. Use the GENXTB form in SQL*Forms to insert your user exit information into the database table.

6. Use the GENXTB utility to read the information from the table and create an IAPXIT source module. Then, compile the source module.

7. Create a new IAP (the SQL*Forms component that runs a form) by linking the standard IAP object modules, your user exit object module, and the IAPXIT object module created in step 6.

8. In the form, define a trigger to call the user exit.

9. Instruct operators to use the new IAP when running the form. This is unnecessary if the new IAP replaces the standard one. For details, refer to your system-specific Oracle manuals.

Writing a User Exit

You can use the following kinds of statements to write your SQL*Forms user exit:

- host-language
- EXEC SQL
- EXEC ORACLE
- EXEC IAF GET
- EXEC IAF PUT

This section focuses on the EXEC IAF `GET` and `PUT` statements, which let you pass values between SQL*Forms and a user exit.

Requirements for Variables

The variables used in EXEC IAF statements must correspond to field names used in the form definition. If a field reference is ambiguous because you did not specify a block name, you get an error. An invalid or ambiguous reference to a form field generates an error.

Host variables must be named in the user exit Declare Section and must be prefixed with a colon (:) in EXEC IAF statements.

> **Note:** : Indicator variables are *not* allowed in EXEC IAF `GET` and `PUT` statements.

The IAF GET Statement

This statement allows your user exit to "get" values from fields on a form and assign them to host variables. The user exit can then use the values in calculations, data manipulations, updates, and so on. The syntax of the `GET` statement follows:

```
EXEC IAF GET field_name1, field_name2, ...
 INTO :host_variable1, :host_variable2, ...;
```

where *field_name* can be any of the following SQL*Forms variables:

- field
- block.field
- system variable
- global variable
- host variable (prefixed with a colon) containing the value of a field, block.field, system variable, or global variable

If *field_name* is not qualified, it must be unique.

The following example shows how a user exit GETs a field value and assigns it to a host variable:

```
EXEC IAF GET employee.job INTO :new_job;
```

All field values are character strings. If it can, `GET` converts a field value to the datatype of the corresponding host variable. If an illegal or unsupported datatype conversion is attempted, an error is generated.

In the last example, a constant is used to specify *block.field*. You can also use a host string to specify block and field names, as follows:

```
set blkfld = 'employee.job';
EXEC IAF GET :blkfld INTO :new_job;
```

Unless the field is unique, the host string must contain the full *block.field* reference with intervening period. For example, the following usage is *invalid*:

```
set blk = 'employee';
set fld = 'job';
EXEC IAF GET :blk.:fld INTO :new_job;
```

You can mix explicit and stored field names in a `GET` statement field list, but not in a single field reference. For example, the following usage is *invalid*:

```
set fld = 'job';
EXEC IAF GET employee.:fld INTO :new_job;
```

The IAF PUT Statement

This statement allows your user exit to put the values of constants and host variables into fields on a form. Thus, the user exit can display on the SQL*Forms screen any value or message you like. The syntax of the `PUT` statement follows:

```
EXEC IAF PUT field_name1, field_name2, ...
 VALUES (:host_variable1, :host_variable2, ...);
```

where *field_name* can be any of the following SQL*Forms variables:

- field
- block.field
- system variable
- global variable

- host variable (prefixed with a colon) containing the value of a field, block.field, system variable, or global variable

The following example shows how a user exit PUTs the values of a numeric constant, string constant, and host variable into fields on a form:

```
EXEC IAF PUT employee.number, employee.name, employee.job
 VALUES (7934, 'MILLER', :new_job);
```

Like `GET`, `PUT` lets you use a host string to specify block and field names, as follows:

```
set blkfld = 'employee.job';
EXEC IAF PUT :blkfld VALUES (:new_job);
```

On character-mode terminals, a value `PUT` into a field is displayed when the user exit returns, rather than when the assignment is made, provided the field is on the current display page. On block-mode terminals, the value is displayed the next time a field is read from the device.

If a user exit changes the value of a field several times, only the last change takes effect.

Calling a User Exit

You call a user exit from a SQL*Forms trigger using a packaged procedure named `USER_EXIT` (supplied with SQL*Forms). The syntax you use is

```
USER_EXIT(user_exit_string [, error_string]);
```

where *user_exit_string* contains the name of the user exit plus optional parameters and *error_string* contains an error message issued by SQL*Forms if the user exit fails. For example, the following trigger command calls a user exit named `LOOKUP`:

```
USER_EXIT('LOOKUP');
```

Notice that the user exit string is enclosed by single (not double) quotes.

Passing Parameters to a User Exit

When you call a user exit, SQL*Forms passes it the following parameters automatically:

Command Line is the user exit string.

Command Line Length is the length (in characters) of the user exit string.

Error Message is the error string (failure message) if one is defined.

Error Message Length is the length of the error string.

In-Query is a Boolean value indicating whether the exit was called in normal or query mode.

However, the user exit string enables pass additional parameters to the user exit. For example, the following trigger command passes two parameters and an error message to the user exit `LOOKUP`:

```
USER_EXIT('LOOKUP 2025 A', 'Lookup failed');
```

You can use this feature to pass field names to the user exit, as the following example shows:

```
USER_EXIT('CONCAT firstname, lastname, address');
```

However, it is up to the user exit, not SQL*Forms, to parse the user exit string.

Returning Values to a Form

When a user exit returns control to SQL*Forms, it must also return a code indicating whether it succeeded, failed, or suffered an irrecoverable error. The return code is an integer constant generated by precompiler (refer to this section: An Example). The three results have the following meanings:

Success: The user exit encountered no errors. SQL*Forms proceeds to the *success* label or the next step, unless the Reverse Return Code switch is set by the calling trigger step.

Failure: The user exit detected an error, such as an invalid value in a field. An optional message passed by the exit appears on the message line at the bottom of the SQL*Forms screen and on the Display Error screen. SQL*Forms responds as it does to a SQL statement that affects no rows.

Fatal error: The user exit detected a condition that makes further processing impossible, such as an execution error in a SQL statement. An optional error message passed by the exit appears on the SQL*Forms Display Error screen. SQL*Forms responds as it does to an irrecoverable SQL error.

If a user exit changes the value of a field, then returns a *failure* or *fatal error* code, SQL*Forms does *not* discard the change. Nor does SQL*Forms discard changes when the Reverse Return Code switch is set and a *success* code is returned.

The IAP Constants

The precompiler generates three symbolic constants for use as return codes. They are prefixed with IAP. For example, the three constants might be IAPSUCC, IAPFAIL, and IAPFTL.

Using the SQLIEM Function

By calling the function SQLIEM, your user exit can specify an error message that SQL*Forms will display on the message line if the trigger step fails or on the Display Error screen if the step causes an irrecoverable error. The specified message replaces any message defined for the step.

The syntax of the SQLIEM function call is

```
SQLIEM (error_message, message_length);
```

where *error_message* and *message_length* are character and integer variables, respectively. The Oracle Precompilers generate the appropriate external function declaration for you. You pass both parameters by reference; that is, you pass their addresses, not their values. SQLIEM is a SQL*Forms function; it cannot be called from other Oracle tools.

Using WHENEVER

You can use the `WHENEVER` statement in an exit to detect invalid datatype conversions (SQLERROR), truncated values `PUT` into form fields (SQLWARNING), and queries that return no rows (`NOT FOUND`).

An Example

The following example shows how a typical user exit is coded. Notice that, like a host program, the user exit has a Declare Section and a SQLCA.

```
-- subroutine MYEXIT
EXEC SQL BEGIN DECLARE SECTION;
 field1 CHARACTER(20);
 field2 CHARACTER(20);
 value1 CHARACTER(20);
 value2 CHARACTER(20);
 result_val CHARACTER(20);
EXEC SQL END DECLARE SECTION;
 errmsg CHARACTER(80);
 errlen INTEGER;
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
-- get field values from form
EXEC IAF GET :field1, :field2 INTO :value1, :value2;
-- manipulate values to obtain result_val
-- put result_val into form field
EXEC IAF PUT result VALUES (:result_val);
return(IAPSUCC); -- trigger step succeeded
sqlerror:
 set errmsg = CONCAT('MYEXIT: ', sqlca.sqlerrm.sqlerrmc);
 set errlen = LENGTH(errmsg);
 sqliem(errmsg, errlen); -- pass error message to SQL*Forms
 return(IAPFAIL); -- trigger step failed
```

For a complete host-language example, see your host -language supplement.

Precompiling and Compiling a User Exit

User exits are precompiled like standalone host programs. Refer to Chapter 6, "Running the Oracle Precompilers"

For instructions on compiling a user exit, see your system-specific Oracle manuals.

Using the GENXTB Utility

The IAP program table IAPXTB in module IAPXIT contains an entry for each user exit linked into IAP. IAPXTB tells IAP the name, location, and host language of each user exit. When you add a new user exit to IAP, you must add a corresponding entry to IAPXTB.

IAPXTB is derived from a database table, also named IAPXTB. You can modify the database table by running the GENXTB form on the operating system command line, as follows:

```
RUNFORM GENXTB username/password
```

A form is displayed that enables you to enter the following information for each user exit you define:

- exit name
- host-language code (COB, FOR, PAS, or PLI)
- date created
- date last modified

- comments

After modifying the IAPXTB database table, use the GENXTB utility to read the table and create an Assembler or C source program that defines the module IAPXIT and the IAPXTB program table it contains. The source language used depends on your operating system. The syntax you use to run the GENXTB utility is

```
GENXTB username/password outfile
```

where *outfile* is the name you give the Assembler or source program that GENXTB creates.

Linking a User Exit into SQL*Forms

Before running a form that calls a user exit, you must link the user exit into IAP. The user exit can be linked into your standard version of IAP or into a special version for those forms that call the exit.

To produce a new executable copy of IAP, link your user exit object module, the standard IAP modules, the IAPXIT module, and any modules needed from the Oracle and host-language link libraries. The details of linking are system-dependent, so check your system-specific Oracle manuals.

Guidelines for SQL*Forms User Exits

The guidelines in this section will help you avoid some common pitfalls.

Naming the Exit

The name of your user exit cannot be an Oracle reserved word. Also avoid using names that conflict with the names of SQL*Forms commands, function codes, and externally defined names used by SQL*Forms.

SQL*Forms converts the name of a user exit to upper case before searching for the exit. Therefore, the exit name must be in upper case in your source code if your host language is case-sensitive.

The name of the user exit entry point in the source code becomes the name of the user exit itself. The exit name must be a valid file name for your host language and operating system.

Connecting to Oracle

User exits communicate with Oracle through the connection made by SQL*Forms. However, a user exit can establish additional connections to any database through SQL*Net. For more information, refer to Concurrent Logons".

Issuing I/O Calls

SQL*Forms I/O routines might conflict with host-language printer I/O routines. If they do, your user exit will be unable to issue printer I/O calls. File I/O is supported but screen I/O is not.

Using Host Variables

Restrictions on the use of host variables in a standalone program also apply to user exits. Host variables must be named in the user exit Declare Section and must be

prefixed with a colon in EXEC SQL and EXEC IAF statements. However, the use of host arrays is not allowed in EXEC IAF statements.

Updating Tables

Generally, a user exit should not UPDATE database tables associated with a form. For example, suppose an operator updates a record in the SQL*Forms work space, then a user exit UPDATEs the corresponding row in the associated database table. When the transaction is COMMITted, the record in the SQL*Forms work space is applied to the table, overwriting the user exit UPDATE.

Issuing Commands

Avoid issuing a COMMIT or ROLLBACK command from your user exit because Oracle will commit or roll back work begun by the SQL*Forms operator, not just work done by the user exit. Instead, issue the COMMIT or ROLLBACK from the SQL*Forms trigger. This also applies to data definition commands (such as ALTER and CREATE) because they issue an implicit COMMIT before and after executing.

EXEC TOOLS Statements

EXEC TOOLS statements support the basic Oracle Toolset (Oracle Forms, Oracle Reports, and Oracle Graphics) by providing a generic way to handle get, set, and exception callbacks from user exits. The following discussion focuses on Oracle Forms but the same concepts apply to Oracle Reports and Oracle Graphics.

Besides EXEC SQL, EXEC ORACLE, and host language statements, you can use the following EXEC TOOLS statements to write an Oracle Forms user exit:

- SET
- GET
- SET CONTEXT
- GET CONTEXT
- MESSAGE

The EXEC TOOLS GET and SET statements replace the EXEC IAF GET and PUT statements used with SQL*Forms. Unlike IAF GET and PUT, TOOLS GET and SET accept indicator variables. The EXEC TOOLS MESSAGE statement replaces the message-handling function SQLIEM. The EXEC TOOLS SET CONTEXT and GET CONTEXT statements are new and not available with SQL*Forms, Version 3.

> **Note:** COBOL and FORTRAN do not have a pointer datatype, so you cannot use the SET CONTEXT and GET CONTEXT statements in a Pro*COBOL or Pro*FORTRAN program.

EXEC TOOLS SET

The EXEC TOOLS SET statement passes values from your user exit to Oracle Forms. Specifically, it assigns the values of host variables and constants to Oracle Forms variables and items. The values are displayed after the user exit returns control to the form.

To code the EXEC TOOLS SET statement, you use the syntax

```
EXEC TOOLS SET form_variable[, ...]
 VALUES ({:host_variable[:indicator] | constant}[, ...]);
```

where *form_variable* is an Oracle Forms field, parameter, system variable, or global variable, or a host variable (prefixed with a colon) containing the name of one of the foregoing items.

In the following Pro*C example, your user exit passes an employee name (with optional indicator) to Oracle Forms:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 char ename[20];
 short ename_ind;
EXEC SQL END DECLARE SECTION;
...
strcpy(ename, "MILLER");
ename_ind = 0;
EXEC TOOLS SET emp.ename VALUES (:ename:ename_ind);
```

In this example, *emp.ename* is an Oracle Forms block.field.

EXEC TOOLS GET

The EXEC TOOLS GET statement passes values from Oracle Forms to your user exit. Specifically, it assigns the values of Oracle Forms variables and items to host variables. As soon as the values are passed, the user exit can use them for any purpose.

To code the EXEC TOOLS GET statement, you use the syntax

```
EXEC TOOLS GET form_variable[, ...]
 INTO :host_variable[:indicator][, ...];
```

where *form_variable* is an Oracle Forms field, parameter, system variable, or global variable, or a host variable containing the name of one of the foregoing items.

In the following example, Oracle Forms passes an employee name from the block.field *emp.ename* to your user exit:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 char ename[20];
EXEC SQL END DECLARE SECTION;
...
EXEC TOOLS GET emp.ename INTO :ename;
```

EXEC TOOLS SET CONTEXT

The EXEC TOOLS SET CONTEXT statement lets you save context information from one user exit call to another. SET CONTEXT names a host-language pointer variable that you can reference later in an EXEC TOOLS GET CONTEXT statement. The pointer variable points to the block of memory in which the context information is stored. With the SET CONTEXT statement, you need not declare a global variable to hold the information.

To code the EXEC TOOLS SET CONTEXT statement, use the syntax

```
EXEC TOOLS SET CONTEXT :host_pointer_variable
 [IDENTIFIED] BY context_name;
```

where the optional keyword IDENTIFIED can be used to improve readability and *context_name* is an undeclared identifier or a character host variable that names the context area.

In the following example, your user exit saves context information for later use:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 char context1[30];
EXEC SQL END DECLARE SECTION;
...
strcpy(context1, "This is context1");
EXEC TOOLS SET CONTEXT :context1 BY my_app1;
```

In this example, the context name *my_app1* is an undeclared identifier. Note that in C, when a char array is used as an argument, the array name is synonymous with a pointer to that array.

EXEC TOOLS GET CONTEXT

The EXEC TOOLS GET CONTEXT statement retrieves the value of a host-language pointer variable into your user exit. The pointer variable points to a block of memory in which context information is stored.

To code the EXEC TOOLS GET CONTEXT statement, use the syntax

```
EXEC TOOLS GET CONTEXT context_name INTO :host_pointer_variable;
```

where *context_name* is an undeclared identifier or a character host variable that names the context area.

In the following Pro*C example, your user exit retrieves a pointer to context information saved earlier:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 char *ctx_ptr;
EXEC SQL END DECLARE SECTION;
...
EXEC TOOLS GET CONTEXT my_app1 INTO :ctx_ptr;
```

In this example, the context name *my_app1* is an undeclared identifier.

EXEC TOOLS MESSAGE

The EXEC TOOLS MESSAGE statement passes a message from your user exit to Oracle Forms. The message is displayed on the Oracle Forms message line after the user exit returns control to the form.

To code the EXEC TOOLS MESSAGE statement, you use the syntax

```
EXEC TOOLS MESSAGE message_text [severity_code];
```

where *message_text* is a quoted string or a character host variable, and the optional *severity_code* is an integer constant or host variable. The MESSAGE statement does *not* accept indicator variables.

In the following Pro*C example, your user exit passes an error message and severity code to Oracle Forms:

```
EXEC TOOLS MESSAGE 'Bad field name! Please reenter.' 15;
```

A

New Features

This appendix looks at the improvements and new features offered by the Oracle Precompilers Release 1.8. Designed to meet the practical needs of professional software developers, these features will help you build effective, reliable applications.

A.1 Fetching NULLs without Using Indicator Variables

With releases 1.5, 1.6, and 1.7 of the Oracle Precompilers, source files that `FETCH` data into host variables without associated indicator variables return an `ORA-01405` message at run time if a `NULL` is returned to the host variable. With release 1.8, when you specify `MODE=ORACLE` and `DBMS=V7`, you can disable the `ORA-01405` message by also specifying `UNSAFE_NULL=YES`.

When developing applications for the Oracle Database, the preferred practice is to include indicator variables for any host variable that might have a `NULL` returned to it. When migrating applications from Oracle Version 6 to Oracle database version 7, however, the `UNSAFE_NULL` option can significantly ease the process.

For more information, see "UNSAFE_NULL" and "Using Indicator Variables".

A.1.1 Using DBMS=V7 and MODE=ORACLE

Applications precompiled with `MODE=ORACLE` and `DBMS=V7` return the `ORA-01405` error at run time if a `NULL` is returned to a host variable when there is no associated indicator variable. When upgrading to Oracle database version 7 with these options specified, you will need to migrate your applications in one of two ways:

- Modify your source code to include the necessary indicator variables
- Specify `UNSAFE_NULL=YES` on the command line

If you are upgrading to Oracle database version 7 and use `DBMS=V7` when precompiling, or if you intend to use new Oracle database version 7 features that are different from Oracle Version 6, in most instances, the change requires minimal modification to your source files. However, if your application may `FETCH` null values into host variables without associated indicator variables, specify `UNSAFE_NULL=YES` to disable the `ORA-01405` message and avoid adding the relevant indicator variables to your source files.

A.1.2 Related Error Messages

For information about precompile time messages associated with the `UNSAFE_NULL` option, see *Oracle Database Error Messages*.

A.2 Additional Array Insert/Select Syntax

The array INSERT and array SELECT syntax of the DB2 precompiler is now supported by the Oracle precompiler. The optional ROWSET and ROWSET STARTING AT clauses are used in the fetch-orientation (FIRST, PRIOR, NEXT, LAST, CURRENT, RELATIVE and ABSOLUTE). For more information about the new INSERT/SELECT syntax, please refer the *Pro*COBOL Programmer's Guide* and *Pro*C/C++ Programmer's Guide*.

A.3 SQL99 Syntax Support

The SQL standard enables the portability of SQL applications across all conforming software products. Oracle features are compliant with the ANSI/ISO SQL99 standard, including ANSI compliant joins. Pro*Cobol supports all SQL99 features that are supported by Oracle database, which means that the SQL99 syntax for the SELECT, INSERT, DELETE, and UPDATE statements and the body of the cursor in a DECLARE CURSOR statement are supported.

A.4 Fixing Execution Plans

To fix execution plans for SQL's used in Pro*C/C++ or Pro*Cobol development environment, you need to use the outline feature of Oracle at the time of precompiling. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If you enable the use of the outline for the statement, Oracle automatically considers the stored hints and tries to generate an execution plan in accordance with those hints. In this way, you can ensure that the performance is not affected when the modules are integrated or deployed into different environments.

You can use the following SQL statements to create outlines in Pro*C/C++ and Pro*Cobol:

- `SELECT`
- `DELETE`
- `UPDATE`
- `INSERT... SELECT`
- `CREATE TABLE... AS SELECT`

If the outline option is set, then the precompiler generates two files, a SQL file and a LOG file at the end of successful precompilation. Command line options `outline` and `outlnprefix` control the generation of the outlines.

Each generated outline name is unique. Because the file names used in the application are unique, this information is used in generating the outline name. In addition, the category name is also prefixed.

Note: Oracle allows only 30 bytes for the outline name. If you exceed the limit, the precompiler will flag an error. You can restrict the length of the outline name by using the `outlnprefix` option.

See Also:

- *Pro*COBOL Programmer's Guide*.
- *Pro*C/C++ Programmer's Guide*.

A.5 Using Implicit Buffered Insert

For improved performance, application developers can reference host arrays in their embedded SQL statements. This provides a means to execute an array of SQL statements with a single round-trip to the database. Despite the significant performance improvements afforded by array execution, some developers choose not to use this capability because it is not ANSI standard. For example, an application written to exploit array execution in Oracle cannot be precompiled using IBM's precompiler.

One workaround is to use buffered INSERT statements, which enable you to gain performance benefits while retaining ANSI standard embedded SQL syntax.

The command line option "max_row_insert" controls the number of rows to be buffered before executing the INSERT statement. By default it is zero and the feature is disabled. To enable this feature, specify any number greater than zero.

> **See Also:** For more information on using the implicit buffer insert feature, please refer to:
>
> - *Pro*COBOL Programmer's Guide*.
> - *Pro*C/C++ Programmer's Guide*.

A.6 Dynamic SQL Statement Caching

Statement caching refers to the feature that provides and manages a cache of statements for each session. In the server, it means that cursors are ready to be used without the statement being parsed again. Statement caching can be enabled in the precompiler applications, which will help in the performance improvement of all applications that rely on the dynamic SQL statements. Performance improvement is achieved by removing the overhead of parsing the dynamic statements on reuse.

You can obtain this performance improvement by using a new command line option, `stmt_cache` (for the statement cache size), which will enable the statement caching of the dynamic statements. By enabling the new option, the statement cache will be created at session creation time. The caching is only applicable for the dynamic statements and the cursor cache for the static statements co-exists with this feature.

The command line option `stmt_cache` can be given any value in the range of 0 to 65535. Statement caching is disabled by default (value 0). The `stmt_cache` option can be set to hold the anticipated number of distinct dynamic SQL statements in the application.

Example A–1 Using the stmt_cache Option

This example demonstrates the use of the `stmt_cache` option. In this program, you insert rows into a table and select the inserted rows by using the cursor in the loop. When the stmt_cache option is used to precompile this program, the performance increases compared to a normal precompilation.

```
/*
 *  stmtcache.pc
 *
 *  NOTE:
 *  When this program is used to measure the performance with and without
 *  stmt_cache option, do the following changes in the program,
 *  1. Increase ROWSCNT to high value, say 10000.
 *  2. Remove all the print statements, usually which comsumes significant
 *     portion of the total program execution time.
```

```
 *
 *  HINT: In Linux, gettimeofday() can be used to measure time.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include <oraca.h>

#define ROWSCNT 10

char    *username = "scott";
char    *password = "tiger";

/* Function prototypes */
void sql_error(char *msg);
void selectdata();
void insertdata();

int main()
{
  EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error");

  /* Connect using the default schema scott/tiger */
  EXEC SQL CONNECT :username IDENTIFIED BY :password;

  /* core functions to insert and select the data */
  insertdata();
  selectdata();

/* Rollback pll the changes and disconnect from Oracle. */
  EXEC SQL ROLLBACK WORK RELEASE;

  exit(0);
}

/*Insert the data for ROWSCNT items into tpc2sc01 */
void insertdata()
{
  varchar dynstmt[80];
  int i;
  varchar ename[10];
  float comm;
  char *str;

  /* Allocates temporary buffer */
  str = (char *)malloc (11 * sizeof(char));

  strcpy ((char *)dynstmt.arr,
          "INSERT INTO bonus (ename, comm) VALUES (:ename, :comm)");
  dynstmt.len = strlen(dynstmt.arr);
  EXEC SQL PREPARE S FROM :dynstmt;

  printf ("Inserts %d rows into bonus table using dynamic SQL statement\n",
          ROWSCNT);
  for (i=1; i<=ROWSCNT; i++)
  {
    sprintf (str, "EMP_%05d",i);
    strcpy (ename.arr, str);
```

```
    comm = i;
    ename.len = strlen (ename.arr);
    EXEC SQL EXECUTE S USING :ename, :comm;
  }

  free(str);
}

/* Select the data using the cursor */
void selectdata()
{
  varchar dynstmt[80];
  varchar ename[10];
  float comm;
  int i;

  strcpy((char *)dynstmt.arr,
         "SELECT ename, comm FROM bonus WHERE comm = :v1");
  dynstmt.len = (unsigned short)strlen((char *)dynstmt.arr);

  printf ("Fetches the inserted rows using using dynamic SQL statement\n\n");
  printf ("  ENAME      COMMISSION\n\n");

  for (i=1; i<=ROWSCNT; i++)
  {
    /* Do the prepare in the loop so that the advantage of stmt_caching
       is visible*/
    EXEC SQL PREPARE S FROM :dynstmt;

    EXEC SQL DECLARE C CURSOR FOR S;
    EXEC SQL OPEN C USING :i;

    EXEC SQL WHENEVER NOT FOUND DO break;

    /* Loop until the NOT FOUND condition is detected. */
    for (;;)
    {
      EXEC SQL FETCH C INTO :ename, :comm;
      ename.arr[ename.len] = '\0';
      printf ("%10s    %7.2f\n", ename.arr, comm);
    }
    /* Close the cursor so that the reparsing is not required for stmt_cache */
    EXEC SQL CLOSE C;
  }
}

void sql_error(char *msg)
{
    printf("\n%s", msg);
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
    oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = '\0';
    oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = '\0';
    printf("\n%s\n", sqlca.sqlerrm.sqlerrmc);
    printf("in \"%s...\"\n", oraca.orastxt.orastxtc);
    printf("on line %d of %s.\n\n", oraca.oraslnr,
       oraca.orasfnm.orasfnmc);

   /* Disable ORACLE error checking to avoid an infinite loop
    * should another error occur within this routine.
    */
```

```
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /* Release resources associated with the cursor. */
    EXEC SQL CLOSE C;

    /* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

A.7 Scrollable Cursors

A scrollable cursor is a work area where Oracle executes SQL statements and stores information that is processed during execution.

When a cursor is executed, the results of the query are placed into a a set of rows called the result set. The result set can be fetched either sequentially or non-sequentially. Non-sequential result sets are called scrollable cursors.

A scrollable cursor enables users to access the rows of a database result set in a forward, backward, and random manner. This scrollable cursor enables the program to fetch any row in the result set. For more information about scrollable cursors, please refer the *Pro*COBOL Programmer's Guide*.

A.8 Platform Endianness Support

Oracle stored unicode data (UTF16) is always in big-endian form. Currently, client applications run on different platforms. Linux and Windows have little-endian representation and Solaris has big-endian representation. When UTF16 data is inserted or selected, Pro*Cobol doesn't convert endian form between server and the client. This leads to corrupted UTF16 (UCS2) strings in the PIC N variable.

Platform endianness (Little-endian form for Linux and Windows, Big-endian form for Solaris) in PIC N variables can be maintained using the command line option `picn_endian`.

New Command Line Option

```
picn_endian={BIG|OS}
```

If picn_endian=big, then PIC N variables are bound with character set ID AL16UTF16.

If picn_endian=os then PIC N variables are bound with character set ID UCS2.

The default value for this option is "big" to preserve the current behavior. This option is ignored if NLS_NCHAR is not AL16UTF16.

Character set form for PIC N variables can be set by using the existing Pro*Cobol command line option:

```
charset_picn={nchar_charset|db_charset}
```

A.9 Flexible B Area Length

The length of B Area for a Pro*Cobol program is limited to 72 when the format is set to ANSI. Cobol compilers now can support B Area length up to 253. This provides a programmer with the flexibility to type a line that is longer than 72 columns. Pro*Cobol now supports B area length up to 253 when a Pro*Cobol application is precompiled with the

```
FORMAT=VARIABLE
```

option.

B

Oracle Reserved Words, Keywords, and Namespaces

This appendix contains the following sections:

- Oracle Reserved Words
- Oracle Keywords
- PL/SQL Reserved Words
- Oracle Reserved Namespaces

This appendix lists words that have a special meaning to Oracle. Each word plays a specific role in the context in which it appears. For example, in an `INSERT` statement, the reserved word `INTO` introduces the tables to which rows will be added. But, in a `FETCH` or `SELECT` statement, the reserved word `INTO` introduces the output host variables to which column values will be assigned.

B.1 Oracle Reserved Words

The following words are reserved by Oracle. That is, they have a special meaning to Oracle and so cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes.

Oracle Reserved Words			
ACCESS	ELSE	MODIFY	START
ADD	EXCLUSIVE	NOAUDIT	SELECT
ALL	EXISTS	NOCOMPRESS	SESSION
ALTER	FILE	NOT	SET
AND	FLOAT	NOTFOUND	SHARE
ANY	FOR	NOWAIT	SIZE
ARRAYLEN	FROM	NULL	SMALLINT
AS	GRANT	NUMBER	SQLBUF
ASC	GROUP	OF	SUCCESSFUL
AUDIT	HAVING	OFFLINE	SYNONYM
BETWEEN	IDENTIFIED	ON	SYSDATE
BY	IMMEDIATE	ONLINE	TABLE
CHAR	IN	OPTION	THEN

Oracle Reserved Words			
CHECK	INCREMENT	OR	TO
CLUSTER	INDEX	ORDER	TRIGGER
COLUMN	INITIAL	PCTFREE	UID
COMMENT	INSERT	PRIOR	UNION
COMPRESS	INTEGER	PRIVILEGES	UNIQUE
CONNECT	INTERSECT	PUBLIC	UPDATE
CREATE	INTO	RAW	USER
CURRENT	IS	RENAME	VALIDATE
DATE	LEVEL	RESOURCE	VALUES
DECIMAL	LIKE	REVOKE	VARCHAR
DEFAULT	LOCK	ROW	VARCHAR2
DELETE	LONG	ROWID	VIEW
DESC	MAXEXTENTS	ROWLABEL	WHENEVER
DISTINCT	MINUS	ROWNUM	WHERE
DROP	MODE	ROWS	WITH

B.2 Oracle Keywords

The following words also have a special meaning to Oracle but are not reserved words and so can be redefined. However, some might eventually become reserved words.

Oracle Keywords			
ADMIN	CURSOR	FOUND	MOUNT
AFTER	CYCLE	FUNCTION	NEXT
ALLOCATE	DATABASE	GO	NEW
ANALYZE	DATAFILE	GOTO	NOARCHIVELOG
ARCHIVE	DBA	GROUPS	NOCACHE
ARCHIVELOG	DEC	INCLUDING	NOCYCLE
AUTHORIZATION	DECLARE	INDICATOR	NOMAXVALUE
AVG	DISABLE	INITRANS	NOMINVALUE
BACKUP	DISMOUNT	INSTANCE	NONE
BEGIN	DOUBLE	INT	NOORDER
BECOME	DUMP	KEY	NORESETLOGS
BEFORE	EACH	LANGUAGE	NORMAL
BLOCK	ENABLE	LAYER	NOSORT
BODY	END	LINK	NUMERIC
CACHE	ESCAPE	LISTS	OFF
CANCEL	EVENTS	LOGFILE	OLD
CASCADE	EXCEPT	MANAGE	ONLY

Oracle Keywords			
CHANGE	EXCEPTIONS	MANUAL	OPEN
CHARACTER	EXEC	MAX	OPTIMAL
CHECKPOINT	EXPLAIN	MAXDATAFILES	OWN
CLOSE	EXECUTE	MAXINSTANCES	PACKAGE
COBOL	EXTENT	MAXLOGFILES	PARALLEL
COMMIT	EXTERNALLY	MAXLOGHISTORY	PCTINCREASE
COMPILE	FETCH	MAXLOGMEMBERS	PCTUSED
CONSTRAINT	FLUSH	MAXTRANS	PLAN
CONSTRAINTS	FREELIST	MAXVALUE	PLI
CONTENTS	FREELISTS	MIN	PRECISION
CONTINUE	FORCE	MINEXTENTS	PRIMARY
CONTROLFILE	FOREIGN	MINVALUE	PRIVATE
COUNT	FORTRAN	MODULE	PROCEDURE
PROFILE	SAVEPOINT	SQLSTATE	TRACING
QUOTA	SCHEMA	STATEMENT_ID	TRANSACTION
READ	SCN	STATISTICS	TRIGGERS
REAL	SECTION	STOP	TRUNCATE
RECOVER	SEGMENT	STORAGE	UNDER
REFERENCES	SEQUENCE	SUM	UNLIMITED
REFERENCING	SHARED	SWITCH	UNTIL
RESETLOGS	SNAPSHOT	SYSTEM	USE
RESTRICTED	SOME	TABLES	USING
REUSE	SORT	TABLESPACE	WHEN
ROLE	SQL	TEMPORARY	WRITE
ROLES	SQLCODE	THREAD	WORK
ROLLBACK	SQLERROR	TIME	

B.3 PL/SQL Reserved Words

The following PL/SQL keywords may require special treatment when used in embedded SQL statements.

PL/SQL Reserved Words			
ABORT	BETWEEN	CRASH	DIGITS
ACCEPT	BINARY_INTEGER	CREATE	DISPOSE
ACCESS	BODY	CURRENT	DISTINCT
ADD	BOOLEAN	CURRVAL	DO
ALL	BY	CURSOR	DROP
ALTER	CASE	DATABASE	ELSE

PL/SQL Reserved Words			
AND	CHAR	DATA_BASE	ELSIF
ANY	CHAR_BASE	DATE	END
ARRAY	CHECK	DBA	ENTRY
ARRAYLEN	CLOSE	DEBUGOFF	EXCEPTION
AS	CLUSTER	DEBUGON	EXCEPTION_INIT
ASC	CLUSTERS	DECLARE	EXISTS
ASSERT	COLAUTH	DECIMAL	EXIT
ASSIGN	COLUMNS	DEFAULT	FALSE
AT	COMMIT	DEFINITION	FETCH
AUTHORIZATION	COMPRESS	DELAY	FLOAT
AVG	CONNECT	DELETE	FOR
BASE_TABLE	CONSTANT	DELTA	FORM
BEGIN	COUNT	DESC	FROM
FUNCTION	NEW	RELEASE	SUM
GENERIC	NEXTVAL	REMR	TABAUTH
GOTO	NOCOMPRESS	RENAME	TABLE
GRANT	NOT	RESOURCE	TABLES
GROUP	NULL	RETURN	TASK
HAVING	NUMBER	REVERSE	TERMINATE
IDENTIFIED	NUMBER_BASE	REVOKE	THEN
IF	OF	ROLLBACK	TO
IN	ON	ROWID	TRUE
INDEX	OPEN	ROWLABEL	TYPE
INDEXES	OPTION	ROWNUM	UNION
INDICATOR	OR	ROWTYPE	UNIQUE
INSERT	ORDER	RUN	UPDATE
INTEGER	OTHERS	SAVEPOINT	USE
INTERSECT	OUT	SCHEMA	VALUES
INTO	PACKAGE	SELECT	VARCHAR
IS	PARTITION	SEPARATE	VARCHAR2
LEVEL	PCTFREE	SET	VARIANCE
LIKE	POSITIVE	SIZE	VIEW
LIMITED	PRAGMA	SMALLINT	VIEWS
LOOP	PRIOR	SPACE	WHEN
MAX	PRIVATE	SQL	WHERE
MIN	PROCEDURE	SQLCODE	WHILE
MINUS	PUBLIC	SQLERRM	WITH
MLSLABEL	RAISE	START	WORK

PL/SQL Reserved Words			
MOD	RANGE	STATEMENT	XOR
MODE	REAL	STDDEV	
NATURAL	RECORD	SUBTYPE	

B.4 Oracle Reserved Namespaces

Table B–1 contains a list of namespaces that are reserved by Oracle. The initial characters of function names in Oracle libraries are restricted to the character strings in this list. Because of potential name conflicts, use function names that do not begin with these characters.

For example, the SQL*Net Transparent Network Service functions all begin with the characters "NS," so you need to avoid naming functions that begin with "NS."

Table B–1 Oracle Reserved Namespaces

Namespace	Library
O	OCI functions
S	function names from SQLLIB and system-dependent libraries
XA	external functions for XA applications only
GEN KP L NA NC ND NL NM NR NS NT NZ TTC UPI	Internal functions

C

Performance Tuning

This appendix contains the following sections:

- What Causes Poor Performance?
- How Can Performance be Improved?
- Using Host Arrays
- Using Embedded PL/SQL
- Optimizing SQL Statements
- Using Indexes
- Taking Advantage of Row-Level Locking
- Eliminating Unnecessary Parsing

This appendix shows you some simple, easy-to-apply methods for improving the performance of your applications. Using these methods, you can often reduce processing time by 25% or more.

C.1 What Causes Poor Performance?

One cause of poor performance is high Oracle communication overhead. Oracle must process SQL statements one at a time. Thus, each statement results in another call to Oracle and higher overhead. In a networked environment, SQL statements must be sent over the network, adding to network traffic. Heavy network traffic can slow down your application significantly.

Another cause of poor performance is inefficient SQL statements. Because SQL is so flexible, you can get the same result with two different statements, but one statement might be less efficient. For example, the following two `SELECT` statements return the same rows (the name and number of every department having at least one employee):

```
EXEC SQL SELECT DNAME, DEPTNO
 FROM DEPT
 WHERE DEPTNO IN (SELECT DEPTNO FROM EMP);
EXEC SQL SELECT DNAME, DEPTNO
 FROM DEPT
 WHERE EXISTS
 (SELECT DEPTNO FROM EMP WHERE DEPT.DEPTNO = EMP.DEPTNO);
```

However, the first statement is slower because it does a time-consuming full scan of the `EMP` table for every department number in the `DEPT` table. Even if the `DEPTNO` column in `EMP` is indexed, the index is not used because the subquery lacks a `WHERE` clause naming `DEPTNO`.

A third cause of poor performance is unnecessary parsing and binding. Recall that before executing a SQL statement, Oracle must parse and bind it. Parsing means examining the SQL statement to make sure it follows syntax rules and refers to valid database objects. Binding means associating host variables in the SQL statement with their addresses so that Oracle can read or write their values.

Many applications manage cursors poorly. This results in unnecessary parsing and binding, which adds noticeably to processing overhead.

C.2 How Can Performance be Improved?

If you are unhappy with the performance of your precompiled programs, there are several ways you can reduce overhead.

You can greatly reduce Oracle communication overhead, especially in networked environments, by:

- Using host arrays
- Using embedded PL/SQL

You can reduce processing overhead --sometimes dramatically-- by:

- Optimizing SQL statements
- Using indexes
- Taking advantage of row-level locking
- Eliminating unnecessary parsing

C.3 Using Host Arrays

Host arrays can boost performance because they let you manipulate an entire collection of data with a single SQL statement. For example, suppose you want to insert salaries for 300 employees into the `EMP` table. Without arrays your program must do 300 individual inserts--one for each employee. With arrays, only one `INSERT` is necessary. Consider the following statement:

```
EXEC SQL INSERT INTO EMP (SAL) VALUES (:salary);
```

If *salary* is a simple host variable, Oracle executes the `INSERT` statement once, inserting a single row into the `EMP` table. In that row, the `SAL` column has the value of *salary*. To insert 300 rows this way, you must execute the `INSERT` statement 300 times.

However, if *salary* is a host array of size 300, Oracle inserts all 300 rows into the `EMP` table at once. In each row, the `SAL` column has the value of an element in the *salary* array.

For more information, see Chapter 9, "Using Host Arrays"

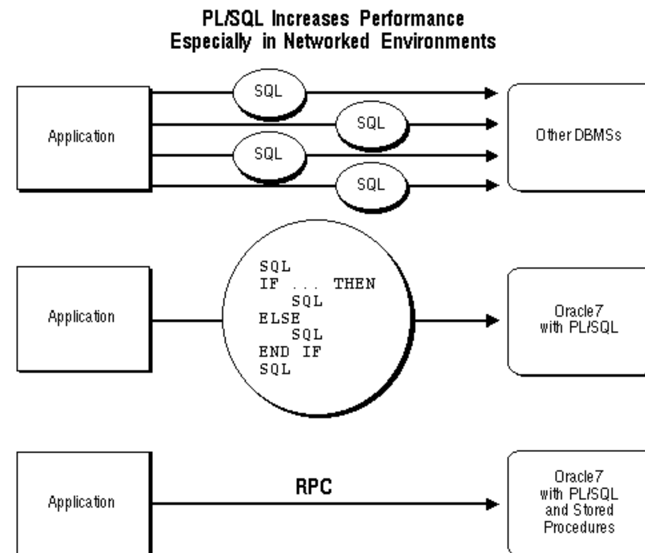
C.4 Using Embedded PL/SQL

As Figure C–1 shows, if your application is database-intensive, you can use control structures to group SQL statements in a PL/SQL block, then send the entire block to Oracle. This can drastically reduce communication between your application and Oracle.

Also, you can use PL/SQL subprograms to reduce calls from your application to Oracle. For example, to execute ten individual SQL statements, ten calls are required, but to execute a subprogram containing ten SQL statements, only one call is required.

Unlike anonymous blocks, PL/SQL subprograms can be compiled separately and stored in an Oracle 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 C–1 PL/SQL Boosts Performance



PL/SQL can also cooperate with Oracle application development tools such as Oracle Forms and Oracle Reports. By adding procedural processing power to these tools, PL/SQL boosts performance. Using PL/SQL, a tool can do any computation quickly and efficiently without calling on Oracle. This saves time and reduces network traffic. For more information, see Chapter 5, "Using Embedded PL/SQL" and the *Oracle Database PL/SQL Language Reference*.

C.5 Optimizing SQL Statements

For every SQL statement, the Oracle optimizer generates an *execution plan*, which is a series of steps that Oracle takes to execute the statement. These steps are determined by rules given in the *Oracle Database Advanced Application Developer's Guide*. Following these rules will help you write optimal SQL statements.

C.5.1 Optimizer Hints

For every SQL statement, the Oracle optimizer generates an *execution plan*, which is a series of steps that Oracle takes to execute the statement. In some cases, you can suggest to Oracle the way to optimize a SQL statement. These suggestions, called *hints*, let you influence decisions made by the optimizer.

Hints are not directives; they merely help the optimizer do its job. Some hints limit the scope of information used to optimize a SQL statement, while others suggest overall strategies. You can use hints to specify the:

- Optimization approach for a SQL statement
- Access path for each referenced table
- Join order for a join
- Method used to join tables

C.5.2 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 -- host arrays
 FROM EMP
 WHERE DEPTNO = :dept_number;
```

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 and hints.

For more information about optimizer hints, see the *Oracle Database Advanced Application Developer's Guide*.

C.5.3 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. Then, you can concentrate your tuning efforts on those statements.

The EXPLAIN PLAN statement shows the execution plan for each SQL statement in your application. You can use the execution plan to identify inefficient SQL statements.

C.6 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.

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 *Oracle Database Advanced Application Developer's Guide*.

C.7 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".

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.

C.8 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.

You can use the ORACA to get cursor cache statistics. See "Using the Oracle Communications Area".

C.8.1 Handling Explicit Cursors

Recall that there are two types of cursors: implicit and explicit. Oracle implicitly declares a cursor for all data definition and data manipulation statements. However, for queries that return more than one row, you must explicitly declare a cursor (or use host arrays). You use the DECLARE CURSOR statement to declare an explicit cursor. 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.

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.

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.

C.8.2 Cursor Control

In general, there are three ways to control an explicitly declared cursor:

- Use the DECLARE, OPEN, and CLOSE statements
- Use the PREPARE, DECLARE, OPEN, and CLOSE statements
- COMMIT closes the cursor when MODE=ANSI

With the first way, beware of unnecessary parsing. The OPEN statement does the parsing, but only if the parsed statement is unavailable because the cursor was closed or never opened. Your program should DECLARE the cursor, reopen it every time the value of a host variable changes, and CLOSE it only when the SQL statement is no longer needed.

With the second way (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, reopen the cursor every time the value of a host variable changes, rePREPARE the SQL statement and reopen the cursor if the SQL statement changes, and CLOSE the cursor only when the SQL statement is no longer needed.

When possible, avoid placing OPEN and CLOSE statements in a loop; this is a potential cause of unnecessary reparsing of the SQL statement. In the next example, both the OPEN and CLOSE statements are inside the outer while loop. When MODE=ANSI, the

CLOSE statement must be positioned as shown, because ANSI requires a cursor to be closed before being reopened.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ename, sal from emp where sal > :salary and
 sal <= :salary + 1000;
salary = 0;
while (salary < 5000)
{
 EXEC SQL OPEN emp_cursor;
 while (SQLCODE==0)
 {
 EXEC SQL FETCH emp_cursor INTO ....
 ...
 }
 salary += 1000;
 EXEC SQL CLOSE emp_cursor;
}
```

With MODE=ORACLE, however, a CLOSE statement can execute without the cursor being opened. By placing the CLOSE statement outside the outer while loop, you can avoid possible reparsing at each iteration of the OPEN statement.

```
...
while (salary < 5000)
{
 EXEC SQL OPEN emp_cursor;
 while (sqlca.sqlcode==0)
 {
 EXEC SQL FETCH emp_cursor INTO ....
 ...
 }
 salary += 1000;
}
EXEC SQL CLOSE emp_cursor;
```

C.8.3 Using the Cursor Management Options

A SQL statement need be parsed only once unless you change its makeup. For example, you change the makeup of a query by adding a column to its select list or WHERE clause. The HOLD_CURSOR, RELEASE_CURSOR, and MAXOPENCURSORS options give you some control over how Oracle manages the parsing and reparsing of SQL statements. Declaring an explicit cursor gives you maximum control over parsing.

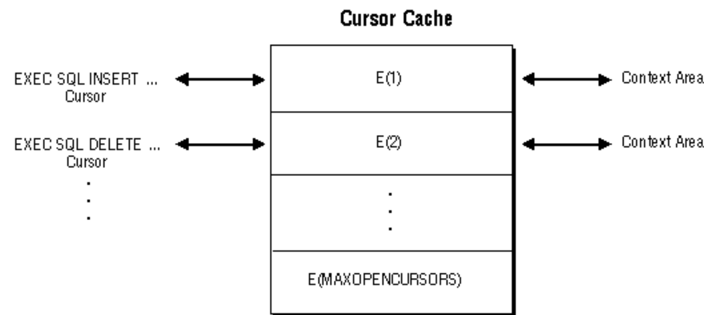
C.8.4 Private SQL Areas and Cursor Cache

When a data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache is a continuously updated area of memory used for cursor management. The cursor cache entry is in turn linked to a private SQL area.

The private SQL area, a work area created dynamically at run time by Oracle, contains the parsed SQL statement, the addresses of host variables, and other information needed to process the statement. An explicit cursor lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

Figure C–2 represents the cursor cache after your program has done an insert and a delete.

Figure C–2 Cursors Linked through the Cursor Cache



C.8.5 Resource Use

The maximum number of open cursors in each user session is set by the Oracle 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 cache entries actually needed, Oracle uses the first cache entry marked as reusable. For example, suppose the cache entry *E*(1) for an INSERT statement is marked as reusable, and the number of cache entries already equals MAXOPENCURSORS. If the program executes a new statement, cache entry *E*(1) and its private SQL area might be reassigned to the new statement. To reexecute the INSERT statement, Oracle would have to reparse it and reassign another cache entry.

Oracle allocates an additional cache entry if it cannot find one to reuse. For example, if MAXOPENCURSORS=8 and all eight entries are active, a ninth is created. If necessary, Oracle keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN_CURSORS. This dynamic allocation adds to processing overhead.

Thus, specifying a low value for MAXOPENCURSORS saves memory but causes potentially expensive dynamic allocations and deallocations of new cache entries. Specifying a high value for MAXOPENCURSORS assures speedy execution but uses more memory.

C.8.6 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, MAXOPENCURSORS is set low at your site to conserve memory.

If a data manipulation statement precedes a data definition statement and they reference the same tables, specify RELEASE_CURSOR=YES for the data manipulation statement. This avoids a conflict between the parse lock obtained by the data manipulation statement and the exclusive lock required by the data definition statement.

When RELEASE_CURSOR=YES, the link between the private SQL area and the cache entry is immediately removed and the private SQL area freed. Even if you specify HOLD_CURSOR=YES, Oracle must still reallocate memory for a private SQL area and reparse the SQL statement before executing it because RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES.

Nonetheless, when RELEASE_CURSOR=YES, the reparse 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.

C.8.7 Frequent Execution

The links between a *frequently* executed SQL statement and its private SQL area should be maintained, because the private SQL area contains all the information needed to execute the statement. Maintaining access to this information makes subsequent execution of the statement much faster.

When HOLD_CURSOR=YES, the link between the cursor and cursor cache is maintained after Oracle executes the SQL statement. Thus, the parsed statement and allocated memory remain available. This is useful for SQL statements that you want to keep active because it avoids unnecessary reparsing.

When HOLD_CURSOR=YES and RELEASE_CURSOR=NO (the default), the link between the cache entry and the private SQL area is maintained after Oracle executes the SQL statement and is not reused unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because the parsed statement and allocated memory remain available.

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 Oracle database version 7, 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 reparsed.

C.8.8 Parameter Interactions

Table C–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 C–1 HOLD_CURSOR 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

D Syntactic and Semantic Checking

This appendix contains the following sections:

- What Is Syntactic and Semantic Checking?
- Controlling the Type and Extent of Checking
- Specifying SQLCHECK=SEMANTICS

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.

D.1 What Is Syntactic and Semantic Checking?

Rules of syntax specify how language elements are sequenced to form valid statements. Thus, *syntactic checking* verifies that keywords, object names, operators, delimiters, and so on are placed correctly in your SQL statement. For example, the following embedded SQL statements contain syntax errors:

```
-- misspelled keyword WHERE
EXEC SQL DELETE FROM EMP WERE DEPTNO = 20;
-- missing parentheses around column names COMM and SAL
EXEC SQL INSERT INTO EMP COMM, SAL VALUES (NULL, 1500);
```

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;
-- undeclared host variable, emp_name
EXEC SQL SELECT * FROM EMP WHERE ENAME = :emp_name;
```

The rules of SQL syntax and semantics are defined in the *Oracle Database SQL Language Reference*.

D.2 Controlling the Type and Extent of Checking

You control the type and extent of checking by specifying the SQLCHECK option on the command line. With SQLCHECK, the type of checking can be syntactic, semantic, or both. The extent of checking can include data manipulation statements and PL/SQL blocks. However, SQLCHECK cannot check dynamic SQL statements because they are not defined fully until run time.

You can specify the following values for SQLCHECK:

- SEMANTICS|FULL
- SYNTAX|LIMITED|NONE

The values SEMANTICS and FULL are equivalent, as are the values SYNTAX and LIMITED. The default value is SYNTAX.

D.3 Specifying SQLCHECK=SEMANTICS

When SQLCHECK=SEMANTICS, the precompiler checks the syntax and semantics of:

- Data manipulation statements such as INSERT and UPDATE
- PL/SQL blocks

However, the precompiler checks only the syntax of remote data manipulation statements (those using the AT *db_name* clause).

The precompiler gets the information for a semantic check from embedded DECLARE TABLE statements or, if you specify the option USERID, by connecting to 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, then 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 Oracle database version 7 set of syntax rules found in the *Oracle Database SQL Language 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, then you *must* specify SQLCHECK=SEMANTICS.

D.3.1 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

D.3.2 Connecting to Oracle

To do a semantic check, the precompiler can connect to an Oracle database that maintains definitions of tables and views referenced in your host program. After connecting to Oracle, 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 Oracle, specify the option USERID on the command line, using the syntax:

```
USERID=username
```

where *username* is a valid Oracle userid. You are prompted for the password. If, instead of a username, you specify

```
USERID=/
```

the precompiler tries to connect 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 null) and *username* is your operating system user or task name.

If you try connecting to Oracle 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.

D.3.3 Using DECLARE TABLE

The precompiler can do a semantic check without connecting to Oracle. To do the check, the precompiler must get information about tables and views from embedded DECLARE TABLE statements. Thus, every table referenced in a data manipulation statement or PL/SQL block must be defined in a DECLARE TABLE statement.

The syntax of the DECLARE TABLE statement is:

```
EXEC SQL DECLARE table_name TABLE
 (col_name col_datatype [DEFAULT expr] [NULL|NOT NULL], ...);
```

where *expr* is any expression that can be used as a default column value in the CREATE TABLE statement.

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.

E

Embedded SQL Commands and Directives

This appendix contains the following sections:

- Summary of Precompiler Directives and Embedded SQL Commands
- About The Command Descriptions
- How to Read Syntax Diagrams
- ALLOCATE (Executable Embedded SQL Extension)
- CLOSE (Executable Embedded SQL)
- COMMIT (Executable Embedded SQL)
- CONNECT (Executable Embedded SQL Extension)
- DECLARE CURSOR (Embedded SQL Directive)
- DECLARE DATABASE (Oracle Embedded SQL Directive)
- DECLARE STATEMENT (Embedded SQL Directive)
- DECLARE TABLE (Oracle Embedded SQL Directive)
- DELETE (Executable Embedded SQL)
- DESCRIBE (Executable Embedded SQL)
- EXECUTE ... END-EXEC (Executable Embedded SQL Extension)
- EXECUTE (Executable Embedded SQL)
- EXECUTE IMMEDIATE (Executable Embedded SQL)
- FETCH (Executable Embedded SQL)
- INSERT (Executable Embedded SQL)
- OPEN (Executable Embedded SQL)
- PREPARE (Executable Embedded SQL)
- ROLLBACK (Executable Embedded SQL)
- SAVEPOINT (Executable Embedded SQL)
- SELECT (Executable Embedded SQL)
- UPDATE (Executable Embedded SQL)
- VAR (Oracle Embedded SQL Directive)
- WHENEVER (Embedded SQL Directive)

This appendix contains descriptions of both SQL92 embedded SQL commands and directives and the Oracle embedded SQL extensions. These commands and directives are prefaced in your source code with the keywords, `EXEC SQL`. Rather than trying to memorize all of the SQL syntax, simply refer to this appendix, which includes the following:

- A summary of embedded SQL commands and directives
- A section about the command descriptions
- How to read syntax diagrams
- An alphabetic listing of the commands and directives

E.1 Summary of Precompiler Directives and Embedded SQL Commands

Embedded SQL commands place DDL, DML, and Transaction Control statements within a procedural language program. Embedded SQL is supported by the Oracle Precompilers. Table E–1 provides a functional summary of the embedded SQL commands and directives.

The Type column in Table E–1 is displayed in the format, *source/type*, where *source* is either SQL92 standard SQL (S) or an Oracle extension (O) and *type* is either an executable (E) statement or a directive (D).

Table E–1 Summary of Embedded SQL Commands and Directives

EXEC SQL Statement	Type	Purpose
ALLOCATE	O/E	To allocate memory for a cursor variable.
CLOSE	S/E	To disable a cursor, releasing the resources it holds.
COMMIT	S/E	To end the current transaction, making all database change permanent (optionally frees resources and disconnects from the database)
CONNECT	O/E	To log on to an Oracle instance.
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.
EXECUTE...END-EXEC	O/E	To execute an anonymous PL/SQL block.
EXECUTE	S/E	To execute a prepared dynamic SQL statement.
EXECUTE IMMEDIATE	S/E	To prepare and execute a SQL statement with no host variables.
FETCH	S/E	To retrieve rows selected by a query.
INSERT	S/E	To add rows to a table or to a view's base table.

Table E–1 Summary of Embedded SQL Commands and Directives

EXEC SQL Statement	Type	Purpose
OPEN	S/E	To execute the query associated with a cursor.
PREPARE	S/E	To parse a dynamic SQL statement.
ROLLBACK	S/E	To end the current transaction, discard all changes in the current transaction, and release all locks (optionally release resources and disconnect from the database).
SAVEPOINT	S/E	To identify a point in a transaction to which you can later roll back.
SELECT	S/E	To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.
UPDATE	S/E	To change existing values in a table or in a view's base table.
VAR	O/D	To override the default datatype and assign a specific Oracle datatype to a host variable.
WHENEVER	S/D	To specify handling for error and warning conditions.

E.2 About The Command Descriptions

The directives, commands, and clauses appear alphabetically. The description of each contains the following sections:

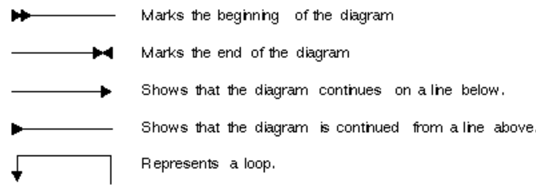
Heading	Meaning
Purpose	describes the basic uses of the command.
Prerequisites	lists privileges you must have and steps that you must take before using the command. Unless otherwise noted, most commands also require that the database be open by your instance.
Syntax	shows the keywords and parameters of the command.
Keywords and Parameters	describes the purpose of each keyword and parameter.
Usage Notes	discusses how and when to use the command.
Examples	shows example statements of the command.
Related Topics	lists related commands, clauses, and sections of this manual.

E.3 How to Read Syntax Diagrams

Easy-to-understand *syntax diagrams* are used to illustrate embedded SQL syntax. They are line-and-arrow drawings that depict valid syntax. If you have never used them, do not worry. This section tells you all you need to know.

After you understand the logical flow of a syntax diagram, it becomes a helpful guide. You can verify or construct any embedded SQL statement by tracing through its syntax diagram.

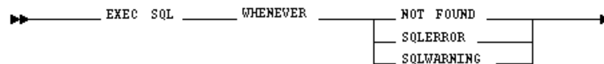
Syntax diagrams use lines and arrows to show how commands, parameters, and other language elements are sequenced to form statements. Trace each diagram from left to right, in the direction shown by the arrows. The following symbols will guide you:



Commands and other keywords appear in uppercase. Parameters appear in lowercase. Operators, delimiters, and terminators appear as usual. Following the conventions defined in the Preface, a semicolon terminates statements.

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:

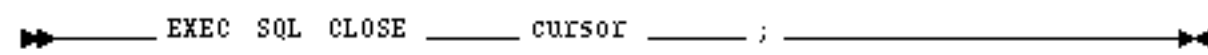


According to the diagram, all of the following statements are valid:

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

E.3.1 Required Keywords and Parameters

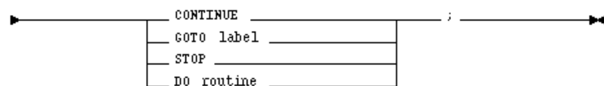
Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *cursor* is a required parameter:



If there is a cursor named *emp_cursor*, then, according to the diagram, the following statement is valid:

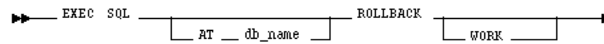
```
EXEC SQL CLOSE emp_cursor;
```

If any of the keywords or parameters in a vertical list appears on the main path, one of them is required. That is, you must choose one of the keywords or parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the four actions:



E.3.2 Optional Keywords and Parameters

If keywords and parameters appear in a vertical list the main path, they are optional. That is, you need not choose one of them. In the following example, instead of traveling down a vertical line, you can continue along the main path:

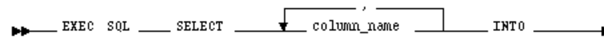


If there is a database named *oracle2*, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL ROLLBACK;
EXEC SQL ROLLBACK WORK;
EXEC SQL AT oracle2 ROLLBACK;
```

E.3.3 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.

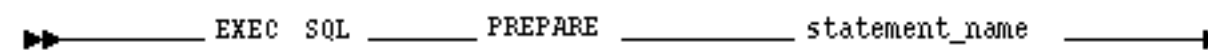


If DEBIT, CREDIT, and BALANCE are column names, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

E.3.4 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 sql_statement FROM :sql_string;
```

E.3.5 Database Objects

The names of Oracle objects, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if an Oracle identifier is enclosed by quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle identifiers are not case-sensitive except when enclosed by quotation marks.

E.4 ALLOCATE (Executable Embedded SQL Extension)

E.4.1 Purpose

To allocate a cursor variable to be referenced in a PL/SQL block.

E.4.2 Prerequisites

A cursor variable of type SQL_CURSOR must be declared before allocating memory for the cursor variable.

E.4.3 Syntax

```
▶▶── EXEC SQL ALLOCATE :cursor_variable ──────────▶◀
```

E.4.4 Keywords and Parameters

:cursor_variable

The cursor variable to be allocated.

E.4.5 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.

Example

This partial example illustrates the use of the ALLOCATE command in a Pro*C/C++ embedded SQL program:

```
EXEC SQL BEGIN DECLARE SECTION;
 SQL_CURSOR emp_cv;
 struct{ ... } emp_rec;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE emp_cv;
EXEC SQL EXECUTE
 BEGIN
 OPEN :emp_cv FOR SELECT * FROM emp;
 END;
END-EXEC;
for (;;)
{ EXEC SQL FETCH :emp_cv INTO :emp_rec;
}
```

E.4.6 Related Topics

"CLOSE (Executable Embedded SQL)", "EXECUTE (Executable Embedded SQL)", and "FETCH (Executable Embedded SQL)"

E.5 CLOSE (Executable Embedded SQL)

E.5.1 Purpose

To disable a cursor, freeing the resources acquired by opening the cursor, and releasing parse locks.

E.5.2 Prerequisites

The cursor or cursor variable must be open and MODE=ANSI.

E.5.3 Syntax

```
>>--- EXEC SQL CLOSE ---+-- cursor ------------+------------->|
                        +-- :cursor_variable --+
```

E.5.4 Keywords and Parameters

`cursor`

A cursor to be closed.

`cursor_variable`

A cursor variable to be closed.

E.5.5 Usage Notes

Rows cannot be fetched from a closed cursor. A cursor need not be closed to be reopened. The HOLD_CURSOR and RELEASE_CURSOR precompiler options alter the effect of the CLOSE command. For information on these options, see Chapter 6, "Running the Oracle Precompilers".

E.5.6 Example

This example illustrates the use of the CLOSE command:

```
EXEC SQL CLOSE emp_cursor;
```

E.5.7 Related Topics

"DECLARE CURSOR (Embedded SQL Directive)", "OPEN (Executable Embedded SQL)", and "PREPARE (Executable Embedded SQL)"

E.6 COMMIT (Executable Embedded SQL)

E.6.1 Purpose

To end your current transaction, making permanent all its changes to the database and optionally freeing all resources and disconnecting from the Oracle database.

E.6.2 Prerequisites

To commit your current transaction, no privileges are necessary.

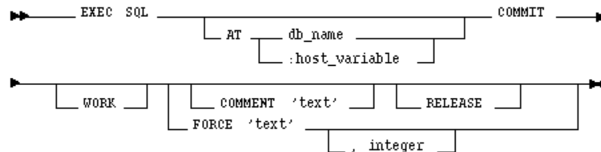
To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

If you are using Oracle in DBMS MAC mode, you can only commit an in-doubt transaction if your DBMS label matches the label the transaction's label and the creation label of the user who originally committed the transaction or if you satisfy one of the following criteria:

- If the transaction's label or the user's creation label is higher than your DBMS label, you must have READUP and WRITEUP system privileges.

- If the transaction's label or the user's creation label is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the transaction's label or the user's creation label is not comparable with your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

E.6.3 Syntax



E.6.4 Keyword and Parameters

AT

Identifies the database to which the COMMIT statement is issued. The database can be identified by either:

- *db_name* is a database identifier declared in a previous DECLARE DATABASE statement.
- *:host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, Oracle 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 Oracle 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 Oracle database.

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.

E.6.5 Usage Notes

Always explicitly commit or rollback the last transaction in your program by using the COMMIT or ROLLBACK command and the RELEASE option. Oracle automatically rolls back changes if the program terminates abnormally.

The COMMIT command has no effect on host variables or on the flow of control in the program. For more information on this command, see Chapter 7, "Defining and Controlling Transactions".

Example

This example illustrates the use of the embedded SQL COMMIT command:

```
EXEC SQL AT sales_db COMMIT RELEASE;
```

E.6.6 Related Topics

"ROLLBACK (Executable Embedded SQL)" and "SAVEPOINT (Executable Embedded SQL)"

E.7 CONNECT (Executable Embedded SQL Extension)

E.7.1 Purpose

To log on to an Oracle database.

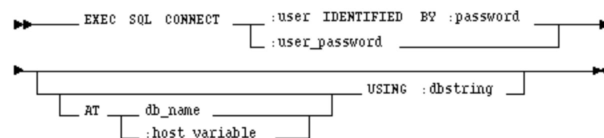
E.7.2 Prerequisites

You must have CREATE SESSION system privilege in the specified database.

If you are using Oracle in DBMS MAC mode, your operating system label must dominate both your creation label and the label at which you were granted CREATE SESSION system privilege. Your operating system label must also fall between the operating system equivalents of DBHIGH and DBLOW, inclusive.

If you are using Oracle in operating system MAC mode, your operating system label must match the label of the database to which you are connecting.

E.7.3 Syntax



E.7.4 Keyword and Parameters

:*user* :*password*

specifies your username and password separately.

:*user_password*

is a single host variable containing the Oracle username and password separated by a slash (/).

To allow Oracle to verify your connection through your operating system, specify "/" as the:*user_password* value.

AT

identifies the database to which the connection is made. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:host_variable is a host variable whose value is a previously declared *db_name*.

USING

specifies the SQL*Net database specification string used to connect to a nondefault database. If you omit this clause, you are connected to your default database.

E.7.5 Usage Notes

A program can have multiple connections, but can only connect once to your default database. For more information on this command, see Chapter 3, "Meeting Program Requirements".

Example

The following example illustrate the use of CONNECT:

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

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
```

E.7.6 Related Topics

"COMMIT (Executable Embedded SQL)", "DECLARE DATABASE (Oracle Embedded SQL Directive)", and "ROLLBACK (Executable Embedded SQL)"

E.8 DECLARE CURSOR (Embedded SQL Directive)

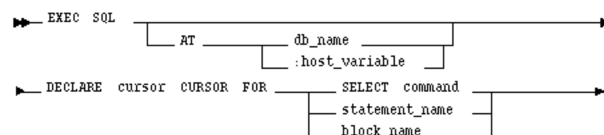
E.8.1 Purpose

To declare a cursor, giving it a name and associating it with a SQL statement or a PL/SQL block.

E.8.2 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.

E.8.3 Syntax



E.8.4 Keywords and Parameters

AT

identifies the database on which the cursor is declared. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, Oracle declares the cursor on your default database.

cursor

is the name of the cursor to be declared.

`SELECT` *command*

is a `SELECT` statement to be associated with the cursor. The following statement cannot contain an `INTO` clause.

statement_name block_name

identifies a SQL statement or PL/SQL block to be associated with the cursor. The *statement_name* or *block_name m*ust be previously declared in a `DECLARE STATEMENT` statement.

E.8.5 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, then the cursor has been opened with an OPEN statement and positioned on a row with a `FETCH` statement. For more information on this command, see Chapter 3, "Meeting Program Requirements".

E.8.6 Example

This example illustrates the use of a `DECLARE  CURSOR` statement:

```
EXEC SQL DECLARE emp_cursor CURSOR
 FOR SELECT ename, empno, job, sal
 FROM emp
 WHERE deptno = :deptno
 FOR UPDATE OF sal
```

E.8.7 Related Topics

"CLOSE (Executable Embedded SQL)", "DECLARE DATABASE (Oracle Embedded SQL Directive)", "DECLARE STATEMENT (Embedded SQL Directive)", "DELETE (Executable Embedded SQL)", "FETCH (Executable Embedded SQL)", "OPEN (Executable Embedded SQL)", "PREPARE (Executable Embedded SQL)", "SELECT (Executable Embedded SQL)", and "UPDATE (Executable Embedded SQL)"

E.9 DECLARE DATABASE (Oracle Embedded SQL Directive)

E.9.1 Purpose

To declare an identifier for a nondefault database to be accessed in subsequent embedded SQL statements.

E.9.2 Prerequisites

You must have access to a username on the nondefault database.

E.9.3 Syntax

```
▶▶─ EXEC SQL DECLARE db_name DATABASE ──────────────────▶◀
```

E.9.4 Keywords and Parameters

db_name

is the identifier established for the nondefault database.

E.9.5 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 command, see Chapter 3, "Meeting Program Requirements".

E.9.6 Example

This example illustrates the use of a DECLARE DATABASE directive:

```
EXEC SQL DECLARE oracle3 DATABASE
```

E.9.7 Related Topics

"COMMIT (Executable Embedded SQL)", "CONNECT (Executable Embedded SQL Extension)", "DECLARE CURSOR (Embedded SQL Directive)", "DECLARE STATEMENT (Embedded SQL Directive)", "DELETE (Executable Embedded SQL)", "EXECUTE (Executable Embedded SQL)", "EXECUTE IMMEDIATE (Executable Embedded SQL)", "INSERT (Executable Embedded SQL)", "SELECT (Executable Embedded SQL)", and "UPDATE (Executable Embedded SQL)"

E.10 DECLARE STATEMENT (Embedded SQL Directive)

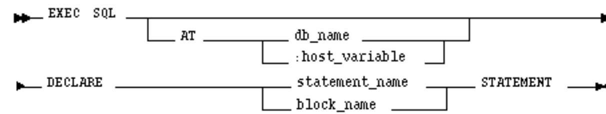
E.10.1 Purpose

To declare an identifier for a SQL statement or PL/SQL block to be used in other embedded SQL statements.

E.10.2 Prerequisites

None.

E.10.3 Syntax



E.10.4 Keywords and Parameters

AT

identifies the database on which the SQL statement or PL/SQL block is declared. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, Oracle declares the SQL statement or PL/SQL block on your default database.

statement_name block_name

is the declared identifier for the statement.

E.10.5 Usage Notes

You must declare an identifier for a SQL statement or PL/SQL block with a DECLARE STATEMENT statement only if a DECLARE CURSOR statement referencing the identifier appears physically (not logically) in the embedded SQL program before the PREPARE statement that parses the statement or block and associates it with its identifier.

The scope of a statement declaration is global within its precompilation unit, like a cursor declaration.For more information on this command, see Chapter 3, "Meeting Program Requirements" and Chapter 10, "Using Dynamic SQL".

E.10.6 Example I

This example illustrates the use of the DECLARE STATEMENT statement:

```
EXEC SQL AT remote_db
 DECLARE my_statement STATEMENT
EXEC SQL PREPARE my_statement FROM :my_string
EXEC SQL EXECUTE my_statement
```

E.10.7 Example II

In this example from a Pro*C/C++ embedded SQL program, the DECLARE STATEMENT statement is required because the DECLARE CURSOR statement precedes the PREPARE statement:

```
EXEC SQL DECLARE my_statement STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR my_statement;
EXEC SQL PREPARE my_statement FROM :my_string;
...
```

E.10.8 Related Topics

"CLOSE (Executable Embedded SQL)", "DECLARE DATABASE (Oracle Embedded SQL Directive)", "FETCH (Executable Embedded SQL)", "OPEN (Executable Embedded SQL)", and "PREPARE (Executable Embedded SQL)"

E.11 DECLARE TABLE (Oracle Embedded SQL Directive)

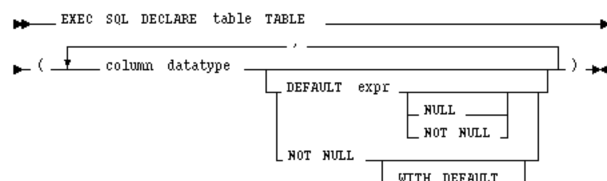
E.11.1 Purpose

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 Oracle Precompilers.

E.11.2 Prerequisites

None.

E.11.3 Syntax



E.11.4 Keywords and Parameters

table

is the name of the declared table.

column

is a column of the *table*.

datatype

is the datatype of a *column*.

DEFAULT

specifies the default value of a *column*.

NULL

specifies that a *column* can contain nulls.

NOT NULL

specifies that a *column* cannot contain nulls.

WITH DEFAULT

is supported for compatibility with the IBM DB2 database.

E.11.5 Usage Notes

For information on using this command, see Chapter 3, "Meeting Program Requirements".

E.11.6 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)
```

E.11.7 Related Topics

None.

E.12 DELETE (Executable Embedded SQL)

E.12.1 Purpose

To remove rows from a table or from a view's base table.

E.12.2 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 delete rows from any table or any view's base table.

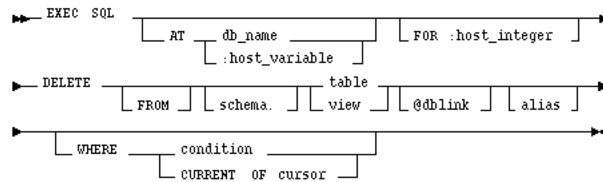
If you are using Oracle in DBMS MAC mode, your DBMS label must dominate the creation label of the table or view or you must meet one of the following criteria:

- If the creation label of the table or view is higher than your DBMS label, you must have READUP and WRITEUP system privileges.
- If the creation label of your table or view is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

In addition, for each row to be deleted, your DBMS label must match the row's label or you must meet one of the following criteria:

- If the row's label is higher than your DBMS label, you must have READUP and WRITEUP system privileges.
- If the row's label is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the row label is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

E.12.3 Syntax



E.12.4 Keywords and Parameters

`AT`

identifies the database to which the `DELETE` statement is issued. The database can be identified by either:

db_name is a database identifier declared in a previous `DECLARE DATABASE` statement.

:host_variable is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the `DELETE` statement is issued to your default database.

`FOR` *:host_integer*

limits the number of times the statement is executed if the `WHERE` clause contains array host variables. If you omit this clause, Oracle executes the statement once for each component of the smallest array.

schema

is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table view

is the name of a table from which the rows are to be deleted. If you specify *view*, Oracle deletes rows from the view's base table.

dblink

is the complete or partial name of a database link to a remote database where the table or view is located. You can only delete rows from a remote table or view if you are using Oracle with the distributed option.

If you omit *dblink*, Oracle assumes that the table or view is located on the local database.

alias

is an alias assigned to the table. Aliases are generally used in `DELETE` statements with correlated queries.

`WHERE`

specifies which rows are deleted:

condition deletes only rows that satisfy the condition. This condition can contain host variables and optional indicator variables.

`CURRENT OF` deletes only the row most recently fetched by the *cursor*. The *cursor* cannot be associated with a `SELECT` statement that performs a join, unless its `FOR UPDATE` clause specifically locks only one table.

If you omit this clause entirely, Oracle deletes all rows from the table or view.

E.12.5 Usage Notes

The host variables in the WHERE clause must be either all scalars or all arrays. If they are scalars, Oracle executes the DELETE statement only once. If they are arrays, Oracle 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 Oracle 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, Oracle returns an error through the SQLCODE of the SQLCA. If you omit the WHERE clause, Oracle raises a warning flag in the fifth component of SQLWARN in the SQLCA. For more information on this command and the SQLCA, see Chapter 8, "Error Handling and Diagnostics".

You can use comments in a DELETE statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement.

E.12.6 Example

This example illustrates the use of the DELETE statement within a Pro*C/C++ embedded SQL program:

```
EXEC SQL DELETE FROM emp
 WHERE deptno = :deptno
 AND job = :job; ...
EXEC SQL DECLARE emp_cursor CURSOR
 FOR SELECT empno, comm
 FROM emp;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH c1
 INTO :emp_number, :commission;
EXEC SQL DELETE FROM emp
 WHERE CURRENT OF emp_cursor;
```

E.12.7 Related Topics

"DECLARE DATABASE (Oracle Embedded SQL Directive)" and "DECLARE STATEMENT (Embedded SQL Directive)"

E.13 DESCRIBE (Executable Embedded SQL)

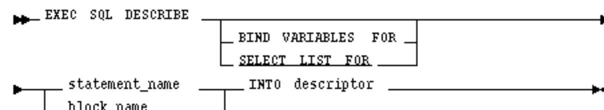
E.13.1 Purpose

To initialize a descriptor to hold descriptions of host variables for a dynamic SQL statement or PL/SQL block.

E.13.2 Prerequisites

You must have prepared the SQL statement or PL/SQL block in a previous embedded SQL `PREPARE` statement.

E.13.3 Syntax



E.13.4 Keywords and Parameters

`BIND VARIABLES`

initializes the descriptor to hold information about the input variables for the SQL statement or PL/SQL block.

`SELECT LIST`

initializes the descriptor to hold information about the select list of a `SELECT` statement.

The default is `SELECT LIST FOR`.

statement_name block_name

identifies a SQL statement or PL/SQL block previously prepared with a `PREPARE` statement.

descriptor

is the name of the descriptor to be initialized.

E.13.5 Usage Notes

You must issue a `DESCRIBE` statement before manipulating the bind or select descriptor within an embedded SQL program.

You cannot describe both input variables and output variables into the same descriptor.

The number of variables found by a `DESCRIBE` statement is the total number of placeholders in the prepare SQL statement or PL/SQL block, rather than the total number of uniquely named placeholders. For more information on this command, see Chapter 10, "Using Dynamic SQL".

E.13.6 Example

This example illustrates the use of the `DESCRIBE` statement in a Pro*C embedded SQL program:

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL DECLARE emp_cursor
 FOR SELECT empno, ename, sal, comm
 FROM emp
```

```
 WHERE deptno = :dept_number
EXEC SQL DESCRIBE BIND VARIABLES FOR my_statement
 INTO bind_descriptor;
EXEC SQL OPEN emp_cursor
 USING bind_descriptor;
EXEC SQL DESCRIBE SELECT LIST FOR my_statement
 INTO select_descriptor;
EXEC SQL FETCH emp_cursor
 INTO select_descriptor;
```

E.13.7 Related Topics

"PREPARE (Executable Embedded SQL)"

E.14 EXECUTE ... END-EXEC (Executable Embedded SQL Extension)

E.14.1 Purpose

To embed an anonymous PL/SQL block into an Oracle Precompiler program.

E.14.2 Prerequisites

None.

E.14.3 Syntax

```
EXEC SQL [AT {db_name | :host_variable}] EXECUTE pl/sql_block END-EXEC
```

E.14.4 Keywords and Parameters

`AT`

identifies the database on which the PL/SQL block is executed. The database can be identified by either:

db_name is a database identifier declared in a previous `DECLARE DATABASE` statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the PL/SQL block is executed on your default database.

pl/sql_block

`END-EXEC`

must appear after the embedded PL/SQL block, regardless of which programming language your Oracle Precompiler program uses. Of course, the keyword `END-EXEC` must be followed by the embedded SQL statement terminator for the specific language.

E.14.5 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 5, "Using Embedded PL/SQL"

E.14.6 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
```

E.14.7 Related Topics

"EXECUTE IMMEDIATE (Executable Embedded SQL)"

E.15 EXECUTE (Executable Embedded SQL)

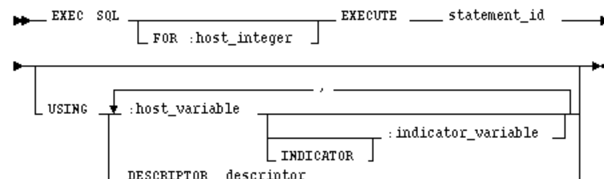
E.15.1 Purpose

To execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block that has been previously prepared with an embedded SQL PREPARE statement.

E.15.2 Prerequisites

You must first prepare the SQL statement or PL/SQL block with an embedded SQL PREPARE statement.

E.15.3 Syntax



E.15.4 Keywords and Parameters

FOR :*host_integer*

limits the number of times the statement is executed when the USING clause contains array host variables If you omit this clause, Oracle executes the statement once for each component of the smallest array.

statement_id

is a precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE command to associate the precompiler identifier with the statement or PL/SQL block.

USING

specifies a list of host variables with optional indicator variables that Oracle substitutes as input variables into the statement to be executed. The host and indicator variables must be either all scalars or all arrays.

E.15.5 Usage Notes

For more information on this command, see Chapter 10, "Using Dynamic SQL".

E.15.6 Example

This example illustrates the use of the EXECUTE statement in a Pro*C/C++ embedded SQL program:

```
EXEC SQL PREPARE my_statement
 FROM :my_string;
EXEC SQL EXECUTE my_statement
 USING :my_var;
```

E.15.7 Related Topics

"DECLARE DATABASE (Oracle Embedded SQL Directive)" and "PREPARE (Executable Embedded SQL)"

E.16 EXECUTE IMMEDIATE (Executable Embedded SQL)

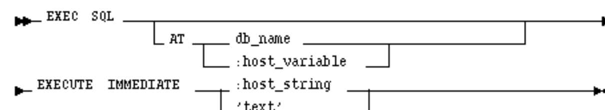
E.16.1 Purpose

To prepare and execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block containing no host variables.

E.16.2 Prerequisites

None.

E.16.3 Syntax



E.16.4 Keywords and Parameters

AT

identifies the database on which the SQL statement or PL/SQL block is executed. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the statement or block is executed on your default database.

:*host_string*

is a host variable whose value is the SQL statement or PL/SQL block to be executed.

text

is a quoted text literal containing the SQL statement or PL/SQL block to be executed.

The SQL statement can only be a `DELETE`, `INSERT`, or `UPDATE` statement.

E.16.5 Usage Notes

When you issue an `EXECUTE IMMEDIATE` statement, Oracle parses the specified SQL statement or PL/SQL block, checking for errors, and executes it. If any errors are encountered, they are returned in the `SQLCODE` component of the SQLCA.

For more information on this command, see Chapter 10, "Using Dynamic SQL".

E.16.6 Example

This example illustrates the use of the `EXECUTE IMMEDIATE` statement:

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM emp WHERE empno = 9460'
```

E.16.7 Related Topics

"EXECUTE (Executable Embedded SQL)" and "PREPARE (Executable Embedded SQL)"

E.17 FETCH (Executable Embedded SQL)

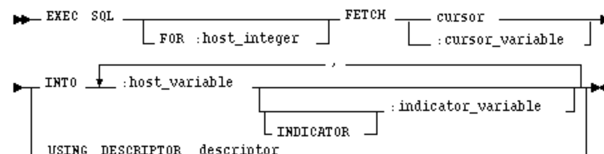
E.17.1 Purpose

To retrieve one or more rows returned by a query, assigning the select list values to host variables.

E.17.2 Prerequisites

You must first open the cursor with an the `OPEN` statement.

E.17.3 Syntax



E.17.4 Keywords and Parameters

`FOR` :*host_integer*

limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle fetches enough rows to fill the smallest array.

cursor

is a cursor that is declared by a `DECLARE CURSOR` statement. The `FETCH` statement returns one of the rows selected by the query associated with the cursor.

:*cursor_variable*

is a cursor variable is allocated an `ALLOCATE` statement. The `FETCH` statement returns one of the rows selected by the query associated with the cursor variable.

`INTO`

specifies a list of host variables and optional indicator variables into which data is fetched. These host variables and indicator variables must be declared within the program.

`USING`

specifies the descriptor referenced in a previous `DESCRIBE` statement. Only use this clause with dynamic embedded SQL, method 4. Also, the `USING` clause does not apply when a cursor variable is used.

E.17.5 Usage Notes

The `FETCH` statement reads the rows of the active set and names the output variables which contain the results. Indicator values are set to -1 if their associated host variable is null. The first `FETCH` statement for a cursor also sorts the rows of the active set, if necessary.

The number of rows retrieved is specified by the size of the output host variables and the value specified in the `FOR` clause. The host variables to receive the data must be either all scalars or all arrays. If they are scalars, Oracle fetches only one row. If they are arrays, Oracle fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle fetches is determined by the smaller of the following values:

- The size of the smallest array
- The value of the :*host_integer* in the optional `FOR` clause

Of course, the number of rows fetched can be further limited by the number of rows that actually satisfy the query.

If a `FETCH` statement does not retrieve all rows returned by the query, the cursor is positioned on the next returned row. When the last row returned by the query has been retrieved, the next `FETCH` statement results in an error code returned in the `SQLCODE` element of the SQLCA.

Note that the `FETCH` command does not contain an `AT` clause. You must specify the database accessed by the cursor in the `DECLARE CURSOR` statement.

You can only move forward through the active set with `FETCH` statements. If you want to revisit any of the previously fetched rows, you must reopen the cursor and fetch each row in turn. If you want to change the active set, you must assign new values to the input host variables in the cursor's query and reopen the cursor.

E.17.6 Example

This example illustrates the FETCH command in a pseudo-code embedded SQL program:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT job, sal FROM emp WHERE deptno = 30;
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :job_title1, :salary1;
 EXEC SQL FETCH emp_cursor INTO :job_title2, :salary2;
...
END LOOP;
...
```

E.17.7 Related Topics

"CLOSE (Executable Embedded SQL)", "DECLARE CURSOR (Embedded SQL Directive)", "OPEN (Executable Embedded SQL)", and "PREPARE (Executable Embedded SQL)"

E.18 INSERT (Executable Embedded SQL)

E.18.1 Purpose

To add rows to a table or to a view's base table.

E.18.2 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.

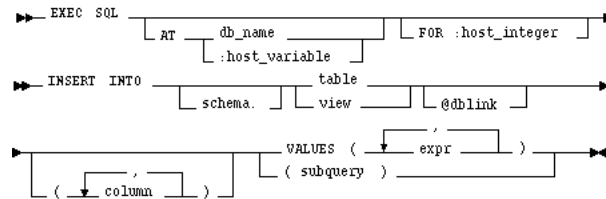
For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

The INSERT ANY TABLE system privilege also enables insert rows into any table or any view's base table.

If you are using Oracle in DBMS MAC mode, your DBMS label must match the creation label of the table or view:

- If the creation label of the table or view is higher than your DBMS label, you must have WRITEUP system privileges.
- If the creation label of the table or view is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the creation label of your table or view is not comparable to your DBMS label, you must have WRITEUP and WRITEDOWN system privileges.

E.18.3 Syntax



E.18.4 Keywords and Parameters

AT

identifies the database on which the INSERT statement is executed. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*

If you omit this clause, the INSERT statement is executed on your default database.

FOR :*host_integer*

limits the number of times the statement is executed if the VALUES clause contains array host variables. If you omit this clause, Oracle executes the statement once for each component in the smallest array.

schema

is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table view

is the name of the table into which rows are to be inserted. If you specify *view*, Oracle inserts rows into the view's base table.

dblink

is a complete or partial name of a database link to a remote database where the table or view is located. You can only insert rows into a remote table or view if you are using Oracle with the distributed option.

If you omit *dblink*, Oracle assumes that the table or view is on the local database.

column

is a column of the table or view. In the inserted row, each column in this list is assigned a value from the VALUES clause or the query.

If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. If you omit the column list altogether, the VALUES clause or query must specify values for all columns in the table.

VALUES

specifies a row of values to be inserted into the table or view. Note that the expressions can be host variables with optional indicator variables. You must specify an expression in the VALUES clause for each column in the column list.

subquery

is a subquery that returns rows that are inserted into the table. The select list of this subquery must have the same number of columns as the column list of the `INSERT` statement.

E.18.5 Usage Notes

Any host variables that appear in the `WHERE` clause must be either all scalars or all arrays. If they are scalars, Oracle executes the `INSERT` statement once. If they are arrays, Oracle 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 Oracle executes the statement is determined by the smaller of the following values:

- size of the smallest array
- the value of the *:host_integer* in the optional FOR clause.

For more information on this command, see Chapter 4, " Using Embedded SQL".

E.18.6 Example I

This example illustrates the use of the embedded SQL `INSERT` command:

```
EXEC SQL
 INSERT INTO emp (ename, empno, sal)
 VALUES (:ename, :empno, :sal);
```

E.18.7 Example II

This example shows an embedded SQL `INSERT` command with a subquery:

```
EXEC SQL
 INSERT INTO new_emp (ename, empno, sal)
 SELECT ename, empno, sal FROM emp
 WHERE deptno = :deptno;
```

E.18.8 Related Topics

"DECLARE DATABASE (Oracle Embedded SQL Directive)"

E.19 OPEN (Executable Embedded SQL)

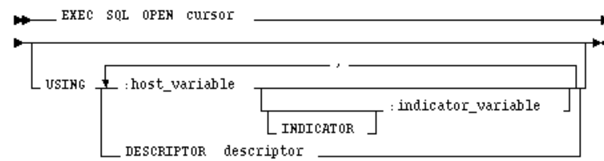
E.19.1 Purpose

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.

E.19.2 Prerequisites

You must declare the cursor with a `DECLARE CURSOR` embedded SQL statement before opening it.

E.19.3 Syntax



E.19.4 Keywords and Parameters

cursor

is the cursor to be opened.

USING

specifies the host variables to be substituted into the WHERE clause of the associated query.

:*host_variable* specifies a host variable with an optional indicator variable to be substituted into the statement associated with the cursor.

DESCRIPTOR

specifies a descriptor that describes the host variables to be substituted into the WHERE clause of the associated query. The *descriptor* must be initialized in a previous DESCRIBE statement.

The substitution is based on position. The host variable names specified in this statement can be different from the variable names in the associated query.

E.19.5 Usage Notes

The OPEN command defines the active set of rows and initializes the cursor just before the first row of the active set. The values of the host variables at the time of the OPEN are substituted in the statement. This command does not actually retrieve rows; rows are retrieved by the FETCH command.

After you have opened a cursor, its input host variables are not reexamined until you reopen the cursor. To change any input host variables and therefore the active set, you must reopen the cursor.

All cursors in a program are in a closed state when the program is initiated or when they have been explicitly closed using the CLOSE command.

You can reopen a cursor without first closing it. For more information on this command, see Chapter 4, " Using Embedded SQL".

E.19.6 Example

This example illustrates the use of the OPEN command in a Pro*C/C++ embedded SQL program:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ename, empno, job, sal
 FROM emp
 WHERE deptno = :deptno;
EXEC SQL OPEN emp_cursor;
```

E.19.7 Related Topics

"CLOSE (Executable Embedded SQL)". "DECLARE CURSOR (Embedded SQL Directive)", "FETCH (Executable Embedded SQL)", and "PREPARE (Executable Embedded SQL)"

E.20 PREPARE (Executable Embedded SQL)

E.20.1 Purpose

To parse a SQL statement or PL/SQL block specified by a host variable and associate it with an identifier.

E.20.2 Prerequisites

None.

E.20.3 Syntax

```
EXEC SQL PREPARE statement_id FROM :host_string
                                   'text'
```

E.20.4 Keywords and Parameters

statement_id

is the identifier to be associated with the prepared SQL statement or PL/SQL block. If this identifier was previously assigned to another statement or block, the prior assignment is superseded.

:host_string

is a host variable whose value is the text of a SQL statement or PL/SQL block to be prepared.

text

is a string literal containing a SQL statement or PL/SQL block to be prepared.

E.20.5 Usage Notes

Any variables that appear in the *:host_string* or *text* are placeholders. The actual host variable names are assigned in the `USING` clause of the `OPEN` command (input host variables) or in the `INTO` clause of the `FETCH` command (output host variables).

A SQL statement is prepared only once, but can be executed any number of times.

E.20.6 Example

This example illustrates the use of a `PREPARE` statement in a Pro*C/C++ embedded SQL program:

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL EXECUTE my_statement;
```

E.20.7 Related Topics

"CLOSE (Executable Embedded SQL)", "DECLARE CURSOR (Embedded SQL Directive)", "FETCH (Executable Embedded SQL)", and "OPEN (Executable Embedded SQL)"

E.21 ROLLBACK (Executable Embedded SQL)

E.21.1 Purpose

To undo work done in the current transaction.

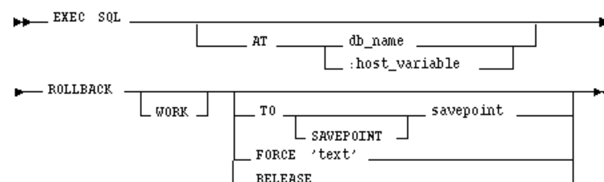
You can also use this command to manually undo the work done by an in-doubt distributed transaction.

E.21.2 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.

E.21.3 Syntax



E.21.4 Keywords and Parameters

WORK

is optional and is provided for ANSI compatibility.

TO

rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.

FORCE

manually rolls back an in-doubt distributed transaction. The transaction is identified by the *text* containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING.

ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

RELEASE

frees all resources and disconnects the application from the Oracle Server. The RELEASE clause is not allowed with SAVEPOINT and FORCE clauses.

E.21.5 Usage Notes

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle 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 Oracle issues an implicit COMMIT statement before and after processing any data definition language statement.

Using the ROLLBACK command without the TO SAVEPOINT clause performs the following operations:

- ends the transaction
- undoes all changes in the current transaction
- erases all savepoints in the transaction
- releases the transaction's locks

Using the ROLLBACK command with the TO SAVEPOINT clause performs the following operations:

- rolls back just the portion of the transaction after the savepoint.
- loses all savepoints created after that savepoint. Note that the named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- releases all table and row locks acquired since the savepoint. Note that other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

It is recommended that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit the transaction and the program terminates abnormally, Oracle rolls back the last uncommitted transaction.

E.21.6 Example I

The following statement rolls back your entire current transaction:

```
EXEC SQL ROLLBACK;
```

E.21.7 Example II

The following statement rolls back your current transaction to savepoint SP5:

```
EXEC SQL ROLLBACK TO SAVEPOINT sp5;
```

E.21.8 Distributed Transactions

Oracle with the distributed option enables 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.

E.21.9 Example III

The following statement manually rolls back an in-doubt distributed transaction:

```
EXEC SQL
 ROLLBACK WORK
 FORCE '25.32.87';
```

E.21.10 Related Topics

"COMMIT (Executable Embedded SQL)" and "SAVEPOINT (Executable Embedded SQL)"

E.22 SAVEPOINT (Executable Embedded SQL)

E.22.1 Purpose

To identify a point in a transaction to which you can later roll back.

E.22.2 Prerequisites

None.

E.22.3 Syntax

```
>>-- EXEC SQL --+-----------------------------+-- SAVEPOINT  savepoint -><
                +- AT --+- db_name --------+--+
                        +- :host_variable -+
```

E.22.4 Keywords and Parameters

AT

identifies the database on which the savepoint is created. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the savepoint is created on your default database.

savepoint

is the name of the savepoint to be created.

E.22.5 Usage Notes

For more information on this command, see Chapter 7, "Defining and Controlling Transactions".

Example

This example illustrates the use of the embedded SQL SAVEPOINT command:

```
EXEC SQL SAVEPOINT save3;
```

E.22.6 Related Topics

"COMMIT (Executable Embedded SQL)" and "ROLLBACK (Executable Embedded SQL)"

E.23 SELECT (Executable Embedded SQL)

E.23.1 Purpose

To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.

E.23.2 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 READ or SELECT privilege on the table or snapshot.

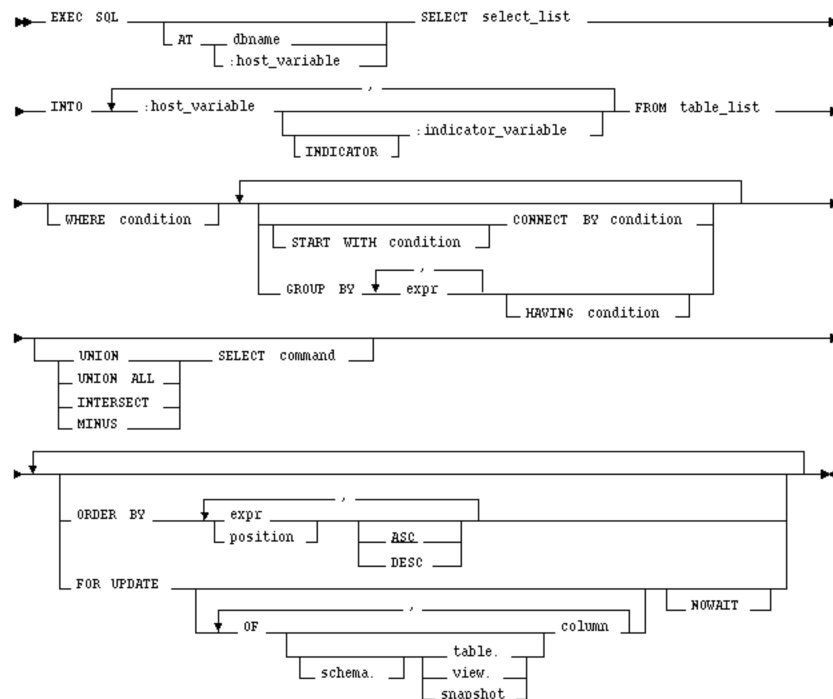
For you to select rows from the base tables of a view, the owner of the schema containing the view must have READ or SELECT privilege on the base tables. Also, if the view is in a schema other than your own, you must have READ or SELECT privilege on the view.

The READ ANY TABLE or SELECT ANY TABLE system privilege also enables select data from any table or any snapshot or any view's base table.

If you are using Oracle in DBMS MAC mode, your DBMS label must dominate the creation label of each queried table, view, or snapshot or you must have READUP system privileges.

The READ privilege cannot be used for SELECT ... FOR UPDATE operations.

E.23.3 Syntax



E.23.4 Keywords and Parameters

AT

identifies the database to which the SELECT statement is issued. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the SELECT statement is issued to your default database.

select_list

identical to the non-embedded SELECT command except that a host variables can be used in place of literals.

INTO

specifies output host variables and optional indicator variables to receive the data returned by the SELECT statement. Note that these variables must be either all scalars or all arrays, but arrays need not have the same size.

WHERE

restricts the rows returned to those for which the condition is TRUE. 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 command.

E.23.5 Usage Notes

If no rows meet the WHERE clause condition, no rows are retrieved and Oracle 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 Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle Database Performance Tuning Guide*.

E.23.6 Example

This example illustrates the use of the embedded SQL SELECT command:

```
EXEC SQL SELECT ename, sal + 100, job
 INTO :ename, :sal, :job
 FROM emp
 WHERE empno = :empno
```

E.23.7 Related Topics

"DECLARE CURSOR (Embedded SQL Directive)", "DECLARE DATABASE (Oracle Embedded SQL Directive)", "EXECUTE (Executable Embedded SQL)", "FETCH (Executable Embedded SQL)", and "PREPARE (Executable Embedded SQL)"

E.24 UPDATE (Executable Embedded SQL)

E.24.1 Purpose

To change existing values in a table or in a view's base table.

E.24.2 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.

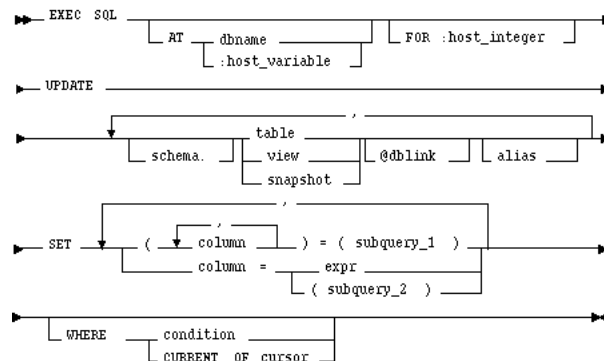
For you to update values in the base table of a view, the owner of the schema containing the view must have UPDATE privilege on the base table. Also, if the view is in a schema other than your own, you must have UPDATE privilege on the view.

The UPDATE ANY TABLE system privilege also enables update values in any table or any view's base table.

If you are using Oracle in DBMS MAC mode, your DBMS label must match the creation label of the table or view:

- If the creation label of the table or view is higher than your DBMS label, you must have READUP and WRITEUP system privileges
- If the creation label of the table or view is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the creation label of your table or view is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

E.24.3 Syntax



E.24.4 Keywords and Parameters

AT

identifies the database to which the UPDATE statement is issued. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the 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, Oracle executes the statement once for each component of the smallest array.

schema

is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table view

is the name of the table to be updated. If you specify *view*, Oracle updates the view's base table.

dblink

is a complete or partial name of a database link to a remote database where the table or view is located. You can only use a database link to update a remote table or view if you are using Oracle with the distributed option.

alias

is a name used to reference the table, view, or subquery elsewhere in the statement.

column

is the name of a column of the table or view that is to be updated. If you omit a column of the table from the SET clause, that column's value remains unchanged.

expr

is the new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables.

subquery_1

is a subquery that returns new values that are assigned to the corresponding columns.

subquery_2

is a subquery that return a new value that is assigned to the corresponding column.

`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.

`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, Oracle updates all rows of the table or view.

E.24.5 Usage Notes

Host variables in the `SET` and `WHERE` clauses must be either all scalars or all arrays. If they are scalars, Oracle executes the `UPDATE` statement only once. If they are arrays, Oracle 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 Oracle 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 Oracle returns an error message through the `SQLCODE` element of the SQLCA. If you omit the `WHERE` clause, all rows are updated and Oracle 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 Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement.

For more information on this command, see Chapter 4, " Using Embedded SQL" and Chapter 7, "Defining and Controlling Transactions".

E.24.6 Examples

The following examples illustrate the use of the embedded SQL `UPDATE` command:

```
EXEC SQL UPDATE emp
 SET sal = :sal, comm = :comm INDICATOR :comm_ind
 WHERE ename = :ename;

EXEC SQL UPDATE emp
 SET (sal, comm) =
 (SELECT AVG(sal)*1.1, AVG(comm)*1.1
 FROM emp)
 WHERE ename = 'JONES';
```

E.24.7 Related Topics

"DECLARE DATABASE (Oracle Embedded SQL Directive)"

E.25 VAR (Oracle Embedded SQL Directive)

E.25.1 Purpose

To perform *host variable equivalencing*, or to assign a specific Oracle external datatype to an individual host variable, overriding the default datatype assignment.

E.25.2 Prerequisites

The host variable must be previously declared in the Declare Section of the embedded SQL program.

E.25.3 Syntax

```
►►— EXEC  SQL  VAR  host_variable   IS  datatype  ——————————————————►◄
```

E.25.4 Keywords and Parameters

host_variable

is the host variable to be assigned an Oracle external datatype.

datatype

is an Oracle external datatype recognized by the Oracle Precompilers (not an Oracle 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 Chapter 3, "Meeting Program Requirements".

E.25.5 Usage Notes

Host variable equivalencing is one kind of datatype equivalencing. Datatype equivalencing is useful for any of the following purposes:

- to automatically null-terminate a character host variable
- to store program data as binary data in the database
- to override default datatype conversion

E.25.6 Example

This example equivalences the host variable `DEPT_NAME` to

the datatype `STRING` and the host variable `BUFFER` to the datatype `RAW(2000)`:

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 dept_name CHARACTER(15); -- default datatype is CHAR
 EXEC SQL VAR dept_name IS STRING; -- reset to STRING
 ...
 buffer CHARACTER(200); -- default datatype is CHAR
 EXEC SQL VAR buffer IS RAW(200); -- refer to RAW
 ...
```

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

E.25.7 Related Topics

None.

E.26 WHENEVER (Embedded SQL Directive)

E.26.1 Purpose

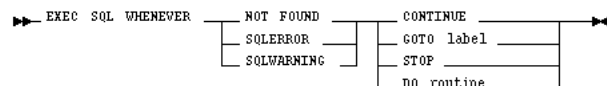
To specify the action to be taken when an error or warning results from executing an embedded SQL program.

E.26.2 Prerequisites

None.

E.26.3 Syntax

The following syntax diagram shows how to construct a `WHENEVER` statement:



E.26.4 Keywords and Parameters

`NOT FOUND`

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`

indicates that the program should branch to the statement named by *label*.

`STOP`

stops program execution.

`DO`

indicates that the program should call a host language routine. The syntax of *routine* depends on your host language. See your language-specific Supplement to the Oracle Precompilers Guide.

E.26.5 Usage Notes

The WHENEVER command allows your program to transfer control to an error handling routine in the event an embedded SQL statement results in an error or warning.

The scope of a WHENEVER statement is positional, rather than logical. A WHENEVER statement applies to all embedded SQL statements that textually follow it in the source file, not in the flow of the program logic. A WHENEVER statement remains in effect until it is superseded by another WHENEVER statement checking for the same condition.

For more information on this command, see Chapter 7, "Defining and Controlling Transactions". Do not confuse the WHENEVER embedded SQL command with the WHENEVER SQL*Plus command.

E.26.6 Example

The following example illustrates the use of the WHENEVER command in a Pro*C/C++ embedded SQL program:

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...
EXEC SQL WHENEVER SQLERROR GOTO sql_error:
...
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK RELEASE;
```

E.26.7 Related Topics

None.

Index

A

abnormal termination, automatic rollback, E-8
active set, 4-7
  changing, 4-8, 4-9
ALLOCATE command, E-5
allocating, cursors, E-6
ANSI/ISO SQL
  compliance, 1-4
  extensions, 6-24
application development process, 2-7
array, 9-1
array fetch, 9-3
array, elements, 9-2
array, operations, 2-5
ARRAYLEN statement, 5-11
ASACC option, 6-11
ASSUME SQLCODE option, 6-11
AT clause
  CONNECT statement, 3-29
  DECLARE CURSOR statement, 3-29
  DECLARE STATEMENT statement, 3-30
  EXECUTE IMMEDIATE statement, 3-30
  of COMMITcommand, E-8
  of DECLARE CURSOR command, E-8
  of DECLARE STATEMENT command, E-13
  of EXECUTE command, E-20
  of EXECUTE IMMEDIATE command, E-22
  of INSERTcommand, E-25
  of SAVEPOINT command, E-32
  of UPDATE command, E-35
  restrictions, 3-29
AUTO_CONNECToption, 6-11
automatic logon, 3-27

B

batch fetch, 9-3
  example, 9-3
  number of rows returned, 9-3
bind descriptor, information in, 10-12
bind variable, 4-13, 10-12
binding, 10-3
blank padding, in multi-byte character strings, 3-25
block data subprogram, used by precompiler, 6-15

C

callback, user exit, 11-10
CHAR column, maximum width, 3-4
CHAR datatype
  external, 3-9
  internal, 3-4
CHAR_MAP precompiler option, 6-12
character strings, multi-byte, 3-24
CHARF data type specifier
  using in TYPE statement, 3-22
  using in VAR statement, 3-22
CHARF data type, external, 3-9
CHARF datat type specifier, 3-22
CHARZ data type, 3-10
chatracter sets, multi byte, 3-24
child cursor, 5-12
CINCR precompiler option, 6-12
CLOSE command, E-6
  examples, E-7
CLOSE statement, 4-9, 4-14
  example, 4-9
CLOSE_ON_COMMIT
  precompiler option, 6-13
closing, cursors, E-6
CMAX precompiler option, 6-13
CMIN precompiler option, 6-14
CNOWAIT precompiler option, 6-14
code page, 3-24
CODE precompiler option, 6-14
column, ROWLABEL, 3-8
comment, 10-16
COMMENT clause, of COMMIT command, E-8
commit, 7-2
  automatic, 7-3
  explicit versus implicit, 7-2
COMMIT command, E-7
  ending a transaction, E-7
  examples, E-9
COMMIT statement, 7-3
  effects, 7-3
  example, 7-3
  RELEASE option, 7-3
  using PL/SQL block, 7-10
  where to place, 7-3
commiting, transactions, E-7

COMMON NAME option, 6-15
COMMON_PARSER precompiler option, 6-16
communication over a network, 3-27
COMP_CHARSET precompiler option, 6-16, 6-17
compilation, 6-49
compliance, ANSI/ISO, 1-5
concurency, 7-1
concurrent logons, 3-26
conditional precompilation, 6-47
  defining symbols, 6-48
  example, 6-47
CONFIG option, 6-6
CONFIG precompiler option, 6-18
configuration file
  system versus user, 6-6
configuration files
  advantages, 6-6
CONNECT statement
  AT clause, 3-28
  enabling a semantic check, D-2
  USING clause, 3-28
connection
  concurrent, 3-30
  default versus non-default, 3-27
  implicit, 3-31
CONTINUE action, 8-23
CONTINUE option, of WHENEVER statement, E-38
CPOOL precompiler option, 6-18
CPP_SUFFIX precompiler option, 6-18
CPP_SUFFIX precompiler options, 6-18
CREATE PROCEDURE statement, 5-13
creating, savepoints, E-31
CTIMEOUT precompiler option, 6-19
CURRENT of cluase, 4-10
  example, 4-10
  mimicking with ROWID, 9-9
  of embedded SQL DELETE command, E-16
  of embedded SQL UPDATE command, E-36
  restrictions, 4-10
current row, 2-6
CURRVAL psuedocolumn, 3-7
cursor, 4-7
  allocating, E-5
  association with query, 4-7
  child, 5-12
  closing, E-6
  declaring, 4-8
  effects on performance, C-5
  explicit versus implicit, 2-5
  naming, 4-8
  parent, 5-12
  reopening, 4-8, 4-9
  restricted scope of, 6-49
  restrictions, 6-49
  using for multiple row query, 4-7
  using more than one, 4-8
cursor cache, 5-12, 8-30, E-5
  gathering stastics about, 8-32
cursor chache
  purpose, C-6
cursor varaible
  opening, 4-13
cursor variable
  closing, 4-14
  fetching from, 4-14
cursor, scope, 4-8
cursors
  fetching rows from, E-22
  opening, E-26

D

data definition language, 4-2
data definition language (DDL)
  description, 4-2
data integrity, 7-1
data manipulation language (DML), 4-4
data type
  host-languages, 3-17
  internal versus external, 2-5
  user defined, 3-17
data type conversion, 3-14
  between internal and external datatypes, 3-15
data type equivalencing, 3-19
  advantages, 3-19
  example, 3-20
  guidelines, 3-22
database link
  defining, 3-32
  using in DELETE command, E-16
  using in delete command, E-16
  using in UPDATE command, E-35
database links
  creating a synonym, 3-32
Date data type
  converting, 3-16
  default format, 3-16
  default value, 3-5
  external, 3-5
  internal, 3-5
  internal format, 3-5
DB2_ARRAY precompiler option, 6-19
DBMS option, 6-20
deadlock, 7-2
  breaking, 7-5
DECIMAL data type, 3-10
declarartion
  host array, 9-2
  host variable, 3-17
declaration
  cursor, 4-8
  indicator variable, 3-18
  of ORACA, 8-29
  SQLCA, 8-4
Declarative SQL statements, 2-2
declarative SQL statements
  using in transactions, 7-2
declare CURSOR command, E-10
  examples, E-11
declare CURSOR statement

, 3-29
declare DATABASE directive, E-11
declare section, 3-1
  example, 3-2
  using more than one, 3-2
DECLARE statement
  example, 4-8
  using in dynamic SQL method 3, 10-10
  where to place, 4-8
declare STATEMENT command, E-12
  example, E-13
  scope of, E-13
DECLARE STATEMENT statement
  AT clause, 3-30
  example, 10-14
  using in dynamic SQL, 10-10
  when required, 10-14
DECLARE TABLE command, E-14
  example, E-15
DECLARE TABLE statement
  need for with AT clause, 3-29
  using with the SQL CHECK option, D-3
declare TABLE statement
  need for with AT clause, 3-29
DEF_SQLCODE precompiler option, 6-21
default connection, 3-27
default database, 3-27
default, setting of LITDELIM option, 6-30
DEFINE option, 6-21
definition, 2-6
delete cascaade, 8-19
DELETE command, E-15
  embedded SQL examples, E-17
DELETE statement
  using SQLERRD(3) filed, 9-10
  WHERE clause, 4-7
DEPT table, 2-8
DESCRIBE command, E-18
  example, E-18
DESCRIBE statement, using in dynamic SQL Method 4, 10-12
directory, 3-2
  current, 3-2
  path for INCLUDE files, 3-2
DISPLAY data type, 3-10
distributed procesing, 3-26
DO action, 8-23
DO option, of WHENEVER command, E-38
DTP model, 3-34
dummy host variables, 10-3
DURATION precompiler option, 6-22
dynamic PL/SQL, 10-15
dynamic SQL
  advantages and disadvantages, 10-2
  choosing the right method, 10-4
  guidelines, 10-4
  overview, 10-1
  using PL/SQL, 10-15
  when useful, 10-2
dynamic SQL Method 1
  command, 10-6
  dexcription, 10-6
  example, 10-6
  requirements, 10-6
dynamic SQL Method 2
  commands, 10-4
  description, 10-7
  example, 10-8
  requirements, 10-7
  using PL/SQL, 10-15
  using the EXECUTE statement, 10-4
  using the PREPARE statement, 10-4
dynamic SQL method 2
  using the DECLARE STATEMENT Statement, 10-14
dynamic SQL Method 3
  compared to method 2, 10-9
  description, 10-9
  example, 10-11
  requirements, 10-9
  using PL/SQL, 10-15
  using the CLOSE statement, 10-10
  using the DECLARE statement, 10-10
  using the FETCH statement, 10-10
  using the OPEN statement, 10-10
  using the PREPARE statement, 10-9
dynamic SQL method 3
  using the DECLARE STATEMENT Statement, 10-14
dynamic SQL Method 4
  overview, 10-11
  using descriptors, 10-11
  using SQLDA, 10-12
  using the DESCRIBE statement, 10-12
  when needed, 10-11
dynamic SQL method 4
  using the DECLARE STATEMENT Statement, 10-14
dynamic SQL Method1
  using EXECUTE IMMEDIATE, 10-6
  using PL/SQL, 10-15
dynamic SQL Method4
  using PL/SQL, 10-15
dynamic SQL statement, 10-1
  binding of host variables, 10-3
  how processed, 10-3
  requirements, 10-2
  using host arrays, 10-14
  using placeholders, 10-3

E

embedded PL/SQL
  advantages, 5-1
  cursor for loops, 5-2
  example, 5-5, 5-6
  need for SQL check option, 5-5
  need for USERID check option, 5-5
  packages, 5-3
  PL/SQL table, 5-3

requirements, 5-5
subprograms, 5-2
user-defines record, 5-4
using %TYPE, 5-2
where allowed, 5-4
embedded SQL
ALLOCATE command, E-5
CLOSE command, E-6
COMMIT command, E-7
CONNECTcommand, E-9
DECLARE cursor command, E-10
DECLARE CURSORcommand, E-10
DECLARE DATABASE command, E-11
DECLARE STATEMENT command, E-12
DECLARE TABLE command, E-14
DELETE command, E-15
DESCRIBE command, E-17
EXECUTE command, E-20
EXECUTE IMMEDIATE command, E-21
EXECUTEcommand, E-19
FETCH command, E-22
INSERT command, E-24
mixing with host-language statement, 2-4
OPEN command, E-26, E-28
referencing indicator variables, 3-18
SAVEPOINT command, E-31
SELECT command, E-32
UPDATE command, E-34
VAR command, E-37
versus interactive SQL, 2-4
WHENEVER command, E-38
embedded SQL statement
referencing host-langauage varaibles, 3-17
syntax, 2-4
embedding PL/SQL blocks in Oracle 7 precompiler programs, E-19
EMP table, 2-8
encoding scheme, 3-24
equivalencing, data type, 3-19
error detection, error reporting, E-39
error handling
alternatives, 8-1
benefits, 8-1
error handling
using the SQLCODE status variable, 8-4
overview, 2-6
SQLCA versus WHENEVER statement, 8-2
SQLCODE status variable, 8-3
using SQLCA, 8-15
using the ORACA structure, 8-28
using the ROLLBACK statement, 7-4
using the SQLGLM function, 8-21, 8-26
using the WHENEVER statement, 8-22
error message
available in SQLCA, 8-18
maximum length, 8-21
using in error reporting, 8-18
using the SQLGLM function, 8-21
error reporting
key components, 8-17
using error messages, 8-17
using status codes, 8-17
using the parse error offset, 8-17
using the rows-processed count, 8-17
using the WHENEVER command, 8-22
using warning flags, 8-17
errors options, 6-23
ERRTYPE
precompiler option, 6-23
exception, PL/SQL, 5-9
EXEC ORACLE DEFINE statement, 6-47
EXEC ORACLE ELSE statement, 6-47
EXEC ORACLE ENDIF statement, 6-47
EXEC ORACLE IFDEF statement, 6-47
EXEC ORACLE IFNDEF statement, 6-47
EXEC ORACLE statement
inline, 6-5
scope of, 6-5
syntax for, 6-5
EXEC SQL clause, 2-4
EXEC TOOLS statements, 11-9
GET, 11-10
MESSAGE, 11-11
SET, 11-10
SET CONTEXT, 11-10, 11-11
executable SQL statement, 2-2, E-19
example, E-20
EXECUTE IMMEDIATE command, E-21
example, E-22
EXECUTE IMMEDIATE statement
AT clause, 3-29
EXECUTE statement, using in dynamic SQL Method 2, 10-7
EXPLAIN PLAN statement, using to improve performance, C-4
Explicit, 3-30
explicit logon, 3-27
multiple, 3-30
single, 3-28
external datatype, 3-8
CHAR, 3-9
CHARF, 3-9
CHARZ, 3-10
DATE, 3-10
DECIMAL, 3-10
DISPLAY, 3-10
FLOAT, 3-10
INTEGER, 3-10, 3-11
LONG, 3-11
LONG VARCHAR, 3-11
LONG VARRAW, 3-11
MLSLABEL, 3-11
NUMBER, 3-12
RAW, 3-12
ROWID, 3-12
STRING, 3-13
UNSIGNED, 3-13
VARCHAR, 3-13
VARCHAR2, 3-13
VARNUM, 3-14

F

features, new, A-1
FETCH command, E-22
  examples, E-24
  used after OPEN command, E-27
FETCH statement, 4-14
  example, 4-14
  INTO clause, 4-14
  using the SQERRD(3), 9-10
fetch, batch, 9-3
fetching, rows from cursors, E-22
FIPS option, 6-24
flag, warning, 8-17
FLOAT datatypes, 3-10
FOR clause, 9-7
  example, 9-7, E-20
  of embedded SQL INSERT command, E-25
  restrictions, 9-8
  using with HOST arrays, 9-8
FOR UPDATE clause, 4-13
FOR UPDATE OF clause, 7-8
FORCE clause
  of COMMIT command, E-8
  of ROLLBACK command, E-29
format mask, 3-16
FORMAT option, 6-24
forward reference, 4-8
full scan, C-4
function prototype
  definition of, 6-14

G

GENXTB form, running, 11-7
globalizaton support, 3-22
  multibyte charecter strings, 3-24
globalizaton support parameter
  currency, 3-23
  DATE FORMAT, 3-23
  DATE LANGUAGE, 3-23
  ISO CURRENCY, 3-23
  LANGUAGE, 3-23
  NUMERIC CHARACTERS, 3-23
  SORT, 3-23
  TERRITORY, 3-23
GOTO action, 8-23
GOTO optio, of WHENEVER command, E-38
guidelines
  datatype equivalencing, 3-22
  dynamic SQL, 10-4
  host variable, 3-18
  separate precompilation, 6-48
  transactions, 7-9
  user exit, 11-8
  WHENEVER statement, 8-24
guielines
  indicator variables, 3-19

H

HEADER precompiler option, 6-25
heap, 8-30
hint, optimizer, C-3
hints
  in DELETE statements, E-17
  in SELECTstatement, E-34
  in UPDATEstatement, E-36
HOLD CURSOR option
  of Oracle precompilers, E-6
HOLD_CURSOR precompiler option, 6-26
host array, 9-1
  advantages, 9-2
  declaring, 9-2
  dimensions, 9-2
  maximum size, 9-2
  referencing, 9-2
  restrictions, 9-4, 9-5, 9-6
  using dynamic SQL statement, 10-14
  using in the DELETE statement, 9-6
  using in the FOR clause, 9-7
  using in the INSERT statement, 9-5
  using in the SELECTstatement, 9-3
  using in the UPDATE statement, 9-5
  using in the WHERE clause, 9-9
  using to improve performance, C-2
  when not allowed, 9-2
host language, 2-2
host option, 6-27
host program, 2-2
host varaible
  in OPEN command, E-27
  multi-byte charecter strings, 3-25
  undeclare, 3-1
  using in EXEC TOOLS statement, 11-9
  using in PL/SQL, 5-5
host variable, 4-1
  assigning a value, 2-4
  declaring, 3-16
  dummy, 10-3
  host variable equivalencing, E-37
  in EXECUTE command, E-21
  in OPEN command, E-26
  output versus input, 4-1
  overview, 2-4
host variables
  using in user exit, 11-3
  where allowed, 2-4
host-language datatype, 3-17

I

IAF GET statement
  example, 11-4
  specifying block and field names, 11-4
  using user exit, 11-4
IAF PUT statement
  example, 11-5
  specifying block and filed names, 11-5
  using user exit, 11-4

IAP, 11-8
implicit logon, 3-31
implicit logons
  multiple, 3-32
  single, 3-32
IMPLICIT_SVPT precompiler option, 6-27
in doubt transaction, 7-9
IN OUT parameter modes, 5-3
IN parameter mode, 5-3
INAME option, 6-27
  when a file extension is required, 6-2
INCLUDE file, 3-2
INCLUDE option, 6-28
INCLUDE statement, 3-2
  using to declare the ORACA, 8-29
  using to declare the SQLCA, 8-16
index, using to improve performance, C-4
indiacator array, 9-1
indicator varaible, 4-2
indicator variable
  guidelines, 3-19
  referencing, 3-18
indicator variables
  used to detetc truncated values, 4-2
  used with multi-byte charecter strings, 3-26
  using in PL/SQL, 5-8
  using to handle nulls, 4-2, 4-3
  using to test for nulls, 4-4
input host variable
  restrictions, 4-2
  where allowed, 4-1
INSERT command, E-24
  embedded SQL examples, E-26
INSERT of no rows, 8-19
  cause of, 8-8
INSERT statement, 4-6
  column list, 4-6
  example, 4-6
  INTO clause, 4-6
  using SQLERRD(3), 9-10
inserting, rows into tables and views, E-24
INTEGER datatype, 3-11
interface
  native, 3-34
  XA, 3-34
internal datatypes
  , 3-4
  CHAR, 3-4
  DATE, 3-5
  definition, 3-3
  LONG, 3-5
  LONG RAW, 3-5
  MLSLABEL, 3-5
  NUMBER, 3-5
  RAW, 3-6
  ROWID, 3-6
  VARCHAR2, 3-6
INTO clause, 4-1, 4-14
  FETCH statement, 4-9
  INSERT statement, 4-6
  of FETCH command, E-23
  of SELECT statement, E-33
  SELECT statement, 4-5
INTYPE precompiler option, 6-29
IRECLEN option, 6-29

J

julian date, 3-5

K

keywords, B-2

L

language support, 1-2
LDA, 3-33
LEVEL pseudocolumn, 3-7
LINES precompiler option, 6-29
link, database, 3-32
linking, 6-49
LITDELIM option, 6-30
  purpose, 6-30
LNAME option, 6-31
location transparency, 3-32
LOCK TABLE statement, 7-8
  example, 7-8
  using the NOWAIT parameter, 7-8
lock, released by ROLLBACK statement, E-30
locking, 7-1, 7-7
  explicit versus implicit, 7-7
  modes, 7-1
  privileges needed, 7-10
  using the FOR UPDATE of clause, 7-7
  using the LOCK TABLE statement, 7-8
logon
  concurrent, 3-26
  explicit, 3-27
Logon Data Area (LDA), 3-33
LONG datatype
  comapred with CHAR, 3-5
  external, 3-11
  internal, 3-4
  restriction, 3-5
LONG RAW column, maximum width, 3-5
LONG RAW datatype
  compared with LONG, 3-5
  conversion, 3-16
  external, 3-5
  internal, 3-11
LONG VAR CHAR datatype, 3-11
LONG VARRAW datatype, 3-11
LRECLEN option, 6-31
LTYPE option, 6-31

M

MAX_ROW_INSERT precompiler option, 6-33
MAXLITERAL option, 6-32
MAXOPENCURSORS option, 6-32

using for separate precompilation, 6-48
what it affects, C-5
Meeting, 3-1
MLSLABEL data type, 3-5
MODE option, 6-33
effect on OPEN, 4-8
mode, parameter, 5-3
monitor, transaction processing, 3-34
multi-byte character sets, 3-25
MULTISUBPROG option, 6-34

N

namespaces, reserved by Oracle, B-5
naming conventions
cursor, 4-8
SQL* Forms user exit, 11-8
naming of database objects, E-5
NATIVE
value of DBMS option, 6-19
native interface, 3-34
NATIVE_TYPES precompiler option, 6-35
network
communicating over, 3-27
protocol, 3-27
reducing network traffic, C-3
NEXTVAL, psuedocolumn, 3-7
nibble, 3-16
NIST, compliance, 1-5
NLS_CHAR precompiler option, 6-35
NLS_LOCAL precompiler option, 6-36
node, definition, 3-27
NOT FOUND condition
WHENEVER clause, E-38
NOWAIT
parameter, 7-8
using the LOCK TABLE statement, 7-8
null
definition, 2-5
detecting, 4-2
hardcode, 4-3
inserting, 4-3
restrictions, 4-4
retrieving, 4-3
testing for, 4-4
null-terminated string, 3-13
NUMBER data type
external, 3-12
internal, 3-5

O

OBJECTS precompiler option, 6-23, 6-36
OCI
declaring LDA, 3-33
embedding calls, 3-33
ONAME option, 6-36
OPEN command, E-26
examples, E-27
OPEN statement, 4-8
example, 4-8
using in dynamic SQL Method 3, 10-10
OPEN_CURSORS parameter, 5-12
OPEN-FOR statement, 4-13
opening, cursors, E-26
optimizer hint, C-3
options, precompiler, 6-2
ORACA, 8-28
declaring, 8-29
enabling, 8-29
example, 8-33
fields, 8-30
gathering cursor cache statistics, 8-32
ORACABC field, 8-30
ORACAID field, 8-30
ORACCHF flag, 8-30
ORACOC field, 8-32
ORADBGF flag, 8-31
ORAHCHF flag, 8-31
ORAHOC field, 8-32
ORAMOC field, 8-32
ORANEX field, 8-33
ORANOR field, 8-33
ORANPR field, 8-33
ORASFNMC field, 8-32
ORASFNML field, 8-32
ORASLNR field, 8-32
ORASTXTC field, 8-32
ORASTXTF flag, 8-31
ORASTXTL field, 8-32
using more than one, 8-29
ORACA option, 6-37
ORACABC field, 8-30
ORACAID field, 8-30
ORACCHF flag, 8-30
Oracle Call Interface, 3-33
Oracle Communications Area, 8-28
Oracle datatypes, 2-5
Oracle Forms, using the EXEC TOOLS statements, 11-9
Oracle identifier, how to form, E-5
Oracle indentifiers, how to form, E-5
Oracle keywords, B-2
Oracle namespaces, B-5
Oracle Open Gateway, using ROWID datatype, 3-13
Oracle Precompilers
advantages, 1-2
function, 1-2
globalization support, 3-24
language support, 1-2
new features, A-1
running, 6-1
using PL/SQL, 5-5
using with OCI, 3-33
Oracle reserved words, B-1
Oracle Toolset, 11-9
ORACOC field, 8-32
ORADBGF flag, 8-31
ORAHCHF flag, 8-31
ORAHOC field, 8-32

ORAMOC field, 8-32
ORANEX field, 8-33
ORANOR field, 8-33
ORANPR field, 8-33
ORASFNMC field, 8-32
ORASFNML field, 8-32
ORASLNR field, 8-32
ORASTXTC field, 8-32
ORASTXTF flag, 8-31
ORASTXTL field, 8-32
ORECLEN option, 6-37
OUT parameter mode, 5-3
OUTLINE precompiler option, 6-38
OUTLNPREFIX precompiler option, 6-38
output host variable, 4-1

P

PAGELEN option, 6-39
parameter modes, 5-3
parent cursor, 5-12
PARSE
  precompiler option, 6-39
parse, 10-3
parse error offset, 8-17
parsing dynamic statements, PREPARE command, E-28
performance
  improving, C-2
  reasons for poor, C-1
placeholder, duplicate, 10-7
  naming, 10-8
  using in dynamic SQL statements, 10-3
plan, execution, C-3
PL/SQL, 1-3
  advantages, 1-3
  and the SQLCA, 8-21
  blocks, embedded in Oracle precompiler programs, E-19
  cursor FOR loop, 5-2
  exception, 5-9
  integrating with server, 5-2
  package, 5-3
  relationship with SQL, 1-3
  reserved words, B-3
  subprogram, 5-2
  user-defined record, 5-4
PL/SQL table, 5-3
precision, 3-5
precompilation, 6-2
  conditional, 6-47
  separate, 6-48
precompilation unit, 6-7
precompiler, 1-1
precompiler command, 6-1
  optional arguments of, 6-2
  required arguments, 6-1
precompiler directives, EXEC SQL DECLARE DATABASE, E-11
precompiler options
abbrevating name, 6-3
ASACC, 6-11
ASSUME_SQLCODE, 6-11
AUTO_CONNECT, 6-11
CHAR_MAP, 6-12
CINCR, 6-12
CLOSE_ON_COMMIT, 6-13
CMAX, 6-13
CMIN, 6-14
CNOWAIT, 6-14
CODE, 6-14
COMMON_NAME, 6-15
COMMON_PARSER, 6-16
COMP_CHARSET, 6-16, 6-17
CONFIG, 6-6, 6-18
CPOOL, 6-18
CPP_SUFFIX, 6-18
CTIMEOUT, 6-19
DB2_ARRAY, 6-19
DBMS, 6-20
DEF_SQLCODE, 6-21
DEFINE, 6-21
displaying, 6-3, 6-7
DURATION, 6-22
entering from a configuration file, 6-6
entering inline, 6-5
entering on the command line, 6-5
ERRORS, 6-23
ERRTYPE, 6-23
FIPS, 6-24
FORMAT, 6-24
Globalization Support_LOCAL, 6-25
HEADER, 6-25
HOLD_CURSOR, 6-26
HOST, 6-27
IMPLICIT_SVPT, 6-27
INAME, 6-27
INCLUDE, 6-28
INTYPE, 6-29
IRECLEN, 6-29
LINES, 6-29
LITDELIM, 6-30
LNAME, 6-31
LRECLEN, 6-31
LTYPE, 6-31
MAX_ROW_INSERT, 6-33
MAXLITERAL, 6-32
MAXOPENCURSORS, 6-32
MODE, 6-33
MULTISUBPROG, 6-34
NATIVE_TYPES, 6-35
NLS_CHAR, 6-35
NLS_LOCAL, 6-36
OBJECTS, 6-23, 6-36
ONAME, 6-36
ORACA, 6-37
ORECLEN, 6-37
OUTLINE, 6-38
OUTLNPREFIX, 6-38
PAGELEN, 6-39

PARSE, 6-39
PREFETCH, 6-40
RELEASE_CURSOR, 6-40
respecifying, 6-7
RUNOUTLINE, 6-41
scope of, 6-7
SELECT_ERROR, 6-41
specifying, 6-5
SQLCHECK, 6-42
STMT_CACHE, 6-43
syntax for, 6-5
THREADS, 6-43
TYPE_CODE, 6-44
UNSAFE_NULL, 6-44
USERID, 6-45
using, 6-10 to 6-46
VARCHAR, 6-46
VERSION, 6-46
XREF, 6-47
PREFETCH precompiler option, 6-40
PREPARE command, E-28
examples, E-28
PREPARE statement
effect on data definition statements, 10-4
using in dynamic SQL, 10-7, 10-9
private SQL area
association with cursors, 2-5
opening, 2-5
purpose, C-6
Program Global Area (PGA), 5-12
program termination, 7-6
programming language support, 1-2
pseudocolumn, 3-6
CURRVAL, 3-7
LEVEL, 3-7
NEXTVAL, 3-7
ROWID, 3-7
ROWNUM, 3-8
SYSDATE, 3-8
UID, 3-8
USER, 3-8
pseudotype, VARCHAR, 3-17

Q

query, 4-5
association with cursor, 4-7
multirow, 4-4
single-row versus multirow, 4-5

R

RAW column, maximum width, 3-6
RAW datatype
compared with CHAR, 3-6
converting, 3-16
external, 3-12
internal, 3-6
restrictions, 3-6
RAWTOHEX function, 3-16
read consistency, 7-2
READ ONLY parameter, using in SET TRANSACTION, 7-7
read-only transaction, 7-6
ending, 7-7
example, 7-7
record, user-defined, 5-4
reference
host array, 9-2
host variable, 3-16
indicator variable, 3-18
RELEASE option, 7-6
COMMIT statement, 7-3
omitting, 7-6
restrictions, 7-6
ROLLBACK statement, 7-4
RELEASE_CURSOR option, 6-40
of Oracle Precompilers, E-7
using to improve performance, C-7
what it affects, C-5
remote database, declaration of, E-11
reserved words, B-1
PL/SQL, B-3
resource manager, 3-34
retrieving rows from a table, embedded SQL, E-32
return code, 11-6
roll back
to a savepoint, E-31
to the same savepoint multiple times, E-30
rollabck
statement-level, 7-4
rollback
automatic, 7-4
purpose, 7-2
ROLLBACK command, E-29
ending a transaction, E-30
examples, E-30
rollback segment, 7-2
ROLLBACK Statement, 7-3
ROLLBACK statement
effects, 7-3
example, 7-4
RELEASE option, 7-4
TO SAVEPOINT clause, 7-5
using in a PL/SQL block, 7-10
using in error-handling routines, 7-4
where to place, 7-4
rolling back, transactions, E-29
row lock
acquiring with FOR UPDATE OF, 7-7
using to improve performance, C-4
when acquired, 7-8
when released, 7-8
ROWID datatype
external, 3-12
internal, 3-6
ROWID pseudocolumn, 3-7
using to mimic CURRENT OF, 7-8, 9-9
ROWLABEL column, 3-8
ROWNUM pseudocolumn, 3-8

rows
    fetching from cursors, E-22
    inserting into tables and views, E-24
    updating, E-34
rows-processed count, 8-19
    using in error reporting, 8-17
RUNOUTLINE precompiler option, 6-41

S

sample database table
    DEPT table, 2-8
    EMP table, 2-8
savepoint, 7-5
SAVEPOINT command, E-31
    example, E-32
SAVEPOINT statement, 7-5
    example, 7-5
savepoint, when erased, 7-6
SAVEPOINTS parameter, 7-6
savepoints, creating, E-31
scale, 3-5
    definition of, 3-20
    when negative, 3-20
scope
    of DECLARE STATEMENT command, E-13
    of precompiler options, 6-7
    of the EXEC ORACLE statement, 6-5
    WHENEVER statement, 8-24
search condition, 4-7
    using in the WHERE clause, 4-7
SELECT command, E-32
    embedded SQL examples, E-34
select descriptor, information in, 10-11
select list, 4-5
SELECT statement, 4-5
    available clauses, 4-6
    example, 4-5
    INTO clause, 4-5
    using host arrays, 9-3
    using the SQLERRD(3) field, 9-10
SELECT_ERROR option, 4-5, 6-41
semantic checking, D-1
    enabling, D-2
    using the SQLCHECK option, D-1
separate precompilation, 6-48
    guidelines, 6-48
    restrictions, 6-49
session, 7-1
sessions, beginning, E-9
SET clause, 4-6
    using a subquery, 4-6
SET TRANSACTION statement, 7-6
    example, 7-6
    READ ONLY parameter, 7-7
    restrictions, 7-7
snapshots, 7-2
SQL code, returned by SQLGLS function, 8-27
SQL Communications Area, 8-15
SQL Descriptor Area, 10-12
SQL standards conformance, 1-5
SQL statement
    controlling transactions, 7-2
    executable versus declarative, 2-2
    optimizing to improve performance, C-3
    static versus dynamic, 2-4
    using to control a cursor, 4-5, 4-7
    using to manipulate Oracle data, 4-4
SQL_CURSOR, E-6
SQL, summary of commands, E-2
SQL*Connect, using ROWID datatype, 3-12
SQL*Forms
    Display Error screen, 11-6
    IAP Constants, 11-6
    returning values to, 11-6
    Reverse Return Code switch, 11-6
    user exit, 11-2
SQL*Net
    concurrent logons, 3-26
    connection syntax, 3-27
    function of, 3-27
    using to connect to Oracle, 3-26
SQL*Plus, 1-3
SQL92
    conformance, 1-5
    deprecated feature, 8-2
    minimum requirement, 1-5
SQLCA, 3-3, 8-15
    components set for a PL/SQL block, 8-21
    declaring, 8-16
    explicit versus implicit checking, 8-2
    fields, 8-18
    interaction with Oracle, 3-3
    using in separate precompilations, 6-48
    using more than one, 8-15
    using with SQL*Net, 8-15
SQLCABC filed, 8-18
SQLCAID field, 8-18
SQLCHECK option, 6-42
    restrictions, D-1
    using the DECLARE TABLE statement, D-2
    using to check syntax, D-1
SQLCODE field, 8-18
SQLCODE status variable, 8-3
SQLCODE variable, interpreting values of, 8-8
SQLERRD, 8-19
SQLERRD(3) field, 9-10
    purpose, 8-17
    using with the FETCH statement, 9-10
SQLERRD(3) filed
    using with batch fetch, 9-4
SQLERRD(5) field, 8-20
SQLERRMC field, 8-19
SQLERRML field, 8-19
SQLERROR condition, 8-22
SQLERROR, WHENEVER command
        condition, E-38
SQLFC parameter, 8-27
SQLGLM function, 8-21
    example, 8-21

SQLGLS function
  parameters, 8-27
  restrictions, 8-27
  SQL codes returned by, 8-27
  syntax, 8-26
  using to obtain SQL text, 8-26
SQLIEM function
  replacement for, 11-9
  using in user exit, 11-6
SQLLDA routine, 3-33
SQLSTATE status variable, 8-2
  class code, 8-9
  coding scheme, 8-9
  declaring, 8-5
  error handling
    SQLSTATE status variable, 8-3
  interpreting values, 8-9
  predefined status code and conditions, 8-10
  subclass code, 8-9
SQLSTM parameter, 8-27
SQLWARN, 8-20
SQLWARN flags, 8-20
SQLWARNING condition, 8-22
SQLWARNING, WHENEVER command condition, E-38
statement-level rollback, 7-4
  breaking deadlocks, 7-5
status code, 8-17
STMLEN parameter, 8-27
STMT_CACHE
  precompiler option, 6-43
STOP action, 8-23
STOP option, of WHENEVER command, E-38
stored subprogram, 5-13
  calling, 5-15
  creating, 5-13
  packaged versus standalone, 5-13
  stored versus inline, C-3
  using to improve performance, C-2
STRING datatype, 3-13
subprogram, PL/SQL, 5-2, 5-13
subquery, 4-6
  example, 4-6
  using in the SET clause, 4-6
  using in the VALUES clause, 4-6
syntactic checking, D-1
syntax diagram
  description of, E-3
  how to read, E-3
  how to use, E-3
  symbols used in, E-3
syntax, embedded SQL, 2-4
SYSDATE function, 3-8
system failure, effect on transactions, 7-3
System Global Area (SGA), 5-13

T

table lock
  acquiring with LOCK TABLE, 7-8
  exclusive, 7-8
  row share, 7-8
  when released, 7-8
tables
  inserting rows into, E-24
  updating rows in, E-34
THREADS
  precompiler option, 6-43
TO clause, of ROLLBACK command, E-29
TO SAVEPOINT clause, 7-5
  restrictions, 7-6
  using in ROLLBACK statement, 7-5
trace facility, using to improve performance, C-4
transaction, 7-2
  subdividing with savepoints, 7-5
  undoing, 7-3
  undoing parts of, 7-5
  when rolled back automatically, 7-3, 7-4
transaction processing
  overview, 2-6
  statements used, 2-6
transaction, contents, 2-6, 7-2
  guidelines, 7-9
  how to begin, 7-2
  how to end, 7-2
  in-doubt, 7-9
  making permanent, 7-3
transaction, read-only, 7-6
transactions
  committing, E-7
  distributed, E-30
  rolling back, E-29
truncated value, 5-9
  detecting, 4-2
truncation error, when generated, 4-4
tuning, performance, C-1
TYPE statement, using the CHARF datatype specifier, 3-22
TYPE_CODE
  precompiler option, 6-44

U

UID function, 3-8
unconditional delete, 8-20
undo a transaction, E-29
UNSAFE_NULL option, 6-44, A-1
UNSIGNED datatype, 3-13
update cascade, 8-19
UPDATE command, E-34
  embedded SQL examples, E-36
UPDATE statement, 4-6
  example, 4-6
  SET clause, 4-6
  using host arrays, 9-5
  using SQLERRD(3), 9-10
updating, rows in tables and views, E-34
user exit, 11-2
  calling from a SQL*Forms trigger, 11-5
  common uses, 11-2

example, 11-7
guidelines, 11-8
linking into IAP, 11-8
meaning of codes returned by, 11-6
naming, 11-8
passing parameters, 11-5
requirements for variables, 11-3
running the GENXTB form, 11-7
statements allowed in, 11-3
steps in developing, 11-3
using EXEC IAF statements, 11-4
using EXEC TOOLS statements, 11-9
using the WHENEVER statement, 11-7
USER function, 3-8
user session, 7-1
user-defined datatype, 3-17
user-defined record, 5-4
USERID option, 6-45
using with the SQLCHECK option, D-2
USING clause
CONNECT statement, 3-28
of FETCH command, E-23
of OPEN command, E-27
using in the EXECUTE statement, 10-8
using indicator variables, 10-8
using dbstring, SQL*Net database id specification, E-10

V

V7
value of DBMS option, 6-19
VALUES clause
INSERT statement, 4-6
of embedded SQL INSERT command, E-25
of INSERT command, E-25
using a subquery, 4-6
VAR command, E-37
examples, E-37
VAR statement, 3-20
parameters, 3-20
using the CHARF datatype specifier, 3-22
VARCHAR datatype, 3-13
VARCHAR pseudotype, 3-17, 5-8
maximum length, 3-17
using with PL/SQL, 5-8
VARCHAR, precompiler option, 6-46
VARCHAR2 column
maximum width, 3-6
VARCHAR2 datatype
external, 3-13
internal, 3-6
variable, 2-4
VARNUM datatype, 3-14
example of output value, 3-22
VARRAW, 3-14
VARRAW datatype, 3-14
VERSION precompiler option, 6-46
views
inserting rows into, E-24
updating rows in, E-34

W

warning flag, 8-17
when empty, 4-9
WHENEVER command, E-38
example, E-39
WHENEVER statement, 8-22
check SQLCA automatically, 8-22
CONTINUE action, 8-23
DO action, 8-23
examples, 8-23
GOTO action, 8-23
guidelines, 8-24
handling end-of-data conditions, 8-25
maintaining addressability, 8-26
NOT FOUND condition, 8-22
overview, 2-6
scope, 8-24
SQLERROR condition, 8-22
SQLWARNING condition, 8-22
STOP action, 8-23
where to place, 8-25
WHERE Clause
DELETE statement, 4-7
WHERE clause, 4-7
of DELETE command, E-16
of UPDATE command, E-36
search condition, 4-7
SELECT statement, 4-5
UPDATE statement, 4-6
using host arrays, 9-9
WORK option
of COMMIT command, E-8
of ROLLBACK command, E-29

X

XA interface, 3-34
X/Open application, 3-34
XREF option, 6-47