

Oracle7TM Server Application Developer's Guide

Release 7.3

February 1996

Part No. A32536-1

ORACLE[®]

Oracle7™ Server Application Developer's Guide, Release 7.3

Part No. A32536-1

Copyright © 1992, 1996 Oracle Corporation

All rights reserved. Printed in the U.S.A.

Contributing Authors: Steven Bobrowski, Maria Pratt, Timothy Smith
Contributors: Richard Allen, George Buzsaki, Atif Chaudhry, Greg Doherty,
Gary Hallmark, Michael Hartstein, Kenneth Jacobs, Hakan Jakobsson, Amit
Jasuja, Robert Jenkins, Jonathan Klein, Robert Kooi, Juan Loaiza, William
Maimone, Andrew Mendelsohn, Mark Moore, Valarie Moore, Ravi Narayanan,
Edward Peeler, Thomas Portfolio, Tuomas Pystnen, Mark Ramacher, Usha
Sundaram, Scott Urman, Peter Vasterd, Joyo Wijaya

This software was not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer's responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

If this software/documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights", as defined in FAR 52.227-14, Rights in Data - General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

Oracle, Pro*Ada, Pro*COBOL, Pro*FORTRAN, Pro*PL/I, Pro*Pascal, SQL*DBA, SQL*Loader, SQL*Net, and SQL*Plus are registered trademarks of Oracle Corporation.

Oracle7, Oracle Forms, Oracle Reports, Oracle Parallel Server, PL/SQL, Pro*C, Pro*C/C++, Trusted Oracle, and Trusted Oracle7 are trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.



Preface

This Guide describes features of the Oracle7 Server, release 7.3. Information in this Guide applies to versions of the Oracle7 Server that run on all platforms, and does not include system-specific information.

Information in this Guide

As an application developer, you should learn about the many Oracle7 features that can ease application development and improve performance.

This Guide describes Oracle7 server features that relate to application development. This Guide does not cover the PL/SQL language, nor does it directly discuss application development on the client side. See the table of contents and Chapter 1 in this Guide for information about the material covered. Chapter 1 also points you to other Oracle documentation that contains related information.

Other Guides

Use the *PL/SQL User's Guide and Reference* to learn PL/SQL, and to get a complete description of this high-level programming language, which is Oracle Corporation's procedural extension to SQL.

The *Programmer's Guide to the Oracle Call Interface* describes the Oracle Call Interface, which you can use to build third-generation language (3GL) applications that access Oracle. The OCI for release 7.3 incorporates many new capabilities. See the section "Using the Oracle Call Interface" on page 2 – 7 for more information about these new calls.

Oracle Corporation also provides the Pro* series of precompilers, which allow you to embed SQL and PL/SQL in your application programs. If you write 3GL application programs in Ada, C, C++, COBOL, or FORTRAN that incorporate embedded SQL, refer to the corresponding precompiler manual. For example, if you program in C or C++, refer to the *Programmer's Guide to the Oracle Pro*C/C++ Precompiler*.

Oracle Developer/2000 is a cooperative development environment that provides several tools including a form builder, reporting tools, and a debugging environment for PL/SQL. If you use Developer/2000, refer to the appropriate Tools documentation.

How this Guide Is Organized

The *Oracle7 Server Application Developer's Guide* is divided into 13 chapters, as described below.

Chapter 1: Information about Application Development

This chapter provides a road map that enables you to determine where to find information about specific application development topics, both in this Guide and in other Oracle technical publications.

Chapter 2: The Application Developer

This chapter provides an overview of the Oracle Server application development process.

Chapter 3: Processing SQL and PL/SQL Statements

This chapter explains the steps that the Oracle Server performs to process the various types of SQL commands and PL/SQL statements.

Chapter 4: Managing Schema Objects

This chapter describes how to manage the objects that can be created in the database domain of a specific user (schema), including tables, views, numeric sequences, and synonyms. Performance enhancements to data retrieval through the use of indexes and clusters are also discussed.

Chapter 5: Selecting a Datatype

This chapter describes how to choose the correct Oracle datatype. The datatypes described include fixed- and variable-length character strings, numeric data, dates, and binary large objects.

Chapter 6: Maintaining Data Integrity

This chapter describes how to use declarative integrity constraints to provide data integrity within an Oracle database.

Chapter 7: Using Procedures and Packages

This chapter describes how to create procedures that can be stored in the database for continued use. Grouping these procedures into packages is also described.

Chapter 8: PL/SQL Input and Output

This chapter describes how to use public and private pipes to allow sessions in the same instance to communicate with one another.

Chapter 9: Using Database Triggers

This chapter describes how to create and debug database triggers. Numerous examples are included.

Chapter 10: Using Dynamic SQL

This chapter describes how you can write stored procedures and anonymous PL/SQL blocks using dynamic SQL.

Chapter 11: Managing Dependencies Among Schema Objects

This chapter describes how to manage the dependencies between related views, procedures, packages, and triggers.

Chapter 12: Signalling Events in the Database with Alerters

This chapter describes how you can design your application to be notified whenever values in the database that are of interest to the application are changed.

Chapter 13: Establishing a Security Policy

This chapter describes how to design a security policy using Oracle's security features.

Notational Conventions

Text Formatting

UPPERCASE	Uppercase text is used to call attention to command keywords, as well as database objects such as tables and stored procedures, procedure parameters, and so on. For example: If you create a private rollback segment, the name of the rollback segment must be included in the ROLLBACK_SEGMENTS parameter of the parameter file.
<i>Italics</i>	Italicized words within text are used to indicate the first occurrence and definition of a term, as in the following example: A <i>database</i> is a collection of data to be treated as a unit. The general purpose of a database is to store and retrieve related information, as needed. Italicized words are also used for book titles and filenames.

Bold

Bold text is used to call special attention to important information. For example:

In summary, remember that this procedure provides a reasonable **estimate** of a table's size, not an exact number of blocks or bytes.

Examples of Commands and Statements

Server Manager and SQL*Plus commands, and SQL statements appear separated from the text of paragraphs in a fixed-width font:

```
ALTER TABLESPACE users ADD DATAFILE 'users2.ora' SIZE 50K;
```

Punctuation

Example statements can include punctuation such as commas or quotation marks. All punctuation given in example statements is required. Depending on the application being used, a semicolon or other terminator might be required to end a statement.

UPPERCASE

Uppercase words in example statements are used to indicate the keywords within Oracle SQL. For example, keywords such as SELECT or INSERT in SQL statements are uppercase in command examples, as well as in running text. However, when issuing SQL statements, you can use upper or lower case.

Lowercase

Lowercase words in example statements are used to indicate words supplied only for the context of the example. For example, lowercase words may indicate the name of a table, column, or file.

Unless enclosed in double quotes ("), Oracle converts the names of objects such as tables and column names to uppercase. Some operating systems are case sensitive, so refer to your operating system-specific documentation to determine whether you must pay attention to case for filenames and other operating system- or language-specific objects.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate, your opinions are the most important input we receive. At the back of this Guide is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this Guide or other Oracle manuals. Even if the form has been used, please feel free to write or FAX any comments you might have. Our address is:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

FAX: +1.415.506.7200.



Contents

Chapter 1	Information About Application Development	1 – 1
	Sources of Information	1 – 2
	Specific Topics	1 – 2
	Business Rules	1 – 2
	Client-Side Tools	1 – 2
	Communicating with 3GL Programs	1 – 3
	Database Constraints	1 – 3
	Database Design	1 – 3
	Datatypes	1 – 3
	Debugging	1 – 3
	Error Handling	1 – 3
	Gateways	1 – 4
	Oracle-Supplied Packages	1 – 4
	PL/SQL	1 – 4
	Schema Objects	1 – 4
	Security	1 – 5
	SQL Statements	1 – 5
	Tools	1 – 5
 Chapter 2	 The Application Developer	 2 – 1
	Assessing Needs	2 – 2
	Designing the Database	2 – 2
	Designing the Application	2 – 4
	Using Available Features	2 – 4
	Using the Oracle Call Interface	2 – 7
	Writing SQL	2 – 7

Enforcing Security in Your Application	2 – 8
Tuning an Application	2 – 8
Maintaining and Updating an Application	2 – 10

Chapter 3

Processing SQL Statements	3 – 1
SQL Statement Execution	3 – 2
FIPS Flagging	3 – 2
Controlling Transactions	3 – 4
Improving Performance	3 – 4
Committing a Transaction	3 – 5
Rolling Back a Transaction	3 – 6
Defining a Transaction Savepoint	3 – 6
Privileges Required for Transaction Management	3 – 7
Read-Only Transactions	3 – 8
The Use of Cursors	3 – 9
Declaring and Opening Cursors	3 – 9
Using a Cursor to Re-Execute Statements	3 – 10
Closing Cursors	3 – 10
Explicit Data Locking	3 – 11
Explicitly Acquiring Table Locks	3 – 12
Privileges Required	3 – 15
Explicitly Acquiring Row Locks	3 – 15
SERIALIZABLE and ROW_LOCKING Parameters	3 – 16
Summary of Non-Default Locking Options	3 – 17
Creating User Locks	3 – 18
The DBMS_LOCK Package	3 – 19
Security	3 – 19
Creating the DBMS_LOCK Package	3 – 20
ALLOCATE_UNIQUE Procedure	3 – 20
REQUEST Function	3 – 21
CONVERT Function	3 – 23
RELEASE Function	3 – 24
SLEEP Procedure	3 – 25
Sample User Locks	3 – 26
Viewing and Monitoring Locks	3 – 27
Concurrency Control Using Serializable Transactions	3 – 27
Serializable Transaction Interaction	3 – 29
Setting the Isolation Level	3 – 30
Referential Integrity and Serializable Transactions	3 – 31
READ COMMITTED and SERIALIZABLE Isolation	3 – 33
Application Tips	3 – 36

Chapter 4

Managing Schema Objects	4 – 1
Managing Tables	4 – 2
Designing Tables	4 – 2
Creating Tables	4 – 3
Altering Tables	4 – 8
Dropping Tables	4 – 9
Managing Views	4 – 10
Creating Views	4 – 10
Replacing Views	4 – 13
Using Views	4 – 14
Dropping Views	4 – 15
Modifying a Join View	4 – 16
Key-Preserved Tables	4 – 18
Rule for DML Statements on Join Views	4 – 18
Using the UPDATABLE_COLUMNS Views	4 – 21
Outer Joins	4 – 21
Managing Sequences	4 – 25
Creating Sequences	4 – 25
Altering Sequences	4 – 26
Using Sequences	4 – 26
Dropping Sequences	4 – 31
Managing Synonyms	4 – 31
Creating Synonyms	4 – 31
Using Synonyms	4 – 32
Dropping Synonyms	4 – 32
Managing Indexes	4 – 33
Creating Indexes	4 – 36
Dropping Indexes	4 – 37
Managing Clusters, Clustered Tables, and Cluster Indexes	4 – 38
Guidelines for Creating Clusters	4 – 38
Performance Considerations	4 – 39
Creating Clusters, Clustered Tables, and Cluster Indexes .	4 – 39
Manually Allocating Storage for a Cluster	4 – 41
Dropping Clusters, Clustered Tables, and Cluster Indexes	4 – 41
Managing Hash Clusters and Clustered Tables	4 – 43
Creating Hash Clusters and Clustered Tables	4 – 43
Controlling Space Usage Within a Hash Cluster	4 – 43
Dropping Hash Clusters	4 – 44
When to Use Hashing	4 – 45
Miscellaneous Management Topics for Schema Objects	4 – 47
Creating Multiple Tables and Views in One Operation	4 – 47
Naming Objects	4 – 48
Name Resolution in SQL Statements	4 – 48

Renaming Schema Objects	4 – 49
Listing Information about Schema Objects	4 – 50

Chapter 5

Selecting a Datatype	5 – 1
Oracle Datatypes	5 – 2
Character Datatypes	5 – 2
NUMBER Datatype	5 – 4
DATE Datatype	5 – 5
LONG Datatype	5 – 7
RAW and LONG RAW Datatypes	5 – 9
ROWIDs and the ROWID Datatype	5 – 9
Summary of Oracle Datatype Information	5 – 13
Trusted Oracle MLSLABEL Datatype	5 – 13
The ALL_LABELS Data Dictionary View	5 – 14
Adding New Labels	5 – 14
Trusted Oracle ROWLABEL Column	5 – 14
Retrieving Row Labels	5 – 15
Modifying Row Labels	5 – 16
Displaying Your DBMS Label	5 – 16
Altering Your DBMS Label	5 – 17
ANSI/ISO, DB2, and SQL/DS Datatypes	5 – 18
Data Conversion	5 – 19
Rule 1: Assignments	5 – 19
Rule 2: Expression Evaluation	5 – 21
Data Conversion for Trusted Oracle	5 – 22
The TO_CHAR Function	5 – 22
The TO_LABEL Function	5 – 22
Formatting Labels	5 – 23
The TO_CHAR and TO_LABEL Functions	5 – 23
Comparing Labels	5 – 24
Guidelines for Defining Labels	5 – 25
Embedded Spaces and Punctuation	5 – 25
Case Sensitivity	5 – 25
Label Components	5 – 25
Numeric Format	5 – 26
Number of Classifications and Categories	5 – 26

Maintaining Data Integrity	6 – 1
Using Integrity Constraints	6 – 2
When to Use Integrity Constraints	6 – 2
Taking Advantage of Integrity Constraints	6 – 3
Using NOT NULL Integrity Constraints	6 – 3
Setting Default Column Values	6 – 4
Choosing a Table's Primary Key	6 – 5
Using UNIQUE Key Integrity Constraints	6 – 6
Using Referential Integrity Constraints	6 – 7
Nulls and Foreign Keys	6 – 7
Relationships Between Parent and Child Tables	6 – 8
Multiple FOREIGN KEY Constraints	6 – 10
Concurrency Control, Indexes, and Foreign Keys	6 – 10
Referential Integrity in a Distributed Database	6 – 13
Using CHECK Integrity Constraints	6 – 13
Restrictions on CHECK Constraints	6 – 14
Designing CHECK Constraints	6 – 14
Multiple CHECK Constraints	6 – 15
CHECK and NOT NULL Integrity Constraints	6 – 15
Defining Integrity Constraints	6 – 15
The CREATE TABLE Command	6 – 16
The ALTER TABLE Command	6 – 16
Required Privileges	6 – 17
Naming Integrity Constraints	6 – 17
Enabling and Disabling Constraints Upon Definition	6 – 18
UNIQUE Key, PRIMARY KEY, and FOREIGN KEY	6 – 18
Enabling and Disabling Integrity Constraints	6 – 18
Why Enable or Disable Constraints?	6 – 19
Integrity Constraint Violations	6 – 19
On Definition	6 – 19
Enabling and Disabling Defined Integrity Constraints	6 – 20
Enabling and Disabling Key Integrity Constraints	6 – 22
Enabling Constraints after a Parallel Direct Path Load	6 – 22
Exception Reporting	6 – 23
Altering Integrity Constraints	6 – 25
Dropping Integrity Constraints	6 – 25
Managing FOREIGN KEY Integrity Constraints	6 – 25
Defining FOREIGN KEY Integrity Constraints	6 – 26
Enabling FOREIGN KEY Integrity Constraints	6 – 27
Listing Integrity Constraint Definitions	6 – 28
Examples	6 – 28

Using Procedures and Packages	7 – 1
PL/SQL	7 – 2
Anonymous Blocks	7 – 2
Database Triggers	7 – 4
Stored Procedures and Functions	7 – 5
Creating Stored Procedures and Functions	7 – 9
Altering Stored Procedures and Functions	7 – 11
Packages	7 – 11
Creating Packages	7 – 13
Creating Packaged Objects	7 – 13
Naming Packages and Package Objects	7 – 14
Dropping Packages and Procedures	7 – 14
Package Invalidations and Session State	7 – 15
Timestamps and Signatures	7 – 16
Timestamps	7 – 16
Disadvantages of the Timestamp Model	7 – 16
Signatures	7 – 17
Controlling Remote Dependencies	7 – 23
Suggestions for Managing Dependencies	7 – 25
Cursor Variables	7 – 26
Declaring and Opening Cursor Variables	7 – 26
Examples	7 – 27
Hiding PL/SQL Code	7 – 29
Error Handling	7 – 30
Declaring Exceptions and Exception Handling Routines ..	7 – 31
Unhandled Exceptions	7 – 32
Handling Errors in Distributed Queries	7 – 33
Handling Errors in Remote Procedures	7 – 33
Compile Time Errors	7 – 34
Debugging	7 – 36
Invoking Stored Procedures	7 – 36
A Procedure or Trigger Calling Another Procedure	7 – 37
Interactively Invoking Procedures From Oracle Tools	7 – 37
Calling Procedures within 3GL Applications	7 – 38
Name Resolution When Invoking Procedures	7 – 39
Privileges Required to Execute a Procedure	7 – 39
Specifying Values for Procedure Arguments	7 – 39
Invoking Remote Procedures	7 – 40
Referencing Remote Objects	7 – 41
Synonyms for Procedures and Packages	7 – 42
Calling Stored Functions from SQL Expressions	7 – 43
Using PL/SQL Functions	7 – 43
Syntax	7 – 44

Naming Conventions	7 – 44
Meeting Basic Requirements	7 – 46
Controlling Side Effects	7 – 47
Overloading	7 – 52
Privileges Required	7 – 52
Supplied Packages	7 – 53
Support for SQL Features	7 – 53
Additional Functionality	7 – 56
Describing Stored Procedures	7 – 57
DBMS_DESCRIBE Package	7 – 57
Security	7 – 57
Types	7 – 58
Errors	7 – 58
DESCRIBE_PROCEDURE Procedure	7 – 59
Listing Information about Procedures and Packages	7 – 65

Chapter 8

PL/SQL Input/Output	8 – 1
Database Pipes	8 – 2
Summary	8 – 2
Creating the DBMS_PIPE Package	8 – 3
Public Pipes	8 – 3
Private Pipes	8 – 4
Errors	8 – 4
CREATE_PIPE	8 – 4
PACK_MESSAGE Procedures	8 – 6
SEND_MESSAGE	8 – 6
RECEIVE_MESSAGE	8 – 8
NEXT_ITEM_TYPE	8 – 9
UNPACK_MESSAGE Procedures	8 – 10
REMOVE_PIPE	8 – 10
Managing Pipes	8 – 11
Purging the Contents of a Pipe	8 – 11
Resetting the Message Buffer	8 – 12
Getting a Unique Session Name	8 – 12
Example 1: Debugging	8 – 12
Example 2: Execute System Commands	8 – 14
Output from Stored Procedures and Triggers	8 – 21
Summary	8 – 21
Creating the DBMS_OUTPUT Package	8 – 22
Errors	8 – 22
ENABLE Procedure	8 – 22
DISABLE Procedure	8 – 23

PUT and PUT_LINE Procedures	8 – 23
GET_LINE and GET_LINES Procedures	8 – 24
Examples Using the DBMS_OUTPUT Package	8 – 25
PL/SQL File I/O	8 – 28
Summary	8 – 28
Security	8 – 29
Declared Types	8 – 31
Exceptions	8 – 31
FOPEN	8 – 32
IS_OPEN	8 – 33
FCLOSE	8 – 34
FCLOSE_ALL	8 – 34
GET_LINE	8 – 35
PUT	8 – 36
NEW_LINE	8 – 37
PUT_LINE	8 – 38
PUTF	8 – 38
FFLUSH	8 – 40

Chapter 9

Using Database Triggers	9 – 1
Designing Triggers	9 – 2
Creating Triggers	9 – 3
Prerequisites	9 – 3
Naming Triggers	9 – 4
The BEFORE/AFTER Options	9 – 4
Triggering Statement	9 – 4
FOR EACH ROW Option	9 – 5
The WHEN Clause	9 – 6
The Trigger Body	9 – 7
Triggers and Handling Remote Exceptions	9 – 9
Restrictions on Creating Triggers	9 – 10
Who is the Trigger User?	9 – 15
Privileges Required to Create Triggers	9 – 15
Privileges for Referenced Schema Objects	9 – 15
When Triggers Are Compiled	9 – 16
Dependencies	9 – 16
Recompiling a Trigger	9 – 17
Migration Issues	9 – 17
Debugging a Trigger	9 – 17
Modifying a Trigger	9 – 18
Enabling and Disabling Triggers	9 – 18
Disabling Triggers	9 – 18

Enabling Triggers	9 – 19
Privileges Required to Enable and Disable Triggers	9 – 19
Listing Information About Triggers	9 – 20
Examples of Trigger Applications	9 – 21
Auditing with Triggers	9 – 21
Integrity Constraints and Triggers	9 – 25
Complex Security Authorizations and Triggers	9 – 33
Transparent Event Logging and Triggers	9 – 34
Derived Column Values and Triggers	9 – 34

Chapter 10

Using Dynamic SQL	10 – 1
Overview	10 – 2
Creating the DBMS_SQL Package	10 – 2
Using DBMS_SQL	10 – 2
Execution Flow	10 – 4
Security	10 – 7
For Oracle Server Users	10 – 7
For Trusted Oracle Server Users	10 – 7
Procedures and Functions	10 – 7
OPEN_CURSOR Function	10 – 9
PARSE Procedure	10 – 9
BIND_VARIABLE Procedures	10 – 10
Processing Queries	10 – 12
DEFINE_COLUMN Procedure	10 – 12
DEFINE_COLUMN_LONG Procedure	10 – 14
EXECUTE Function	10 – 14
EXECUTE_AND_FETCH Function	10 – 15
FETCH_ROWS Function	10 – 15
COLUMN_VALUE Procedure	10 – 16
COLUMN_VALUE_LONG Procedure	10 – 17
VARIABLE_VALUE Procedure	10 – 18
Processing Updates, Inserts and Deletes	10 – 19
IS_OPEN Function	10 – 20
CLOSE_CURSOR Procedure	10 – 20
Locating Errors	10 – 21
LAST_ERROR_POSITION Function	10 – 21
LAST_ROW_COUNT Function	10 – 21
LAST_ROW_ID Function	10 – 21
LAST_SQL_FUNCTION_CODE Function	10 – 21
Examples	10 – 22

Chapter 11	Managing Dependencies Among Schema Objects	11 – 1
	Dependency Issues	11 – 2
	Avoiding Runtime Recompilation	11 – 2
	Manually Recompiling	11 – 4
	Manually Recompiling Views	11 – 5
	Manually Recompiling Procedures and Functions	11 – 5
	Manually Recompiling Packages	11 – 5
	Manually Recompiling Triggers	11 – 6
	Listing Dependency Management Information	11 – 6
	The Dependency Tracking Utility	11 – 7
 Chapter 12	 Signalling Events in the Database with Alerters	 12 – 1
	Overview	12 – 2
	Creating the DBMS_ALERT Package	12 – 3
	Security	12 – 3
	Errors	12 – 3
	Using Alerts	12 – 4
	REGISTER Procedure	12 – 4
	REMOVE Procedure	12 – 5
	SIGNAL Procedure	12 – 5
	WAITANY Procedure	12 – 6
	WAITONE Procedure	12 – 6
	Checking for Alerts	12 – 7
	SET_DEFAULTS Procedure	12 – 8
	Example of Using Alerts	12 – 8

Chapter 13

Establishing a Security Policy 13 – 1

Application Security Policy 13 – 2

 Application Administrators 13 – 2

 Roles and Application Privilege Management 13 – 2

 Enabling Application Roles 13 – 3

 Restricting Application Roles from Tool Users 13 – 5

 Schemas 13 – 8

Managing Privileges and Roles 13 – 8

 Creating a Role 13 – 10

 Enabling and Disabling Roles 13 – 10

 Dropping Roles 13 – 13

 Granting and Revoking Privileges and Roles 13 – 14

 Granting to, and Revoking from, the User Group PUBLIC 13 – 19

 When Do Grants and Revokes Take Effect? 13 – 20

 How Do Grants Affect Dependent Objects? 13 – 20

Index

CHAPTER

1

Information About Application Development

This chapter lists some of the topics discussed in this Guide, and tells you where you can get information about each topic. The topics are arranged in alphabetic order for quick reference.

Sources of Information

The Oracle7 Server is a large product. There are over 20 books that form the documentation for the Oracle Server and languages products. In addition to these, there are several books in the Network documentation set that provide important information on using SQL*Net to connect client applications to Oracle7 servers.

In this chapter, you can get some basic information about Oracle application development products, and the documentation for these products. In addition to Oracle documentation, there is an ever-increasing set of trade books about Oracle7. Visit your local technical or university bookstore to discover the titles available.

Specific Topics

Business Rules

You can enforce business rules in your Oracle application using integrity constraints on columns of a table, or by using triggers. See Chapter 6 in this Guide for a description of integrity constraints, and Chapter 9 for a discussion of database triggers.

See the *Oracle7 Server Concepts* manual for a high-level discussion of business rules.

Client-Side Tools

Oracle provides a number of tools to help you develop your applications. Oracle's Developer/2000 tool set offers *Procedure Builder*, a PL/SQL development environment with a client-side debugger, as well as other tools that generate forms and reports.

CASE tools to help you in database design are available as part of the Oracle Designer/2000 product.

Communicating with 3GL Programs

Application developers frequently ask how they can access 3GL routines (such as C or C++ functions) or operating system services from PL/SQL code that is running on a server. One way to do this is to use the Oracle-supplied DBMS_PIPE package. See Chapter 8 for a detailed discussion of this package. This package is not transaction safe, so it must be used with care.

It is not possible to call a 3GL routine directly from PL/SQL in release 7.3. This capability will be offered in a future release of the Oracle Server.

Database Constraints

How to use database constraints such as NOT NULL, PRIMARY KEY, and FOREIGN KEY is described in Chapter 6 of this Guide. See the *Oracle7 Server Concepts* manual for a basic introduction to constraints.

Database Design

Database design is not discussed exhaustively in this Guide. See the *Oracle7 Server Concepts* for a basic discussion. Refer to the *Oracle7 Server Tuning* manual for tips on designing performance into your database. You can also refer to the Oracle Designer/2000 product for tools for advanced database design.

Datatypes

Oracle internal datatypes are described in Chapter 5 of this Guide. For a more comprehensive treatment of datatypes, see the *Programmer's Guide to the Oracle Call Interface*.

Debugging

You can use the DBMS_OUTPUT and the DBMS_PIPE packages for first-level debugging of your PL/SQL code. See Chapter 8 in this Guide for more information.

Error Handling

When errors or warnings occur as you compile or run an Oracle application, the information is sent to you through an Oracle error code, usually accompanied by a short error message. See the *Oracle7 Server*

Messages manual for a complete listing of Oracle server, precompiler, and PL/SQL error codes and messages.

PL/SQL can also generate exceptions at runtime. See the *PL/SQL User's Guide and Reference* for a list of predefined PL/SQL exceptions and their causes.

Gateways

You can use Oracle's Open Gateway technology to access data on non-Oracle databases, and even on non-relational data sources. See the *Oracle Open Gateway Toolkit Guide* for information about developing gateway applications.

Oracle-Supplied Packages

Oracle supplies a set of PL/SQL packages to assist your application development. Most of the packages' names begin with the prefix DBMS_, for example DBMS_OUTPUT or DBMS_SQL. A few have other prefixes, such as UTL_FILE (for PL/SQL file I/O).

These supplied packages are documented in this Guide, mainly in Chapters 7, 8, 9, 10, and 12. See pages 7 – 53 and following for a complete list of the Oracle-supplied packages, and references to where they are documented.

PL/SQL

The primary source for documentation of the PL/SQL language is the *PL/SQL User's Guide and Reference*. How you use PL/SQL in application development is documented both in that Guide as well as in this manual. There are also several trade books available that cover the PL/SQL language.

Schema Objects

This Guide documents how to create, modify, and delete schema objects such as tables, views, packages, procedures, and sequences. However, you should be familiar with the material in the *Oracle7 Server Concepts* manual for introductory material. For example, you might want to read the chapter in the Concepts manual called "Procedures and Packages" before looking at Chapter 7 ("Using Procedures and Packages") in this Guide.

Security

Your primary reference for security in this Guide is Chapter 13. Security issues are also discussed in the *Oracle7 Server Concepts* manual.

SQL Statements

Your primary reference for the SQL language is the *Oracle7 Server SQL Reference*. That manual covers Oracle's implementation of the SQL language in depth. It includes syntax diagrams that summarize the form of all SQL commands.

Tools

See the section "Client-Side Tools" on page 1 – 2.

The Application Developer

This chapter briefly outlines the steps involved in designing and implementing an Oracle database application. More detailed information needed to perform these tasks is provided later in this Guide. Although the specific tasks vary depending upon the type and complexity of the application being developed, in general the responsibilities of the application developer include the following:

- designing the database structure for the application
- designing and developing the database application
- writing SQL code
- enforcing security in the application
- tuning an application
- maintaining and updating applications

This book is not meant to serve as a textbook on database or application design. If you are not already familiar with these areas, you should consult a text for guidance. Where appropriate, you are directed to other sections of this document for additional information.

Assessing Needs

The first step in designing a usable application is determining what problem you are trying to solve. It is important that you do not focus entirely on the data, but rather on how the data is being used. In designing your application you should try to answer the following questions:

- Who will be using this application?
- What are they trying to accomplish by using this application?
- How will they be accomplishing these tasks?

You should involve the end-user as much as possible early in the design phase. This helps eliminate problems that can stem from misunderstandings about the purpose of the application. After you gain a better understanding of the tasks that the end-users of the application are trying to perform, you can then determine the data that is necessary to complete these tasks. In this step, you need to look at each task and decide:

- What data must be available to perform this task?
- How must this data be processed?
- How can these results be meaningfully presented?
- What are the potential future uses of this application?

It is important that your audience has a clear understanding of your proposed solution. It is also important that your application be designed to accommodate the changing needs of your audience.

Designing the Database

At this point, you are ready to begin designing your data model. This model will allow you to determine how your data can be most efficiently stored and used. The Entity-Relationship model is often used to map a real-world system to a relational database management system.

The Entity-Relationship model categorizes all elements of a system as either an entity (a person, place, or thing) or a relationship between entities. Both constructs are represented by the same structure, a table. For example, in an order entry system, parts are entities, as are orders. Both part and order information is represented in tables. The relationship of which parts are requested by which order is also

represented by a third table. The application of the Entity–Relationship model requires the following steps.

- First, identify the entities of your system and construct a table to represent each entity.
- Second, identify the relationships between the entities and either extend the current tables or create new tables to represent these relationships.
- Third, identify attributes of each entity and extend the tables to include these attributes.

When modeling a system with the Entity–Relationship model, you will often include a step called normalization. Textbooks on database design will tell you how to achieve Third Normal Form. Each table must have exactly one primary key and, in third normal form, all of the data in a table is dependent solely upon the table's primary key. You might find it necessary to violate normal form on occasion to achieve a desired performance level.

Proper application of the Entity–Relationship model results in well designed tables. The benefits of a set of well designed tables include the following:

- reduced storage of redundant data, which eliminates the cost of updating duplicates and avoids the risk of inconsistent results based on duplicates
- increased ability to effectively enforce integrity constraints
- increased ability to adapt to the growth and change of the system
- increased productivity based on the inherent flexibility of well designed relational systems

Oracle Corporation's products for database design, CASE*Dictionary, CASE*Method, and CASE*Designer, can help improve, automate, and document designs. See the CASE manuals for additional information.

After determining the overall structure of the tables in your database, you must next design the structure of these tables. This process involves selecting the proper datatype for each column and assigning each column a meaningful name. You can find information about selecting the appropriate Oracle datatype in Chapter 5 of this Guide.

If you are creating an application that runs on a distributed database, you must also determine where to locate this data and any links that are necessary to access the data across the network; see *Oracle7 Server Distributed Systems, Volume I* for additional information.

Designing the Application

After completing your database design, you are ready to begin designing the application itself. This, too, is an iterative process, and might also cause you to rethink your database design. As much as possible, you should involve your audience in these design decisions. You should make your application available to the end-users as early as possible in order for them to provide you with the feedback needed to fine tune your design.

There are many tools available, from Oracle Corporation as well as other vendors, to aid in the development and implementation of your application. Your first task is to evaluate the available tools and select those that are most appropriate.

Using Available Features

You must next determine how to implement your requirements using the features available in Oracle, as well as any other tools and utilities that you selected in the previous step. The features and tools that you choose to use to implement your application can significantly affect the performance of your application. The more effort you put into designing an efficient application, the less time you will have to spend tuning the application once it is complete.

Several of the more useful features available to Oracle application developers are listed below. Each of these topics is discussed in detail later in this book.

Integrity Constraints

Integrity constraints allow you to define certain requirements for the data that can be included in a table, and to ensure that these requirements are met regardless of how the data is entered. These constraints are included as part of the table definition, and require no programming to be enforced; see Chapter 6 for instructions on their use.

Stored Procedures and Packages

Commonly used procedures can be written once in PL/SQL and stored in the database for repeated use by applications. This ensures consistent behavior among applications, and can reduce your development and testing time.

Related procedures can be grouped into packages, which have a package specification separate from the package body. The package body can be altered and recompiled without affecting the package

specification. This allows you to make changes to the package body that are not visible to end-users, and that do not require objects referencing the specification to be recompiled. See Chapter 7 for additional information.

Database Triggers

Complex business rules that cannot be enforced using declarative integrity constraints can be enforced using triggers. Triggers, which are similar to PL/SQL anonymous blocks, are automatically executed when a triggering statement is issued, regardless of the user or application. See Chapter 9 for additional information.

Database triggers can have such diverse uses as performing value-based auditing, maintaining derived data values, and enforcing complex security or integrity rules. By moving this code from your application into database triggers, you can ensure that all applications behave in a uniform manner.

Cost-Based Optimizer

The cost-based optimization method uses statistics about tables, along with information about the available indexes, to select an execution plan for SQL statements. This allows even inexperienced users to submit complex queries without having to worry about performance.

As an application designer, there may be times when you have knowledge of the data in your table that is not available to the optimizer, and that allows you to select a better execution path. In these cases, you can provide hints to the optimizer to allow it to select the proper execution path. See the *Oracle7 Server Tuning* manual for more information.

Shared SQL

Shared SQL allows multiple users to share a single runtime copy of procedures and SQL statements, significantly reducing memory requirements. If two identical SQL statements are issued, the shared SQL area used to process the first instance of the statement is reused for the processing of the subsequent instances of the same statement.

You should coordinate with your database administrator (DBA), as well as other application developers, to establish guidelines to ensure that statements and blocks that perform similar tasks can use the same shared SQL areas as often as possible. See your *Oracle7 Server Tuning* manual for additional information.

National Language Support

Oracle supports both single and multi-byte character encoding schemes. Because language-dependent data is stored separately from the code, you can easily add new languages and language-specific features (such as date formats) without altering your application code. Refer to the *Oracle7 Server Reference* manual for more information on national language support.

Locking

By default, Oracle provides row-level locking, allowing multiple users to access different rows of the same table without lock contention. Although this greatly reduces the chances of deadlocks occurring, you should still take care in designing your application to ensure that deadlocks do not occur.

Online transaction processing applications—that is, applications with multiple users concurrently modifying different rows of the same table—benefit the most from row-level locking. You should design your application with this feature in mind.

Oracle locks are also available to you for use within your applications. These locks are provided as part of the DBMS_LOCK package, which is described in Chapter 3.

Profiles

Profiles can be used to enforce per-query and per-session limits on resource use. When designing your applications, you might want to consider if any users have been denied access to the system due to limited resources. Profiles make it possible to allow these infrequent users limited access to the database. If you choose to allow access to these users, you must consider their requirements when formulating your design. Profiles are generally controlled by the database administrator. Consult your database administrator to determine if access can be granted to additional users and to identify this audience.

Sequences

You can use sequence numbers to automatically generate unique keys for your data, and to coordinate keys across multiple rows or tables. The sequence number generator eliminates the serialization caused by programmatically generating unique numbers by locking the most recently used value and then incrementing it. Sequence numbers can also be read from a sequence number cache, instead of disk, further increasing their speed.

Industry Standards Compliance

Oracle is designed to conform to industry standards. If your applications must conform to industry standards, you should consult the *Oracle7 Server SQL Reference* manual for a detailed explanation of Oracle's conformance to SQL standards.

Using the Oracle Call Interface

If you are developing applications that use the Oracle Call Interface (OCI), you should be aware that the OCI for Oracle7 release 7.3 offers many new calls. These provide

- new connection functionality
- the ability to insert and delete parts of a LONG or LONG RAW column, in addition to the previous capability to select pieces of these columns
- use arrays of C structs for bind and define operations
- a thread-safe library for OCI applications

Writing SQL

All operations performed on the information in an Oracle database are executed using SQL statements. After you have completed the design of your application, you need to begin designing the SQL statements that you will use to implement this design. You should have a thorough understanding of SQL before you begin to write your application. A general description of how SQL statements are executed is provided in Chapter 3 of this document. For more detailed information, see the *Oracle7 Server SQL Reference* manual.

You can significantly improve the performance of your application by tuning the SQL statements it uses. Tuning SQL statements is explained in detail in the *Oracle7 Server Tuning* manual.

Enforcing Security in Your Application

Your application design is not complete until you have determined the security requirements for the application. As part of your application design, you identified what tasks each user or group of users needed to perform. Now you must determine what privileges are required to perform these tasks. It is important to the security of the database that these users have no more access than is necessary to complete their tasks.

By having your application enable the appropriate roles when a user runs the application, you can ensure that the user can only access the database as you originally planned. Because roles are typically granted to users by the database administrator, you should coordinate with your database administrator to ensure that each user is granted access to the roles required by your application for a designated task. See Chapter 13 for more information on developing a security policy.

Tuning an Application

There are two important areas to think about when tuning your database application:

- tuning your SQL statements
- tuning your application design

Information on tuning your SQL statements, including how to use the cost-based optimization method, is included in the *Oracle7 Server Tuning* manual. Tuning your application design ideally occurs before you begin to implement your application. Before beginning your design, you should carefully read about each of the features described in this document and consider which features best suit your requirements. Some design decisions that you should consider are outlined below.

- Where possible, enforce business rules with integrity constraints rather than programmatically. See page 6 – 2 for a discussion of when to use integrity constraints.
- To improve performance, use PL/SQL. A description of how PL/SQL improves performance is included in the *Oracle7 Server Tuning* manual.
- Use packages to further improve performance and reduce runtime recompilations. Packages are described in Chapter 7.
- Use cached sequence numbers to generate primary key values; see page 4 – 29.
- Use array processing to reduce the number of calls to Oracle; see the *Oracle7 Server Tuning* manual.
- Use VARCHAR2 to store character data instead of CHAR, which blank-pads data; see page 5 – 3.
- Store LONG and LONG RAW data in tables separate from related data and use referential integrity to relate them. This allows you to access the related data without having to read the LONG or LONG RAW data; see page 5 – 8.
- Use the SET_MODULE and SET_ACTION procedures in the DBMS_APPLICATION_INFO package to record the name of the executing module or transaction in the database for use later when tracking the performance of various modules. Registering the application allows system administrators and performance tuning specialists to track performance by module. System administrators can also use this information to track resource usage by module. When an application registers with the database, its name and actions are recorded in the V\$SESSION and V\$SQLAREA views. Registering applications is described in the *Oracle7 Server Tuning* manual.

You should also work with your database administrator to determine how the database can be tuned to accommodate your application. More detailed information on tuning your application, as well as information on database tuning, is included in the *Oracle7 Server Tuning* manual.

Maintaining and Updating an Application

If you are upgrading an existing application, or writing a new application to run on an existing database, you must follow many of the same procedures described earlier in this section. You must identify and understand the needs of your audience and design your application to accommodate them.

You must also work closely with the database administrator to determine

- what existing applications are available and how they are being used
- what data is available, if any can be eliminated, or if any additional data must be collected
- if any modifications must be made to the database structure, and how to make these changes in the least disruptive manner

Processing SQL Statements

This chapter describes how Oracle processes Structured Query Language (SQL) statements. Topics include the following:

- SQL statement execution
- controlling transactions
- read-only transactions
- the use of cursors with SQL statements
- explicit data locking
- creating user locks
- concurrency control

Although some Oracle tools and applications simplify or mask the use of SQL, all database operations are performed using SQL. Any other data access method would circumvent the security built into Oracle and potentially compromise data security and integrity.

SQL Statement Execution

Figure 3 – 1 outlines the stages commonly used to process and execute a SQL statement. In some cases, these steps might be executed in a slightly different order. For example, the DEFINE stage could occur just before the FETCH stage, depending on how your code is written.

For many Oracle tools, several of the phases are performed automatically. Most users need not be concerned with or aware of this level of detail. However, you might find this information useful when writing Oracle applications. Refer to the *Oracle7 Server Tuning* manual for a description of each phase of SQL statement processing for each type of SQL statement.

FIPS Flagging

The Federal Information Processing Standard for SQL (FIPS 127–2) requires a way to identify SQL statements that use vendor-supplied extensions. Oracle provides a FIPS flagger to help you write portable applications.

When FIPS flagging is active, your SQL statements are checked to see whether they include extensions that go beyond the ANSI/ISO SQL92 standard. If any non-standard constructs are found, the Oracle Server flags them as errors and displays the violating syntax.

The FIPS flagging feature supports flagging through interactive SQL statements submitted using Server Manager or SQL*Plus. The Oracle Precompilers and SQL*Module also support FIPS flagging of embedded and module language SQL.

When flagging is on and non-standard SQL is encountered, the message returned is

```
ORA-00097: Use of Oracle SQL feature not in SQL92 level Level
```

where *level* can be either ENTRY, INTERMEDIATE, or FULL.

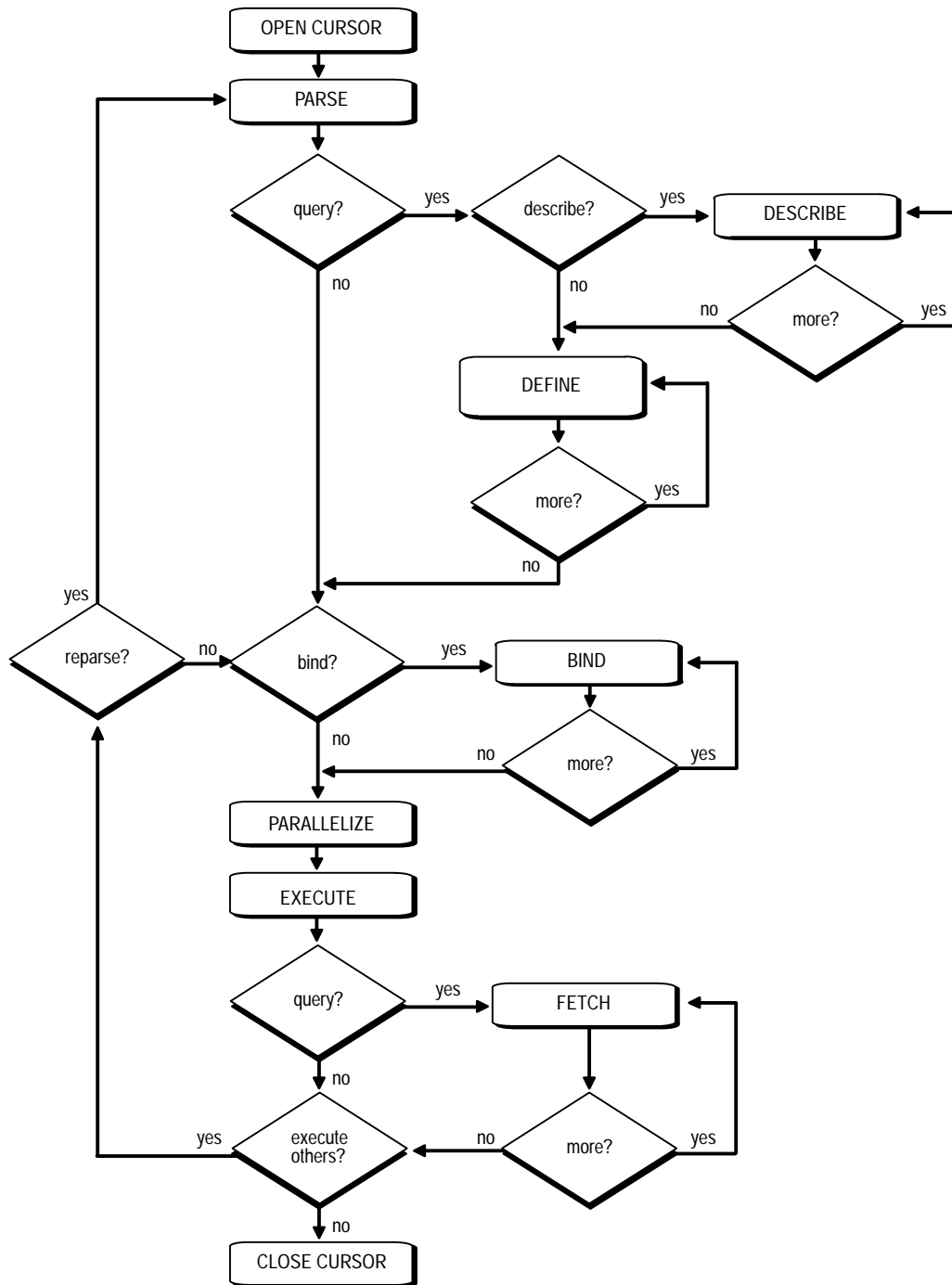


Figure 3 – 1 The Stages in Processing a SQL Statement

Controlling Transactions

In general, only application designers using the programming interfaces to Oracle are concerned with which types of actions should be grouped together as one transaction. Transactions must be defined properly so work is accomplished in logical units and data is kept consistent. A transaction should consist of all of the necessary parts for one logical unit of work, no more and no less. Data in all referenced tables should be in a consistent state before the transaction begins and after it ends. Transactions should consist of only the SQL statements or PL/SQL blocks that comprise one consistent change to the data.

A transfer of funds between two accounts (the transaction or logical unit of work), for example, should include the debit to one account (one SQL statement) and the credit to another account (one SQL statement). Both actions should either fail or succeed together as a unit of work; the credit should not be committed without the debit. Other non-related actions, such as a new deposit to one account, should not be included in the transfer of funds transaction.

Improving Performance

In addition to determining which types of actions form a transaction, when you design an application you must also determine if you can take any additional measures to improve performance. You should consider the following performance enhancements when designing and writing your application. Unless otherwise noted, each of these features is described in the *Oracle7 Server Tuning* manual.

- Use the `BEGIN_DISCRETE_TRANSACTION` procedure to improve the performance of short, non-distributed transactions.
- Use the `SET TRANSACTION` command with the `USE ROLLBACK SEGMENT` parameter to explicitly assign a transaction to an appropriate rollback segment. This can eliminate the need to dynamically allocate additional extents, which can reduce overall system performance.
- Use the `SET TRANSACTION` command with the `ISOLATION LEVEL` set to `SERIALIZABLE` to get ANSI/ISO serializable transactions. See page 3 – 27 in this chapter, and also Chapter 10 in the *Oracle7 Server Concepts* manual.

- Establish standards for writing SQL statements so that you can take advantage of shared SQL areas. Oracle recognizes identical SQL statements and allows them to share memory areas. This reduces memory storage usage on the database server, thereby increasing system throughput.
- Use the ANALYZE command to collect statistics that can be used by Oracle to implement a cost-based approach to SQL statement optimization. You can supply additional "hints" to the optimizer as needed.
- Call the DBMS_APPLICATION_INFO.SET_ACTION procedure before beginning a transaction to register and name a transaction for later use when measuring performance across an application. You should specify what type of activity a transaction performs so that the system tuners can later see which transactions are taking up the most system resources.
- Increase user productivity and query efficiency by including user-written PL/SQL functions in SQL expressions as described on page 7 – 43.
- Create explicit cursors when writing a PL/SQL application.
- When writing precompiler programs, increasing the number of cursors using MAX_OPEN_CURSORS can often reduce the frequency of parsing and improve performance. The use of cursors is described on page 3 – 9 of this Guide.

Committing a Transaction

To commit a transaction, use the COMMIT command. The following two statements are equivalent and commit the current transaction:

```
COMMIT WORK;
COMMIT;
```

The COMMIT command allows you to include the COMMENT parameter along with a comment (less than 50 characters) that provides information about the transaction being committed. This option is useful for including information about the origin of the transaction when you commit distributed transactions:

```
COMMIT COMMENT 'Dallas/Accts_pay/Trans_type 10B';
```

For additional information about committing in-doubt distributed transactions, see *Oracle7 Server Distributed Systems, Volume I*.

Rolling Back a Transaction

To roll back an entire transaction or a part of a transaction (that is, to a savepoint), use the ROLLBACK command. For example, either of the following statements rolls back the entire current transaction:

```
ROLLBACK WORK;  
ROLLBACK;
```

The WORK option of the ROLLBACK command has no function.

To roll back to a savepoint defined in the current transaction, the TO option of the ROLLBACK command must be used. For example, either of the following statements rolls back the current transaction to the savepoint named POINT1:

```
ROLLBACK TO SAVEPOINT point1;  
ROLLBACK TO point1;
```

For additional information about rolling back in-doubt distributed transactions see *Oracle7 Server Distributed Systems, Volume I*.

Defining a Transaction Savepoint

To define a *savepoint* in a transaction, use the SAVEPOINT command. The following statement creates the savepoint named ADD_EMP1 in the current transaction:

```
SAVEPOINT add_emp1;
```

If you create a second savepoint with the same identifier as an earlier savepoint, the earlier savepoint is erased. After a savepoint has been created, you can roll back to the savepoint.

There is no limit on the number of active savepoints per session. An active savepoint is one that has been specified since the last commit or rollback.

An Example of COMMIT, SAVEPOINT, and ROLLBACK

The following series of SQL statements illustrates the use of COMMIT, SAVEPOINT, and ROLLBACK statements within a transaction:

<i>SQL Statement</i>	<i>Results</i>
SAVEPOINT A;	First savepoint of this transaction.
DELETE . . . ;	First DML statement of this transaction.
SAVEPOINT b;	Second savepoint of this transaction.
INSERT INTO . . . ;	Second DML statement of this transaction.
SAVEPOINT c;	Third savepoint of this transaction.
UPDATE . . . ;	Third DML statement of this transaction.
ROLLBACK TO c;	UPDATE statement is rolled back, savepoint C remains defined.
ROLLBACK TO b;	INSERT statement is rolled back, savepoint C is lost, savepoint B remains defined.
ROLLBACK TO c;	ORA-01086 error; savepoint C no longer defined.
INSERT INTO . . . ;	New DML statement in this transaction.
COMMIT;	Commits all actions performed by the first DML statement (the DELETE statement) and the last DML statement (the second INSERT statement). All other statements (the second and the third statements) of the transaction had been rolled back before the COMMIT. The savepoint A is no longer active.

Privileges Required for Transaction Management

No privileges are required to control your own transactions; any user can issue a COMMIT, ROLLBACK, or SAVEPOINT statement within a transaction.

Read-Only Transactions

By default, the consistency model for Oracle guarantees statement-level read consistency, but does not guarantee transaction-level read consistency (repeatable reads). If you want transaction-level read consistency and your transaction does not require updates, you can specify a *read-only transaction*. After indicating that your transaction is read-only, you can execute as many queries as you like against any database table, knowing that the results of each query in the read-only transaction are consistent with respect to a single point in time.

A read-only transaction does not acquire any additional data locks to provide transaction-level read consistency. The multi-version consistency model used for statement-level read consistency is used to provide transaction-level read consistency; all queries return information with respect to the system control number (SCN) determined when the read-only transaction begins. Because no data locks are acquired, other transactions can query and update data being queried concurrently by a read-only transaction.

Changed data blocks queried by a read-only transaction are reconstructed using data from rollback segments. Therefore, long running read-only transactions sometimes receive a “snapshot too old” error (ORA-01555). Create more, or larger, rollback segments to avoid this. Alternatively, you could issue long-running queries when online transaction processing is at a minimum, or you could obtain a shared lock on the table you were querying, prohibiting any other modifications during the transaction.

A read-only transaction is started with a SET TRANSACTION statement that includes the READ ONLY option. For example:

```
SET TRANSACTION READ ONLY;
```

The SET TRANSACTION statement must be the first statement of a new transaction; if any DML statements (including queries) or other non-DDL statements (such as SET ROLE) precede a SET TRANSACTION READ ONLY statement, an error is returned. Once a SET TRANSACTION READ ONLY statement successfully executes, only SELECT (without a FOR UPDATE clause), COMMIT, ROLLBACK, or non-DML statements (such as SET ROLE, ALTER SYSTEM, LOCK TABLE) are allowed in the transaction. Otherwise, an error is returned. A COMMIT, ROLLBACK, or DDL statement terminates the read-only transaction (a DDL statement causes an implicit commit of the read-only transaction and commits in its own transaction).

The Use of Cursors

PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you can explicitly declare a cursor to process the rows individually.

A *cursor* is a handle to a specific private SQL area. In other words, a cursor can be thought of as a name for a specific private SQL area. A PL/SQL *cursor variable* enables the retrieval of multiple rows from a stored procedure. Cursor variables allow you to pass cursors as parameters in your 3GL application. Cursor variables are described in the *PL/SQL User's Guide and Reference*.

Although most Oracle users rely on the automatic cursor handling of the Oracle utilities, the programmatic interfaces offer application designers more control over cursors. In application development, a cursor is a named resource available to a program, which can be specifically used for parsing SQL statements embedded within the application.

Declaring and Opening Cursors

There is no absolute limit to the total number of cursors one session can have open at one time, subject to two constraints:

- Each cursor requires virtual memory, so a session's total number of cursors is limited by the memory available to that process.
- A system-wide limit of cursors per session is set by the initialization parameter named `OPEN_CURSORS` found in the parameter file (such as `INIT.ORA`). Parameters are described in the *Oracle7 Server Reference* manual.

Explicitly creating cursors for precompiler programs can offer some advantages in tuning those applications. For example, increasing the number of cursors can often reduce the frequency of parsing and improve performance. If you know how many cursors may be required at a given time, you can make sure you can open that many simultaneously.

Using a Cursor to Re-Execute Statements

After each stage of execution, the cursor retains enough information about the SQL statement to re-execute the statement without starting over, as long as no other SQL statement has been associated with that cursor. This is illustrated in Figure 3 – 1 on page 3 – 3. Notice that the statement can be re-executed without including the parse stage.

By opening several cursors, the parsed representation of several SQL statements can be saved. Repeated execution of the same SQL statements can thus begin at the describe, define, bind, or execute step, saving the repeated cost of opening cursors and parsing.

Closing Cursors

Closing a cursor means that the information currently in the associated private area is lost and its memory is deallocated. Once a cursor is opened, it is not closed until one of the following events occurs:

- The user program terminates its connection to the server.
- If the user program is an OCI program or precompiler application, it explicitly closes any open cursor during the execution of that program. (However, when this program terminates, any cursors remaining open are implicitly closed.)

Explicit Data Locking

Oracle always performs necessary locking to ensure data concurrency, integrity, and statement-level read consistency. However, options are available to override the default locking mechanisms. Situations where it would be advantageous to override the default locking of Oracle include the following:

- An application desires transaction-level read consistency or “repeatable reads”—transactions must query a consistent set of data for the duration of the transaction, knowing that the data has not been changed by any other transactions of the system. Transaction-level read consistency can be achieved by using explicit locking, read-only transactions, serializable transactions, or overriding default locking for the system.
- An application requires a transaction to have exclusive access to a resource. To proceed with its statements, the transaction with exclusive access to a resource does not have to wait for other transactions to complete.

The automatic locking mechanisms can be overridden at two different levels:

transaction level	Transactions including the following SQL statements override Oracle’s default locking: the LOCK TABLE command, the SELECT command including the FOR UPDATE clause, and the SET TRANSACTION command with the READ ONLY or ISOLATION LEVEL SERIALIZABLE options. Locks acquired by these statements are released after the transaction is committed or rolled back.
system level	An instance can be started with non-default locking by adjusting the initialization parameters SERIALIZABLE and ROW LOCKING.

The following sections describe each option available for overriding the default locking of Oracle. The initialization parameter DML_LOCKS determines the maximum number of DML locks allowed (see the *Oracle7 Server Reference* manual for a discussion of parameters). The default value should be sufficient; however, if you are using additional manual locks, you may need to increase this value.



Warning: If you override the default locking of Oracle at any level, be sure that the overriding locking procedures operate correctly; that is, be sure that data integrity is guaranteed, data concurrency is acceptable, and deadlocks are not possible or are appropriately handled.

Explicitly Acquiring Table Locks

A transaction explicitly acquires the specified table locks when a LOCK TABLE statement is executed. A LOCK TABLE statement manually overrides default locking. When a LOCK TABLE statement is issued on a view, the underlying base tables are locked. The following statement acquires exclusive table locks for the EMP and DEPT tables on behalf of the containing transaction:

```
LOCK TABLE emp, dept
    IN EXCLUSIVE MODE NOWAIT;
```

You can specify several tables or views to lock in the same mode; however, only a single lock mode can be specified per LOCK TABLE statement.

Note: When a table is locked, all rows of the table are locked. No other user can modify the table.

You can also indicate if you do or do not want to wait to acquire the lock. If you specify the NOWAIT option, you only acquire the table lock if it is immediately available. Otherwise an error is returned to notify that the lock is not available at this time. In this case, you can attempt to lock the resource at a later time. If NOWAIT is omitted, the transaction does not proceed until the requested table lock is acquired. If the wait for a table lock is excessive, you might want to cancel the lock operation and retry at a later time; you can code this logic into your applications.

Note: A distributed transaction waiting for a table lock can timeout waiting for the requested lock if the elapsed amount of time reaches the interval set by the initialization parameter DISTRIBUTED_LOCK_TIMEOUT. Because no data has been modified, no actions are necessary as a result of the time-out. Your application should proceed as if a deadlock has been encountered. For more information on distributed transactions, refer to *Oracle7 Server Distributed Systems, Volume I*.

The following paragraphs provide guidance on when it can be advantageous to acquire each type of table lock using the LOCK TABLE command.

ROW SHARE and ROW EXCLUSIVE

```
LOCK TABLE table IN ROW SHARE MODE;
```

```
LOCK TABLE table IN ROW EXCLUSIVE MODE;
```

Row share and row exclusive table locks offer the highest degree of concurrency. Conditions that possibly warrant the explicit acquisition of a row share or row exclusive table lock include the following:

- Your transaction needs to prevent another transaction from acquiring an intervening share, share row, or exclusive table lock for a table before the table can be updated in your transaction. If another transaction acquires an intervening share, share row, or exclusive table lock, no other transactions can update the table until the locking transaction commits or rolls back.
- Your transaction needs to prevent a table from being altered or dropped before the table can be modified later in your transaction.

SHARE

```
LOCK TABLE table IN SHARE MODE;
```

Share table locks are rather restrictive data locks. The following conditions could warrant the explicit acquisition of a share table lock:

- Your transaction only queries the table and requires a consistent set of the table's data for the duration of the transaction (that is, requires transaction-level read consistency for the locked table).
- It is acceptable if other transactions attempting to update the locked table concurrently must wait until all transactions with the share table locks commit or roll back.
- It is acceptable to allow other transactions to acquire concurrent share table locks on the same table, also allowing them the option of transaction-level read consistency.



Warning: Your transaction may or may not update the table later in the same transaction. However, if multiple transactions concurrently hold share table locks for the same table, no transaction can update the table (even if row locks are held as the result of a `SELECT . . . FOR UPDATE` statement). Therefore, if concurrent share table locks on the same table are common, updates cannot proceed and deadlocks will be common. In this case, use share row exclusive or exclusive table locks instead.

For example, assume that two tables, EMP and BUDGET, require a consistent set of data in a third table, DEPT. That is, for a given department number, you want to update the information in both of these tables, and ensure that no new members are added to the department between these two transactions.

Although this scenario is quite rare, it can be accommodated by locking the DEPT table in SHARE MODE, as shown in the following example. Because the DEPT table is not highly volatile, few, if any, users would need to update it while it was locked for the updates to EMP and BUDGET.

```
LOCK TABLE dept IN SHARE MODE
UPDATE EMP
  SET sal = sal * 1.1
  WHERE deptno IN
    (SELECT deptno FROM dept WHERE loc = 'DALLAS')
UPDATE budget
  SET totalsal = totalsal * 1.1
  WHERE deptno IN
    (SELECT deptno FROM dept WHERE loc = 'DALLAS')

COMMIT /* This releases the lock */
```

SHARE ROW EXCLUSIVE

```
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE;
```

Conditions that warrant the explicit acquisition of a share row exclusive table lock include the following:

- Your transaction requires both transaction-level read consistency for the specified table and the ability to update the locked table.
- You are not concerned about explicit row locks being obtained (that is, via `SELECT . . . FOR UPDATE`) by other transactions, which may or may not make `UPDATE` and `INSERT` statements in the locking transaction wait to update the table (that is, deadlocks might be observed).
- You only want a single transaction to have the above behavior.

EXCLUSIVE `LOCK TABLE table IN EXCLUSIVE MODE;`

Conditions that warrant the explicit acquisition of an exclusive table lock include the following:

- Your transaction requires immediate update access to the locked table. Therefore, if your transaction holds an exclusive table lock, other transactions cannot lock specific rows in the locked table.
- Your transaction also observes transaction-level read consistency for the locked table until the transaction is committed or rolled back.
- You are not concerned about low levels of data concurrency, making transactions that request exclusive table locks wait in line to update the table sequentially.

Privileges Required

You can automatically acquire any type of table lock on tables in your schema; however, to acquire a table lock on a table in another schema, you must have the LOCK ANY TABLE system privilege or any object privilege (for example, SELECT or UPDATE) for the table.

Explicitly Acquiring Row Locks

You can override default locking with a SELECT statement that includes the FOR UPDATE clause. SELECT ... FOR UPDATE is used to acquire exclusive row locks for selected rows (as an UPDATE statement does) in anticipation of actually updating the selected rows.

You can use a SELECT ... FOR UPDATE statement to lock a row without actually changing it. For example, several triggers in Chapter 9 show how to implement referential integrity. In the EMP_DEPT_CHECK trigger (see page 9 – 27), the row that contains the referenced parent key value is locked to guarantee that it remains for the duration of the transaction; if the parent key is updated or deleted, referential integrity would be violated.

SELECT ... FOR UPDATE statements are often used by interactive programs that allow a user to modify fields of one or more specific rows (which might take some time); row locks on the rows are acquired so that only a single interactive program user is updating the rows at any given time.

If a `SELECT ... FOR UPDATE` statement is used when defining a cursor, the rows in the return set are locked before the first fetch, when the cursor is opened; rows are not individually locked as they are fetched from the cursor. Locks are only released when the transaction that opened the cursor is committed or rolled back; locks are not released when a cursor is closed.

Each row in the return set of a `SELECT ... FOR UPDATE` statement is locked individually; the `SELECT ... FOR UPDATE` statement waits until the other transaction releases the conflicting row lock. Therefore, if a `SELECT ... FOR UPDATE` statement locks many rows in a table and the table experiences reasonable update activity, it would most likely improve performance if you instead acquired an exclusive table lock.

When acquiring row locks with `SELECT ... FOR UPDATE`, you can indicate if you do or do not want to wait to acquire the lock. If you specify the `NOWAIT` option, you only acquire the row lock if it is immediately possible. Otherwise, an error is returned to notify you that the lock is not possible at this time. In this case, you can attempt to lock the row later. If `NOWAIT` is omitted, the transaction does not proceed until the requested row lock is acquired. If the wait for a row lock is excessive, users might want to cancel the lock operation and retry later; you can code such logic into your applications.

As described on page 3 – 12, a distributed transaction waiting for a row lock can timeout waiting for the requested lock if the elapsed amount of time reaches the interval set by the initialization parameter `DISTRIBUTED_LOCK_TIMEOUT`.

SERIALIZABLE and ROW_LOCKING Parameters

When an instance is started, two initialization parameters determine how the instance handles locking: `SERIALIZABLE` and `ROW_LOCKING`. By default, `SERIALIZABLE` is set to `FALSE` and `ROW_LOCKING` is set to `ALWAYS`. In almost every case, these parameters should not be altered. They are provided for sites that must run in ANSI/ISO compatible mode, or that want to use applications written to run with earlier versions of Oracle. Only these sites should consider altering these parameters, as there is a significant performance degradation caused by using other than the defaults. For detailed explanations of these parameters, see the *Oracle7 Server Reference manual*.

The settings for these parameters should be changed only when an instance is shut down. If multiple instances are accessing a single database, all instances should use the same setting for these parameters.

Summary of Non-Default Locking Options

Two initialization parameters are available to change the way locking occurs at the system level. As a result, there are three global settings other than the default. Table 3 – 1 summarizes the non-default settings and why you might choose to run your database system in a non-default way.

Case	Description	SERIALIZABLE	ROW_LOCKING
1	Equivalent to Version 5 and earlier Oracle releases (no concurrent inserts, updates, or deletes in a table).	FALSE	INTENT
2	ANSI compatible.	TRUE	ALWAYS
3	ANSI compatible, with table-level locking (no concurrent inserts, updates, or deletes in a table).	TRUE	INTENT

Table 3 – 1 Summary of Non-Default Locking Options

Table 3 – 2 illustrates the difference in locking behavior resulting from the three possible settings of the initialization parameters, as shown in Table 3 – 1.

STATEMENT	CASE 1		CASE 2		CASE 3	
	row	table	row	table	row	table
SELECT	–	–	–	S	–	S
INSERT	X	SRX	X	RX	X	SRX
UPDATE	X	SRX	X	SRX	X	SRX
DELETE	X	SRX	X	SRX	X	SRX
SELECT . . .FOR UPDATE	X	RS	X	S	X	S
LOCK TABLE . . . IN . .	RS	RS	RS	RS	RS	RS
ROW SHARE MODE						
ROW EXCLUSIVE MODE						
SHARE MODE						
SHARE ROW EXCLUSIVE MODE	SRX	SRX	SRX	SRX	SRX	SRX
EXCLUSIVE MODE	X	X	X	X	X	X
DDL statements	–	X	–	X	–	X

Table 3 – 2 Non-default Locking Behavior

Creating User Locks

You can use Oracle Lock Management services for your applications. It is possible to request a lock of a specific mode, give it a unique name recognizable in another procedure in the same or another instance, change the lock mode, and release it. Because a reserved user lock is the same as an Oracle lock, it has all the functionality of an Oracle lock, such as deadlock detection. Be certain that any user locks used in distributed transactions are released upon COMMIT, or an undetected deadlock may occur.

The DBMS_LOCK Package

The Oracle Lock Management services are available through procedures in the DBMS_LOCK package. Table 3 – 3 summarizes the procedures available in the DBMS_LOCK package.

Function/Procedure	Description	Refer to Page
ALLOCATE_UNIQUE	Allocate a unique lock ID to a named lock.	3 – 20
REQUEST	Request a lock of a specific mode.	3 – 21
CONVERT	Convert a lock from one mode to another.	3 – 23
RELEASE	Release a lock.	3 – 24
SLEEP	Put a procedure to sleep for a specified time.	3 – 25

Table 3 – 3 DBMS_LOCK Package Functions and Procedures

User locks never conflict with Oracle locks because they are identified with the prefix “UL”. You can view these locks using the Server Manager lock monitor screen or the appropriate fixed views.

User locks are automatically released when a session terminates.



Warning: This implementation does not efficiently support more than a few hundred locks per session. Oracle **strongly** recommends that you develop a standard convention be developed for using these user locks. This avoids conflicts among procedures trying to use the same locks. For example, you might want to include your company name as part of the lock name to ensure that your lock names do not conflict with lock names used in any Oracle supplied applications.

Security

There might be operating system-specific limits on the maximum number of total locks available. This **must** be considered when using locks or making this package available to other users. Consider granting the EXECUTE privilege only to specific users or roles. A better alternative would be to create a cover package limiting the number of locks used and grant EXECUTE privilege to specific users. An example of a cover package is documented in the DBMSLOCK.SQL package specification file. (See the commented-out package LOCK_100_TO_200.)

Creating the DBMS_LOCK Package

To create the DBMS_LOCK package, submit the DBMSLOCK.SQL and PRVTLOCK.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script. See page 7 – 39 for information on granting the necessary privileges to users who will be executing this package.

ALLOCATE_UNIQUE Procedure

Lock identifiers are used to allow applications to coordinate their use of locks. User-assigned lock identifiers can be a number in the range of 0 to 1073741823, or locks can be identified by name. If you choose to identify locks by name, you can use ALLOCATE_UNIQUE to generate a unique lock identification number for these named locks.



Warning: Named user locks may be less efficient, as Oracle uses SQL to determine the lock associated with a given name.

The parameters for the ALLOCATE_UNIQUE procedure are described in Table 3 – 4. The syntax for this procedure is shown below.

```
DBMS_LOCK.ALLOCATE_UNIQUE(lockname          IN VARCHAR2,
                           lockhandle        OUT VARCHAR2,
                           expiration_secs    IN INTEGER
                           DEFAULT 864000);
```

Parameter	Description
LOCKNAME	<p>Specify the name of the lock for which you want to generate a unique ID. The first session to call <code>ALLOCATE_UNIQUE</code> with a new lock name causes a unique lock ID to be generated and stored in the <code>DBMS_LOCK_ALLOCATED</code> table. The handle to this ID is then returned for this call, and all subsequent calls (usually by other sessions). Lock IDs assigned by <code>ALLOCATE_UNIQUE</code> are in the range of 1073741824 to 1999999999.</p> <p>Do not use lock names beginning with <code>ORA\$</code>; these names are reserved for products supplied by Oracle Corporation.</p>
LOCKHANDLE	<p>Returns to the caller the handle to the lock ID generated by <code>ALLOCATE_UNIQUE</code>. You can use this handle in subsequent calls to <code>REQUEST</code>, <code>CONVERT</code>, and <code>RELEASE</code>. <code>LOCKHANDLE</code> can be up to <code>VARCHAR2(128)</code>.</p> <p>A handle is returned instead of the actual lock ID to reduce the chance that a programming error can accidentally create an incorrect, but valid, lock ID. This provides better isolation between different applications that are using this package.</p> <p>All sessions using a lock handle returned by <code>ALLOCATE_UNIQUE</code> using the same lock name are referring to the same lock. Different sessions can have different lock handles for the same lock, so do not pass lock handles from one session to another.</p>
EXPIRATION_SECS	<p>Specify the number of seconds to wait after the last <code>ALLOCATE_UNIQUE</code> has been performed on a given lock, before allowing that lock to be deleted from the <code>DBMS_LOCK_ALLOCATED</code> table. The default waiting period is 10 days. You should not delete locks from this table. Subsequent calls to <code>ALLOCATE_UNIQUE</code> may delete expired locks to recover space.</p>

Table 3 – 4 DBMS_LOCK.ALLOCATE_UNIQUE Procedure Parameters

REQUEST Function

To request a lock with a given mode, use the `REQUEST` function. `REQUEST` is an overloaded function that accepts either a user-defined lock identifier, or the lock handle returned by the `ALLOCATE_UNIQUE` procedure.

The parameters for the `REQUEST` function are described in Table 3 – 5 and the possible return values and their meanings are described in Table 3 – 6. The syntax for this function is shown below.


```
DBMS_LOCK.REQUEST(id                IN INTEGER ||
                  lockhandle         IN VARCHAR2,
                  lockmode           IN INTEGER DEFAULT X_MODE,
                  timeout             IN INTEGER DEFAULT MAXWAIT,
                  release_on_commit  IN  BOOLEAN DEFAULT FALSE,
RETURN INTEGER;
```

The default values, such as X_MODE and MAXWAIT, are defined in the DBMS_LOCK package specification. See the package specification, available on-line, for the current default values.

Parameter	Description
ID or LOCKHANDLE	Specify the user assigned lock identifier, from 0 to 1073741823, or the lock handle, returned by ALLOCATE_UNIQUE, of the lock whose mode you want to change.
LOCKMODE	<p>Specify the mode that you are requesting for the lock. The available modes and their associated integer identifiers are listed below. The abbreviations for these locks, as they appear in the V\$ views and Server Manager monitors are shown in parentheses.</p> <p>1 – null mode 2 – row share mode (ULRS) 3 – row exclusive mode (ULRX) 4 – share mode (ULS) 5 – share row exclusive mode (ULRSX) 6 – exclusive mode (ULX)</p> <p>Each of these lock modes is explained in the <i>Oracle7 Server Concepts</i> manual.</p>
TIMEOUT	Specify the number of seconds to continue trying to grant the lock. If the lock cannot be granted within this time period, the call returns a value of 1 (timeout).
RE- LEASE_ON_COM- MIT	Set this parameter to TRUE to release the lock on commit or rollback. Otherwise, the lock is held until it is explicitly released or until the end of the session.

Table 3 – 5 DBMS_LOCK.REQUEST Function Parameters

Return Value	Description
0	success
1	timeout
2	deadlock
3	parameter error
4	already own lock specified by ID or LOCKHANDLE
5	illegal lock handle

Table 3 – 6 DBMS_LOCK.REQUEST Function Return Values

CONVERT Function

To convert a lock from one mode to another, use the CONVERT function. CONVERT is an overloaded function that accepts either a user-defined lock identifier, or the lock handle returned by the ALLOCATE_UNIQUE procedure.

The parameters for the CONVERT function are described in Table 3 – 7 and the possible return values and their meanings are described in Table 3 – 8. The syntax for this function is shown below.

```
DBMS_LOCK.CONVERT(  
    id          IN INTEGER ||  
    lockhandle  IN VARCHAR2,  
    lockmode    IN INTEGER,  
    timeout     IN NUMBER DEFAULT MAXWAIT)  
RETURN INTEGER;
```

Parameter	Description
ID or LOCKHANDLE	Specify the user assigned lock identifier, from 0 to 1073741823, or the lock handle, returned by ALLOCATE_UNIQUE, of the lock whose mode you want to change.
LOCKMODE	<p>Specify the new mode that you want to assign to the given lock. The available modes and their associated integer identifiers are listed below. The abbreviations for these locks, as they appear in the V\$ views and Server Manager monitors are shown in parentheses.</p> <p>1 – null mode 2 – row share mode (ULRS) 3 – row exclusive mode (ULRX) 4 – share mode (ULS) 5 – share row exclusive mode (ULRSX) 6 – exclusive mode (ULX)</p> <p>Each of these lock modes is explained in the <i>Oracle7 Server Concepts</i> manual.</p>
TIMEOUT	Specify the number of seconds to continue trying to change the lock mode. If the lock cannot be converted within this time period, the call returns a value of 1 (timeout).

Table 3 – 7 DBMS_LOCK.CONVERT Function Parameters

Return Value	Description
0	success
1	timeout
2	deadlock
3	parameter error
4	don't own lock specified by ID or LOCKHANDLE
5	illegal lock handle

Table 3 – 8 DBMS_LOCK.CONVERT Function Return Values

RELEASE Function

To explicitly release a lock previously acquired using the REQUEST function, use the RELEASE function. Locks are automatically released at the end of a session. RELEASE is an overloaded function that accepts either a user-defined lock identifier, or the lock handle returned by the ALLOCATE_UNIQUE procedure.

The parameters for the RELEASE function are described in Table 3 – 9 and the possible return values and their meanings are described in Table 3 – 10. The syntax for this function is shown below.

```
DBMS_LOCK.RELEASE(id          IN INTEGER)
RETURN INTEGER;
DBMS_LOCK.RELEASE(lockhandle IN VARCHAR2)
RETURN INTEGER;
```

Parameter	Description
ID or LOCKHANDLE	Specify the user–assigned lock identifier, from 0 to 1073741823, or the lock handle, returned by ALLOCATE_UNIQUE, of the lock that you want to release.

Table 3 – 9 DBMS_LOCK.RELEASE Function Parameter

Return Value	Description
0	success
3	parameter error
4	do not own lock specified by ID or LOCKHANDLE
5	illegal lock handle

Table 3 – 10 DBMS_LOCK.RELEASE Function Return Values

SLEEP Procedure

To suspend the session for a given period of time, use the SLEEP procedure.

The parameters for the SLEEP procedure are described in Table 3 – 11. The syntax for the SLEEP procedure is shown below.

```
DBMS_LOCK.SLEEP(seconds IN NUMBER);
```

Parameter	Description
SECONDS	Specify the amount of time, in seconds, to suspend the session. The smallest increment can be entered in hundredths of a second; for example, 1.95 is a legal time value.

Table 3 – 11 DBMS_LOCK.SLEEP Procedure Parameters

Sample User Locks

Some uses of user locks are:

- providing exclusive access to a device, such as a terminal
- providing application-level enforcement of read locks
- detect when a lock is released and cleanup after the application
- synchronizing applications and enforce sequential processing

The following Pro*COBOL precompiler example shows how locks can be used to ensure that there are no conflicts when multiple people need to access a single device.

```
*****
* Print Check
* Any cashier may issue a refund to a customer returning goods.
* Refunds under $50 are given in cash, above that by check.
* This code prints the check. The one printer is opened by all
* the cashiers to avoid the overhead of opening and closing it
* for every check. This means that lines of output from multiple
* cashiers could become interleaved if we don't ensure exclusive
* access to the printer. The DBMS_LOCK package is used to
* ensure exclusive access.
*****
CHECK-PRINT
*
*   Get the lock "handle" for the printer lock.
MOVE "CHECKPRINT" TO LOCKNAME-ARR.
MOVE 10 TO LOCKNAME-LEN.
EXEC SQL EXECUTE
    BEGIN DBMS_LOCK.ALLOCATE_UNIQUE ( :LOCKNAME, :LOCKHANDLE );
    END; END-EXEC.
*
*   Lock the printer in exclusive mode (default mode).
EXEC SQL EXECUTE
    BEGIN DBMS_LOCK.REQUEST ( :LOCKHANDLE );
    END; END-EXEC.
*   We now have exclusive use of the printer, print the check.

...

*
*   Unlock the printer so other people can use it
*
EXEC SQL EXECUTE
    BEGIN DBMS_LOCK.RELEASE ( :LOCKHANDLE );
    END; END-EXEC.
```

Viewing and Monitoring Locks

Oracle provides two facilities to display locking information for ongoing transactions within an instance:

Server Manager Monitors (Lock and Latch Monitors)	The Monitor feature of Server Manager provides two monitors for displaying lock information of an instance. Refer to the <i>Oracle7 Server Manager User's Guide</i> for complete information about the Server Manager monitors.
UTLLOCKT.SQL	The UTLLOCKT.SQL script displays a simple character lock wait-for graph in tree structured fashion. Using any <i>ad hoc</i> SQL tool (such as SQL*Plus) to execute the script, it prints the sessions in the system that are waiting for locks and the corresponding blocking locks. The location of this script file is operating system dependent. (You must have run the CATBLOCK.SQL script before using UTLLOCKT.SQL.)

Concurrency Control Using Serializable Transactions

By default, the Oracle Server permits concurrently executing transactions to modify, add, or delete rows in the same table, and in the same data block. Changes made by one transaction are not seen by another concurrent transaction until the transaction that made the changes commits.

If a transaction (A) attempts to update or delete a row that has been locked by another transaction B (by way of a DML or SELECT FOR UPDATE statement), then A's DML command blocks until B commits or rolls back. Once B commits, transaction A can see changes that B has made to the database.

For most applications, this concurrency model is the appropriate one. In some cases, however, it is advantageous to allow transactions to be serializable. Serializable transactions must execute in such a way that they appear to be executing one at a time (serially), rather than concurrently. In other words, concurrent transactions executing in serialized mode are only permitted to make database changes that they could have made if the transactions were scheduled to run one after the other.

The ANSI/ISO SQL standard SQL92 defines three possible kinds of transaction interaction, and four levels of isolation that provide increasing protection against these interactions. These interactions and isolation levels are summarized in Table 3 – 12.

Isolation Level	Dirty Read (1)	Non-Repeatable Read (2)	Phantom Read (3)
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not possible	Possible	Possible
REPEATABLE READ	Not possible	Not possible	Possible
SERIALIZABLE	Not possible	Not possible	Not possible

Notes: (1) A transaction can read uncommitted data changed by another transaction.
(2) A transaction re-reads data committed by another transaction and sees the new data.
(3) A transaction can re-execute a query, and discover new rows inserted by another committed transaction.

Table 3 – 12 ANSI Isolation Levels

The behavior of Oracle with respect to these isolation levels is summarized below.

READ UNCOMMITTED	Oracle never permits “dirty reads.” This is not required for high throughput with Oracle.
READ COMMITTED	Oracle meets the READ COMMITTED isolation standard. This is the default mode for all Oracle applications. Note that since an Oracle query only sees data that was committed at the beginning of the query (the snapshot time), Oracle offers more consistency than actually required by the ANSI/ISO SQL92 standards for READ COMMITTED isolation.
REPEATABLE READ	Oracle does not support this isolation level, except as provided by SERIALIZABLE.
SERIALIZABLE	You can set this isolation level using the SET TRANSACTION command or the ALTER SESSION command, as described on page 3 – 30.

Serializable Transaction Interaction

Figure 3 – 2 shows how a serializable transaction (Transaction B) interacts with another transaction (A, which can be either SERIALIZABLE or READ COMMITTED).

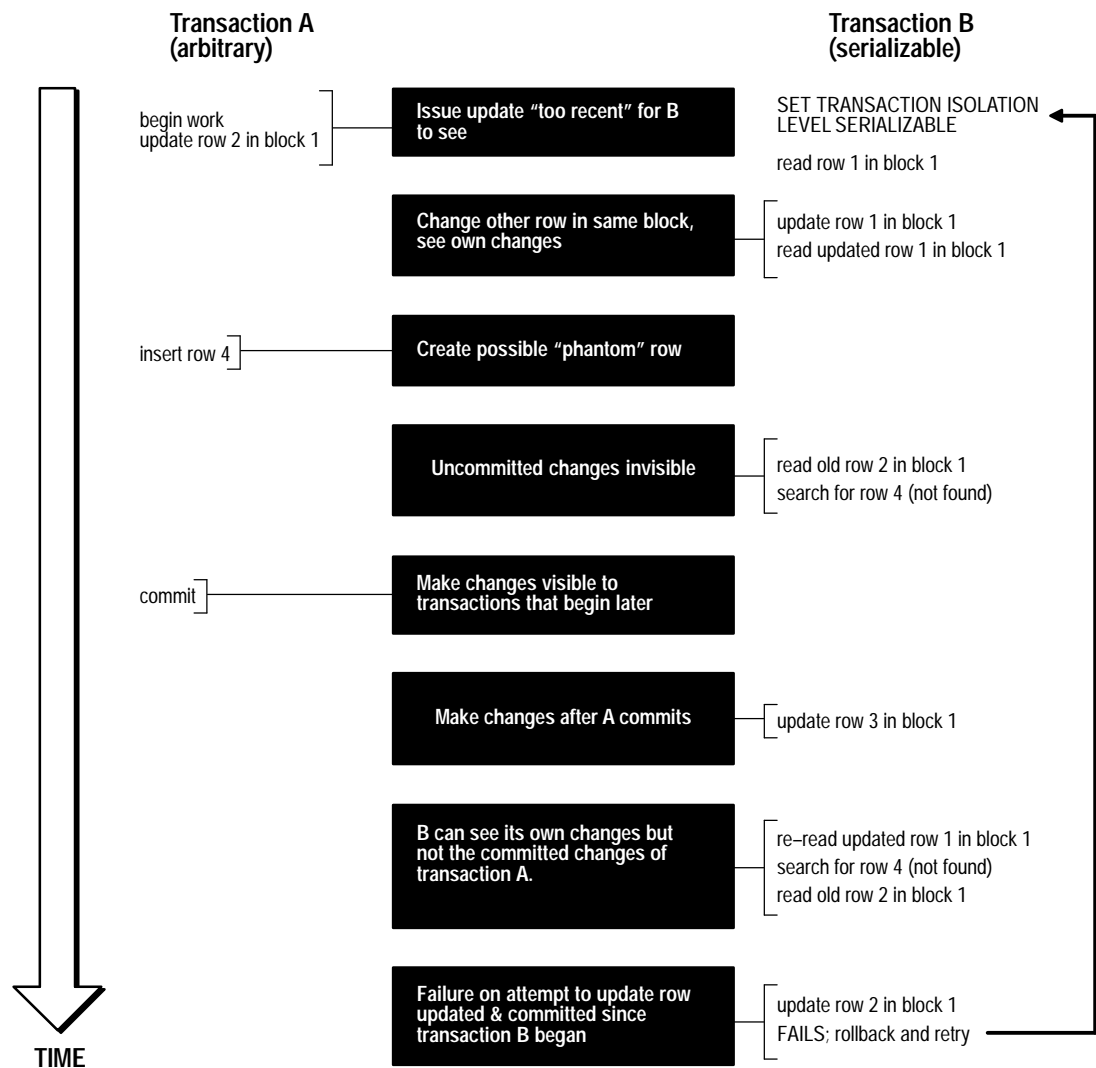


Figure 3 – 2 Time Line for Two Transactions

When a serializable transaction fails with an ORA-08177 error (“cannot serialize access”), the application can take any of several actions:

- commit the work executed to that point
- execute additional, different, statements, perhaps after rolling back to a prior savepoint in the transaction
- roll back the entire transaction and try it again

Oracle stores control information in each data block to manage access by concurrent transactions. To use the `SERIALIZABLE` isolation level, you must use the `INITRANS` clause of the `CREATE TABLE` or `ALTER TABLE` command to set aside storage for this control information. To use serializable mode, `INITRANS` must be set to at least 3.

Setting the Isolation Level

You can change the isolation level of a transaction using the `ISOLATION LEVEL` clause of the `SET TRANSACTION` command. The `SET TRANSACTION` command must be the first command issued in a transaction. If it is not, the following error is issued:

```
ORA-01453: SET TRANSACTION must be first statement of transaction
```

Use the `ALTER SESSION` command to set the transaction isolation level on a session-wide basis. See the *Oracle7 Server SQL Reference* for the complete syntax of the `SET TRANSACTION` and `ALTER SESSION` commands.

The INITRANS Parameter

Oracle stores control information in each data block to manage access by concurrent transactions. Therefore, if you set the transaction isolation level to serializable, you must use the `ALTER TABLE` command to set `INITRANS` to at least 3. This parameter will cause Oracle to allocate sufficient storage in each block to record the history of recent transactions that accessed the block. Higher values should be used for tables that will undergo many transactions updating the same blocks.

Referential Integrity and Serializable Transactions

Because Oracle7 does not use read locks, even in `SERIALIZABLE` transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level should not assume that the data they read will not change during the execution of the transaction (even though such changes are not visible to the transaction). Database inconsistencies can result unless such application-level consistency checks are coded carefully, even when using `SERIALIZABLE` transactions. Note, however, that the examples shown in this section are applicable for both `READ COMMITTED` and `SERIALIZABLE` transactions.

Figure 3 – 3 two different transactions that perform application-level checks to maintain the referential integrity parent/child relationship between two tables. One transaction reads the parent table to determine that a row with a specific primary key value exists before inserting corresponding child rows. The other transaction checks to see that no corresponding detail rows exist before proceeding to delete a parent row. In this case, both transactions assume (but do not ensure) that data they read will not change before the transaction completes.

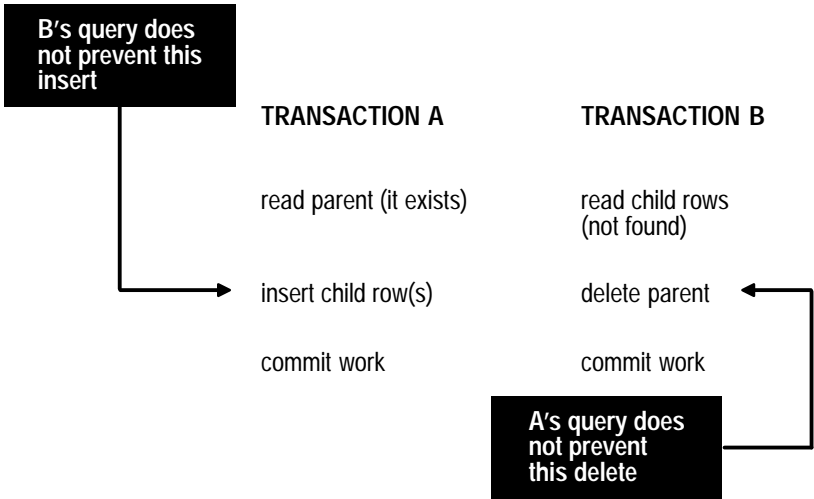


Figure 3 – 3 Referential Integrity Checks

Note that the read issued by transaction A does not prevent transaction B from deleting the parent row. Likewise, transaction B's query for child rows does not prevent the insertion of child rows by transaction A. Therefore the above scenario leaves in the database a child row with no corresponding parent row. This result would occur even if both A and B are `SERIALIZABLE` transactions, because neither transaction prevents the other from making changes in the data it reads to check consistency.

As this example illustrates, for some transactions, application developers must specifically ensure that the data read by one transaction is not concurrently written by another. This requires a greater degree of transaction isolation than defined by SQL92 `SERIALIZABLE` mode.

Using `SELECT FOR UPDATE`

Fortunately, it is straightforward in Oracle7 to prevent the anomaly described above. Transaction A can use `SELECT FOR UPDATE` to query and lock the parent row and thereby prevent transaction B from deleting the row. Transaction B can prevent transaction A from gaining access to the parent row by reversing the order of its processing steps. Transaction B first deletes the parent row, and then rolls back if its subsequent query detects the presence of corresponding rows in the child table.

Referential integrity can also be enforced in Oracle7 using database triggers, instead of a separate query as in transaction A above. For example, an `INSERT` into the child table can fire a `PRE-INSERT` row-level trigger to check for the corresponding parent row. The trigger queries the parent table using `SELECT FOR UPDATE`, ensuring that parent row (if it exists) will remain in the database for the duration of the transaction inserting the child row. If the corresponding parent row does not exist, the trigger rejects the insert of the child row.

SQL statements issued by a database trigger execute in the context of the SQL statement that caused the trigger to fire. All SQL statements executed within a trigger see the database in the same state as the triggering statement. Thus, in a `READ COMMITTED` transaction, the SQL statements in a trigger see the database as of the beginning of the triggering statement's execution, and in a transaction executing in `SERIALIZABLE` mode, the SQL statements see the database as of the beginning of the transaction. In either case, the use of `SELECT FOR UPDATE` by the trigger will correctly enforce referential integrity as explained above.

READ COMMITTED and SERIALIZABLE Isolation

Oracle7 gives the application developer a choice of two transaction isolation levels with different characteristics. Both the READ COMMITTED and SERIALIZABLE isolation levels provide a high degree of consistency and concurrency. Both levels provide the contention-reducing benefits of Oracle's "read consistency" multi-version concurrency control model and exclusive row-level locking implementation, and are designed for real-world application deployment. The rest of this section compares the two isolation modes and provides information helpful in choosing between them.

Transaction Set Consistency

A useful way to describe the READ COMMITTED and SERIALIZABLE isolation levels in Oracle7 is to consider the following:

- a collection of database tables (or any set of data)
- a particular sequence of reads of rows in those tables
- the set of transactions committed at any particular time

An operation (a query or a transaction) is "transaction set consistent" if all its reads return data written by the same set of committed transactions. In an operation that is not transaction set consistent, some reads reflect the changes of one set of transactions, and other reads reflect changes made by other transactions. An operation that is not transaction set consistent in effect sees the database in a state that reflects no single set of committed transactions.

Oracle7 provides transactions executing in READ COMMITTED mode with transaction set consistency on a per-statement basis (since all rows read by a query must have been committed before the query began). Similarly, Oracle7 SERIALIZABLE mode provides transaction set consistency on a per-transaction basis, since all statements in a SERIALIZABLE transaction execute with respect to an image of the database as of the beginning of the transaction.

In other database systems (unlike in Oracle7), a single query run in READ COMMITTED mode provides results that are not transaction set consistent. The query is not transaction set consistent because it may see only a subset of the changes made by another transaction. This means, for example, that a join of a master table with a detail table could see a master record inserted by another transaction, but not the corresponding details inserted by that transaction, or vice versa. Oracle7's READ COMMITTED mode will not experience this effect, and so provides a greater degree of consistency than read-locking systems.

In read-locking systems, at the cost of preventing concurrent updates, SQL92 REPEATABLE READ isolation provides transaction set consistency at the statement level, but not at the transaction level. The absence of phantom protection means two queries issued by the same transaction can see data committed by different sets of other transactions. Only the throughput-limiting and deadlock-susceptible SERIALIZABLE mode in these systems provides transaction set consistency at the transaction level.

Functionality Comparison Summary

Table 3 – 13 summarizes key similarities and differences between READ COMMITTED and SERIALIZABLE transactions as implemented in Oracle7 release 7.3.

	READ COMMITTED	SERIALIZABLE
Dirty write	Not possible	Not possible
Dirty read	Not possible	Not possible
Non-repeatable read	Possible	Not possible
Phantoms	Possible	Not possible
Compliant with ANSI/ISO SQL 92	Yes	Yes
Read snapshot time	Statement	Transaction
Transaction set consistency	Statement level	Transaction level
Row-level locking	Yes	Yes
Readers block writers	No	No
Writers block readers	No	No
Different-row writers block writers	No	No
Same-row writers block writers	Yes	Yes
Waits for blocking transaction	Yes	Yes
Subject to "can't serialize access" error	No	Yes
Error after blocking transaction aborts	No	No
Error after blocking transaction commits	No	Yes

Table 3 – 13 Read Committed vs. Serializable Transactions

Choosing an Isolation Level

Application designers and developers should choose an isolation level that is appropriate to the specific application and workload, and may choose different isolation levels for different transactions. The choice should be based on performance and consistency needs, and consideration of application coding requirements.

For environments with many concurrent users rapidly submitting transactions, designers must assess transaction performance requirements in terms of the expected transaction arrival rate and response time demands, and choose an isolation level that provides the required degree of consistency while satisfying performance expectations. Frequently, for high performance environments, the choice of isolation levels involves making a tradeoff between consistency and concurrency (transaction throughput).

Both Oracle7 isolation modes provide high levels of consistency and concurrency (and performance) through the combination of row-level locking and Oracle's multi-version concurrency control system. Because readers and writers don't block one another in Oracle7, while queries still see consistent data, both READ COMMITTED and SERIALIZABLE isolation provide a high level of concurrency for high performance, without the need for reading uncommitted ("dirty") data.

READ COMMITTED isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results (due to phantoms and non-repeatable reads) for some transactions. The SERIALIZABLE isolation level provides somewhat more consistency by protecting against phantoms and non-repeatable reads, and may be important where a read/write transaction executes a query more than once. However, SERIALIZABLE mode requires applications to check for the "can't serialize access" error, and can significantly reduce throughput in an environment with many concurrent transactions accessing the same data for update. Application logic that checks database consistency must take into account the fact reads don't block writes in either mode.

Application Tips

When a transaction runs in serializable mode, any attempt to change data that was changed by another transaction since the beginning of the serializable transaction results in the following error:

```
ORA-08177: Can't serialize access for this transaction.
```

When you get an ORA-08177 error, the appropriate action is to roll back the current transaction, and re-execute it. After a rollback, the transaction acquires a new transaction snapshot, and the DML operation is likely to succeed.

Since a rollback and repeat of the transaction is required, it is good development practice to put DML statements that might conflict with other concurrent transactions towards the beginning of your transaction, whenever possible.

Managing Schema Objects

This chapter discusses the procedures necessary to create and manage the different types of objects contained in a user's schema. The topics included are as follows:

- managing tables
- managing views
- modifying a join view
- managing sequences
- managing synonyms
- managing indexes
- managing clusters
- listing information about schema objects

Note: Procedures, functions, and packages are discussed in Chapter 7; dependency information is in Chapter 11. If you are using symmetric replication, you should refer to *Oracle7 Server Distributed Systems, Volume II* for additional information on managing schema objects, such as snapshots. If you are using Trusted Oracle, there are additional privileges required and issues to consider when managing schema objects; refer to the *Trusted Oracle7 Server Administrator's Guide*.

Managing Tables

A table is the data structure that holds data in a relational database. A table is comprised of rows and columns.

A table can represent a single entity that you want to track within your system. Such a table might represent a list of the employees within your organization or the orders placed for your company's products.

A table can also represent a relationship between two entities. Such a table could be used to portray the association between employees and their job skills or the relationship of products to orders. Within the tables, foreign keys are used to represent relationships.

Although some well designed tables might both represent an entity and describe the relationship between that entity and another entity, most tables should represent either an entity or a relationship. For example, the EMP table describes the employees in a firm, but this table also includes a foreign key column, DEPTNO, which represents the relationships of employees to departments.

The following sections explain how to create, alter, and drop tables. Some simple guidelines to follow when managing tables in your database are included; see the *Oracle7 Server Administrator's Guide* for more suggestions. You should also refer to a text on relational database or table design.

Designing Tables

You should consider the following guidelines when designing your tables:

- Use descriptive names for tables, columns, indexes, and clusters.
- Be consistent in abbreviations and in the use of singular and plural forms of table names and columns.
- Document the meaning of each table and its columns with the COMMENT command.
- Normalize each table.
- Select the appropriate datatype for each column.
- Define columns that allow nulls last, to conserve storage space.
- Cluster tables whenever appropriate, to conserve storage space and optimize performance of SQL statements.

Before creating a table, you should also determine whether to use integrity constraints. Integrity constraints can be defined on the columns of a table to enforce the business rules of your database automatically; see Chapter 6 for guidelines.

Creating Tables

To create a table, use the SQL command `CREATE TABLE`. For example, if the user `SCOTT` issues the following statement, he creates a non-clustered table named `EMP` in his schema that is physically stored in the `USERS` tablespace. Notice that integrity constraints are defined on several columns of the table.

```
CREATE TABLE emp (  
    empno      NUMBER(5) PRIMARY KEY,  
    ename      VARCHAR2(15) NOT NULL,  
    job        VARCHAR2(10),  
    mgr        NUMBER(5),  
    hiredate   DATE DEFAULT (sysdate),  
    sal        NUMBER(7,2),  
    comm       NUMBER(7,2),  
    deptno     NUMBER(3) NOT NULL  
              CONSTRAINT dept_fkey REFERENCES dept)  
  
PCTFREE 10  
PCTUSED 40  
TABLESPACE users  
STORAGE (  INITIAL 50K  
           NEXT 50K  
           MAXEXTENTS 10  
           PCTINCREASE 25 );
```

Managing the Space Usage of Data Blocks

The following sections explain how to use the `PCTFREE` and `PCTUSED` parameters to do the following:

- increase the performance of writing and retrieving a data or index segment
- decrease the amount of unused space in data blocks
- decrease the amount of row chaining between data blocks

The PCTFREE default is 10 percent; any integer from 0 to 99 is acceptable, as long as the sum of PCTFREE and PCTUSED does not exceed 100. (If PCTFREE is set to 99, Oracle puts at least one row in each block, regardless of row size. If rows are very small and blocks very large, even more than one row might fit.)

A lower PCTFREE

- reserves less room for updates to existing table rows
- allows inserts to fill the block more completely
- might save space, because the total data for a table or index is stored in fewer blocks (more rows or entries per block)
- increases processing costs because blocks frequently need to be reorganized as their free space area becomes filled with new or updated data
- potentially increases processing costs and space required if updates to rows or index entries cause rows to grow and span blocks (because UPDATE, DELETE, and SELECT statements might need to read more blocks for a given row and because chained row pieces contain references to other pieces)

A higher PCTFREE

- reserves more room for future updates to existing table rows
- might require more blocks for the same amount of inserted data (inserting fewer rows per block)
- lessens processing costs because blocks infrequently need reorganization of their free space area
- might improve update performance, because Oracle must chain row pieces less frequently, if ever

In setting PCTFREE, you should understand the nature of the table or index data. Updates can cause rows to grow. When using NUMBER, VARCHAR2, LONG, or LONG RAW, new values might not be the same size as values they replace. If there are many updates in which data values get longer, increase PCTFREE; if updates to rows do not affect the total row width, then PCTFREE can be low.

Your goal is to find a satisfactory tradeoff between densely packed data (low PCTFREE, full blocks) and good update performance (high PCTFREE, less-full blocks).

PCTFREE also affects the performance of a given user's queries on tables with uncommitted transactions belonging to other users. Assuring read consistency might cause frequent reorganization of data in blocks that have little free space.

PCTFREE for Non-Clustered Tables If the data in the rows of a non-clustered table is likely to increase in size over time, reserve space for these updates. If you do not reserve room for updates, updated rows are likely to be chained between blocks, reducing I/O performance associated with these rows.

PCTFREE for Clustered Tables The discussion for non-clustered tables also applies to clustered tables. However, if PCTFREE is reached, new rows from **any** table contained in the same cluster key go into a new data block chained to the existing cluster key.

PCTFREE for Indexes Indexes infrequently require the use of free space for updates to index data. Therefore, the PCTFREE value for index segment data blocks is normally very low (for example, 5 or less).

Specifying PCTUSED

Once the percentage of free space in a data block reaches PCTFREE, no new rows are inserted in that block until the percentage of space used falls below PCTUSED. Oracle tries to keep a data block at least PCTUSED full. The percent is of block space available for data after overhead is subtracted from total space.

The default for PCTUSED is 40 percent; any integer between 0 and 99, inclusive, is acceptable as long as the sum of PCTUSED and PCTFREE does not exceed 100.

A lower PCTUSED

- usually keeps blocks less full than a higher PCTUSED
- reduces processing costs incurred during UPDATE and DELETE statements for moving a block to the free list when the block has fallen below that percentage of usage
- increases the unused space in a database

A higher PCTUSED

- usually keeps blocks fuller than a lower PCTUSED
- improves space efficiency
- increases processing cost during INSERTs and UPDATEs

Choosing Associated PCTUSED and PCTFREE Values

If you decide not to use the default values for PCTFREE and PCTUSED, use the following guidelines.

- The sum of PCTFREE and PCTUSED must be equal to or less than 100.
- If the sum is less than 100, the ideal compromise of space utilization and I/O performance is a sum of PCTFREE and PCTUSED that differs from 100 by the percentage of space in the available block that an average row occupies. For example, assume that the data block size is 2048 bytes, minus 100 bytes of overhead, leaving 1948 bytes available for data. If an average row requires 195 bytes, or 10% of 1948, then an appropriate combination of PCTUSED and PCTFREE that sums to 90% would make the best use of database space.
- If the sum equals 100, Oracle attempts to keep no more than PCTFREE free space, and the processing costs are highest.
- Fixed block overhead is not included in the computation of PCTUSED or PCTFREE.
- The smaller the difference between 100 and the sum of PCTFREE and PCTUSED (as in PCTUSED of 75, PCTFREE of 20), the more efficient space usage is at some performance cost.

Examples of Choosing PCTFREE and PCTUSED Values

The following examples illustrate correctly specifying values for PCTFREE and PCTUSED in given scenarios.

Example 1	Scenario:	Common activity includes UPDATE statements that increase the size of the rows. Performance is important.
	Settings:	PCTFREE = 20 PCTUSED = 40
	Explanation:	PCTFREE is set to 20 to allow enough room for rows that increase in size as a result of updates. PCTUSED is set to 40 so that less processing is done during high update activity, thus improving performance.

Example 2	Scenario:	Most activity includes INSERT and DELETE statements, and UPDATE statements that do not increase the size of affected rows. Performance is important.
	Settings:	PCTFREE = 5 PCTUSED = 60
	Explanation:	PCTFREE is set to 5 because most UPDATE statements do not increase row sizes. PCTUSED is set to 60 so that space freed by DELETE statements is used relatively soon, yet the amount of processing is minimized.
Example 3	Scenario:	The table is very large; therefore, storage is a primary concern. Most activity includes read-only transactions; therefore, query performance is important.
	Settings:	PCTFREE = 5 PCTUSED = 90
	Explanation:	PCTFREE is set to 5 because UPDATE statements are rarely issued. PCTUSED is set to 90 so that more space per block is used to store table data. This setting for PCTUSED reduces the number of data blocks required to store the table's data and decreases the average number of data blocks to scan for queries, thereby increasing the performance of queries.

Privileges Required to Create a Table

To create a new table in your schema, you must have the CREATE TABLE system privilege. To create a table in another user's schema, you must have the CREATE ANY TABLE system privilege. Additionally, the owner of the table must have a quota for the tablespace that contains the table, or the UNLIMITED TABLESPACE system privilege.

Altering Tables

You might alter a table in an Oracle database for any of the following reasons:

- to add one or more new columns to the table
- to add one or more integrity constraints to a table
- to modify an existing column's definition (datatype, length, default value, and NOT NULL integrity constraint)
- to modify data block space usage parameters (PCTFREE, PCTUSED)
- to modify transaction entry settings (INITRANS, MAXTRANS)
- to modify storage parameters (NEXT, PCTINCREASE, etc.)
- to enable or disable integrity constraints associated with the table
- to drop integrity constraints associated with the table

When altering the column definitions of a table, you can only increase the length of an existing column, unless the table has no records. You can also decrease the length of a column in an empty table. For columns of datatype CHAR, increasing the length of a column might be a time consuming operation that requires substantial additional storage, especially if the table contains many rows. This is because the CHAR value in each row must be blank-padded to satisfy the new column length.

If you change the datatype (for example, from VARCHAR2 to CHAR), the data in the column does not change. However, the length of new CHAR columns might change, due to blank-padding requirements.

Use the SQL command ALTER TABLE to alter a table, as in

```
ALTER TABLE emp
  PCTFREE 30
  PCTUSED 60;
```

Altering a table has the following implications:

- If a new column is added to a table, the column is initially null. You can add a column with a NOT NULL constraint to a table only if the table does not contain any rows.
- If a view or PL/SQL program unit depends on a base table, the alteration of the base table might affect the dependent object, and always invalidates the dependent object.

Privileges Required to Alter a Table

To alter a table, the table must be contained in your schema, or you must have either the ALTER object privilege for the table or the ALTER ANY TABLE system privilege.

Dropping Tables

Use the SQL command DROP TABLE to drop a table. For example, the following statement drops the EMP table:

```
DROP TABLE emp;
```

If the table that you are dropping contains any primary or unique keys referenced by foreign keys of other tables, and you intend to drop the FOREIGN KEY constraints of the child tables, include the CASCADE option in the DROP TABLE command, as in

```
DROP TABLE emp CASCADE CONSTRAINTS;
```

Dropping a table has the following effects:

- The table definition is removed from the data dictionary. All rows of the table are then inaccessible.
- All indexes and triggers associated with the table are dropped.
- All views and PL/SQL program units that depend on a dropped table remain, yet become invalid (not usable).
- All synonyms for a dropped table remain, but return an error when used.
- All extents allocated for a non-clustered table that is dropped are returned to the free space of the tablespace and can be used by any other object requiring new extents.
- All rows corresponding to a clustered table are deleted from the blocks of the cluster.
- If the table is a master table for snapshots, Oracle does not drop the snapshots, but does drop the snapshot log. The snapshots can still be used, but they cannot be refreshed unless the table is re-created.

If you want to delete all of the rows of a table, but keep the table definition, you should use the TRUNCATE TABLE command. This command is described in the *Oracle7 Server Administrator's Guide*.

Privileges Required to Drop a Table

To drop a table, the table must be contained in your schema or you must have the DROP ANY TABLE system privilege.

Managing Views

A *view* is a logical representation of another table or combination of tables. A view derives its data from the tables on which it is based. These tables are called *base tables*. Base tables might in turn be actual tables or might be views themselves.

All operations performed on a view actually affect the base table of the view. You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables.

Views can provide a different representation (such as subsets or supersets) of the data that resides within other tables and views. Views are very powerful because they allow you to tailor the presentation of data to different types of users.

The following sections explain how to create, replace, and drop views using SQL commands.

Creating Views

Use the SQL command CREATE VIEW to create a view. You can define views with any query that references tables, snapshots, or other views; however, the query that defines a view cannot contain the ORDER BY or FOR UPDATE clauses. For example, the following statement creates a view on a subset of data in the EMP table:

```
CREATE VIEW sales_staff AS
  SELECT empno, ename, deptno
  FROM emp
  WHERE deptno = 10
  WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

The query that defines the SALES_STAFF view references only rows in department 10. Furthermore, the WITH CHECK OPTION creates the view with the constraint that INSERT and UPDATE statements issued against the view are not allowed to create or result in rows that the view cannot select. Considering the example above, the following INSERT statement successfully inserts a row into the EMP table via the SALES_STAFF view:

```
INSERT INTO sales_staff VALUES (7584, 'OSTER', 10);
```

However, the following INSERT statement is rolled back and returns an error because it attempts to insert a row for department number 30, which could not be selected using the SALES_STAFF view:

```
INSERT INTO sales_staff VALUES (7591, 'WILLIAMS', 30);
```

The following statement creates a view that joins data from the EMP and DEPT tables:

```
CREATE VIEW division1_staff AS
  SELECT ename, empno, job, dname
  FROM emp, dept
  WHERE emp.deptno IN (10, 30)
  AND emp.deptno = dept.deptno;
```

The DIVISION1_STAFF view is defined by a query that joins information from the EMP and DEPT tables. The WITH CHECK OPTION is not specified in the CREATE VIEW statement because rows cannot be inserted into or updated in a view defined with a query that contains a join that uses the WITH CHECK OPTION; see page 4 – 14 and pages 4 – 16 and following.

Expansion of Defining Queries at View Creation Time

In accordance with the ANSI/ISO standard, Oracle expands any wildcard in a top-level view query into a column list when a view is created and stores the resulting query in the data dictionary; any subqueries are left intact. The column names in an expanded column list are enclosed in quote marks to account for the possibility that the columns of the base object were originally entered with quotes and require them for the query to be syntactically correct.

As an example, assume that the DEPT view is created as follows:

```
CREATE VIEW dept AS SELECT * FROM scott.dept;
```

Oracle stores the defining query of the DEPT view as

```
SELECT "DEPTNO", "DNAME", "LOC" FROM scott.dept
```

Views created with errors do not have wildcards expanded. However, if the view is eventually compiled without errors, wildcards in the defining query are expanded.

Creating Views with Errors

Assuming no syntax errors, a view can be created (with errors) even if the defining query of the view cannot be executed. For example, if a view is created that refers to a non-existent table or an invalid column of an existing table, or if the owner of the view does not have the required privileges, the view can still be created and entered into the data dictionary.

You can only create a view with errors by using the **FORCE** option of the **CREATE VIEW** command:

```
CREATE FORCE VIEW AS ...;
```

When a view is created with errors, Oracle returns a message that indicates the view was created with errors. The status of such a view is left as **INVALID**. If conditions later change so that the query of an invalid view can be executed, the view can be recompiled and become valid. Oracle dynamically compiles the invalid view if you attempt to use it.

Privileges Required to Create a View

To create a view, you must have been granted the following privileges:

- You must have the **CREATE VIEW** system privilege to create a view in your schema or the **CREATE ANY VIEW** system privilege to create a view in another user's schema. These privileges can be acquired explicitly or via a role.
- The **owner** of the view must have been explicitly granted the necessary privileges to access all objects referenced within the definition of the view; the owner cannot have obtained the required privileges through roles. Also, the functionality of the view is dependent on the privileges of the view's owner. For example, if you (the view owner) are granted only the **INSERT** privilege for Scott's **EMP** table, you can create a view on his **EMP** table, but you can only use this view to insert new rows into the **EMP** table.
- If the view owner intends to grant access to the view to other users, the owner must have received the object privileges to the base objects with the **GRANT OPTION** or the system privileges with the **ADMIN OPTION**; if not, the view owner has insufficient privileges to grant access to the view to other users.

Replacing Views

To alter the definition of a view, you must replace the view using one of the following methods:

- A view can be dropped and then re-created. When a view is dropped, all grants of corresponding view privileges are revoked from roles and users. After the view is re-created, necessary privileges must be regranted.
- A view can be replaced by redefining it with a CREATE VIEW statement that contains the OR REPLACE option. This option is used to replace the current definition of a view but preserve the present security authorizations. For example, assume that you create the SALES_STAFF view, as given in a previous example. You also grant several object privileges to roles and other users. However, now you realize that you must redefine the SALES_STAFF view to correct the department number specified in the WHERE clause of the defining query, because it should have been 30. To preserve the grants of object privileges that you have made, you can replace the current version of the SALES_STAFF view with the following statement:

```
CREATE OR REPLACE VIEW sales_staff AS
  SELECT empno, ename, deptno
  FROM emp
  WHERE deptno = 30
  WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

Replacing a view has the following effects:

- Replacing a view replaces the view's definition in the data dictionary. All underlying objects referenced by the view are not affected.
- If previously defined but not included in the new view definition, the constraint associated with the WITH CHECK OPTION for a view's definition is dropped.
- All views and PL/SQL program units dependent on a replaced view become invalid.

Privileges Required to Replace a View

To replace a view, you must have all of the privileges needed to drop the view, as well as all of those required to create the view.

Using Views

Views can be queried in the same manner as tables. For example, to query the DIVISION1_STAFF view, enter a valid SELECT statement that references the view:

```
SELECT * FROM division1_staff;
```

ENAME	EMPNO	JOB	DNAME
-----	-----	-----	-----
CLARK	7782	MANAGER	ACCOUNTING
KING	7839	PRESIDENT	ACCOUNTING
MILLER	7934	CLERK	ACCOUNTING
ALLEN	7499	SALESMAN	SALES
WARD	7521	SALESMAN	SALES
JAMES	7900	CLERK	SALES
TURNER	7844	SALESMAN	SALES
MARTIN	7654	SALESMAN	SALES
BLAKE	7698	MANAGER	SALES

With some restrictions, rows can be inserted into, updated in, or deleted from a base table using a view. The following statement inserts a new row into the EMP table using the SALES_STAFF view:

```
INSERT INTO sales_staff
VALUES (7954, 'OSTER', 30);
```

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, rows cannot be inserted into, updated in, or deleted from the base tables using the view.
2. If a view is defined with the WITH CHECK OPTION, a row cannot be inserted into, or updated in, the base table (using the view) if the view cannot select the row from the base table.
3. If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, a row cannot be inserted into the base table using the view.
4. If the view was created by using an expression, such as DECODE(deptno, 10, 'SALES', ...), rows cannot be inserted into or updated in the base table using the view.

The constraint created by the WITH CHECK OPTION of the SALES_STAFF view only allows rows that have a department number of 10 to be inserted into, or updated in, the EMP table. Alternatively, assume that the SALES_STAFF view is defined by the following statement (that is, excluding the DEPTNO column):

```
CREATE VIEW sales_staff AS
    SELECT empno, ename
    FROM emp
    WHERE deptno = 10
    WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

Considering this view definition, you can update the EMPNO or ENAME fields of existing records, but you cannot insert rows into the EMP table via the SALES_STAFF view because the view does not let you alter the DEPTNO field. If you had defined a DEFAULT value of 10 on the DEPTNO field, you could perform inserts.

Referencing Invalid Views When a user attempts to reference an invalid view, Oracle returns an error message to the user:

```
ORA-04063: view 'view_name' has errors
```

This error message is returned when a view exists but is unusable due to errors in its query (whether it had errors when originally created or it was created successfully but became unusable later because underlying objects were altered or dropped).

Privileges Required to Use a View

To issue a query or an INSERT, UPDATE, or DELETE statement against a view, you must have the SELECT, INSERT, UPDATE, or DELETE object privilege for the view, respectively, either explicitly or via a role.

Dropping Views

Use the SQL command DROP VIEW to drop a view, as in

```
DROP VIEW sales_staff;
```

Privileges Required to Drop a View

You can drop any view contained in your schema. To drop a view in another user's schema, you must have the DROP ANY VIEW system privilege.

Modifying a Join View

The Oracle Server for release 7.3 allows you, with some restrictions, to modify views that involve joins.

In previous releases of the Oracle Server, you could not issue UPDATE, INSERT, or DELETE statements against a join view. Consider the following simple view:

```
CREATE VIEW emp_view AS
  SELECT ename, empno, deptno FROM emp;
```

This view does not involve a join operation. If you issue the SQL statement:

```
UPDATE emp_view SET ename = 'CAESAR' WHERE empno = 7839;
```

then the EMP base table that underlies the view changes, and employee 7839's name changes from KING to CAESAR in the EMP table.

However, if you create a view such as

```
CREATE VIEW emp_dept AS
  SELECT e.empno, e.ename, e.deptno, d.dname, d.loc
     FROM emp e, dept d    /* JOIN operation */
  WHERE e.deptno = d.deptno
        AND d.loc IN ('DALLAS', 'NEW YORK', 'BOSTON');
```

then in Oracle Server releases prior to 7.3 you could not modify either the EMP or the DEPT base table through this view, because it involves a join operation. A statement such as

```
UPDATE emp_dept_view SET ename = 'JOHNSON'
  WHERE ename = 'SMITH';
```

would have failed with an ORA-01732 error.

```
ORA-01732: "data manipulation operations not legal on this view"
```

Oracle7 release 7.3 allows you to modify such a view, subject to the restrictions described below.

A *modifiable join view* is a view that contains more than one table in the top-level FROM clause of the SELECT statement, and that does *not* contain any of the following:

- DISTINCT operator
- aggregate functions: AVG, COUNT, GLB, MAX, MIN, STDDEV, SUM, or VARIANCE
- set operations: UNION, UNION ALL, INTERSECT, MINUS
- GROUP BY or HAVING clauses
- START WITH or CONNECT BY clauses
- ROWNUM pseudocolumn

A further restriction on which join views are modifiable is that if a view is a join on other nested views, then the other nested views must be mergeable into the top level view. See Chapter 5 in the *Oracle7 Server Tuning* manual for more information about mergeable views.

Example Tables

The examples in this section use the familiar EMP and DEPT tables. However, the examples work only if you explicitly define the primary and foreign keys in these tables, or define unique indexes. Here are the appropriately constrained table definitions for EMP and DEPT:

```
CREATE TABLE dept (  
    deptno    NUMBER(4) PRIMARY KEY,  
    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),  
    FOREIGN KEY (DEPTNO) REFERENCES DEPT(DEPTNO));
```

You could also omit the primary and foreign key constraints listed above, and create a UNIQUE INDEX on DEPT (DEPTNO) to make the following examples work.

Key-Preserved Tables

The concept of a *key-preserved table* is fundamental to understanding the restrictions on modifying join views. A table is key preserved if every key of the table can also be a key of the result of the join. So, a key-preserved table has its keys preserved through a join.

Note: It is not necessary that the key or keys of a table be selected for it to be key preserved. It is sufficient that if the key or keys were selected, then they would also be key(s) of the result of the join.



Attention: The key-preserving property of a table does not depend on the actual data in the table. It is, rather, a property of its schema and not of the data in the table. For example, if in the EMP table there was at most one employee in each department, then DEPT.DEPTNO would be unique in the result of a join of EMP and DEPT, but DEPT would still not be a key-preserved table.

If you SELECT all rows from EMP_DEPT_VIEW defined on page 4 – 16 above, the results are

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK
7369	SMITH	20	RESEARCH	DALLAS
7876	ADAMS	20	RESEARCH	DALLAS
7902	FORD	20	RESEARCH	DALLAS
7788	SCOTT	20	RESEARCH	DALLAS
7566	JONES	20	RESEARCH	DALLAS

8 rows selected.

In this view, EMP is a key-preserved table, because EMPNO is a key of the EMP table, and also a key of the result of the join. DEPT is *not* a key-preserved table, because although DEPTNO is a key of the DEPT table, it is not a key of the join.

Rule for DML Statements on Join Views

Any UPDATE, INSERT, or DELETE statement on a join view can modify only one underlying base table.

UPDATE Statements

The following example shows an UPDATE statement that successfully modifies the EMP_DEPT view (shown on page 4 – 16):

```
UPDATE emp_dept
  SET sal = sal * 1.10
  WHERE deptno = 10;
```

The following UPDATE statement would be disallowed on the EMP_DEPT view:

```
UPDATE emp_dept
  SET loc = 'BOSTON'
  WHERE ename = 'SMITH';
```

This statement fails with an ORA-01779 error (“cannot modify a column which maps to a non key-preserved table”), because it attempts to modify the underlying DEPT table, and the DEPT table is not key preserved in the EMP_DEPT view.

In general, all modifiable columns of a join view must map to columns of a key-preserved table. If the view is defined using the WITH CHECK OPTION clause, then all join columns and all columns of repeated tables are not modifiable.

So, for example, if the EMP_DEPT view were defined using WITH CHECK OPTION, the following UPDATE statement would fail:

```
UPDATE emp_dept
  SET deptno = 10
  WHERE ename = 'SMITH';
```

The statement fails because it is trying to update a join column.

DELETE Statements

You can delete from a join view provided there is *one and only one* key-preserved table in the join.

The following DELETE statement works on the EMP_DEPT view:

```
DELETE FROM emp_dept
  WHERE ename = 'SMITH';
```

This DELETE statement on the EMP_DEPT view is legal because it can be translated to a DELETE operation on the base EMP table, and because the EMP table is the only key-preserved table in the join.

In the following view, a DELETE operation cannot be performed on the view because both E1 and E2 are key-preserved tables:

```
CREATE VIEW emp_emp AS
  SELECT e1.ename, e2.empno, deptno
  FROM emp e1, emp e2
  WHERE e1.empno = e2.empno;
```

If a view is defined using the WITH CHECK OPTION clause and the key-preserved table is repeated, then rows cannot be deleted from such a view. For example:

```
CREATE VIEW emp_mgr AS
  SELECT e1.ename, e2.ename mname
  FROM emp e1, emp e2
  WHERE e1.mgr = e2.empno
  WITH CHECK OPTION;
```

No deletion can be performed on this view because the view involves a self-join of the table that is key preserved.

INSERT Statements

The following INSERT statement on the EMP_DEPT view succeeds:

```
INSERT INTO emp_dept (ename, empno, deptno)
  VALUES ('KURODA', 9010, 40);
```

because only one key-preserved base table is being modified (EMP), and 40 is a valid DEPTNO in the DEPT table (thus satisfying the FOREIGN KEY integrity constraint on the EMP table).

An INSERT statement such as

```
INSERT INTO emp_dept (ename, empno, deptno)
  VALUES ('KURODA', 9010, 77);
```

would fail for the same reason that such an UPDATE on the base EMP table would fail: the FOREIGN KEY integrity constraint on the EMP table is violated.

An INSERT statement such as

```
INSERT INTO emp_dept (empno, ename, loc)
  VALUES (9010, 'KURODA', 'BOSTON');
```

would fail with an ORA-01776 error (“cannot modify more than one base table through a view”).

An INSERT cannot, implicitly or explicitly, refer to columns of a non-key-preserved table. If the join view is defined using the WITH CHECK OPTION clause, then you cannot perform an INSERT to it.

Using the UPDATABLE_COLUMNS Views

To assist you in using the capability of modifying join views, three new views have been created in Oracle7. These views are defined in Table 4 – 1.

View Name	Description
USER_UPDATABLE_COLUMNS	Shows all columns in all tables and views in the user's schema that are modifiable.
DBA_UPDATABLE_COLUMNS	Shows all columns in all tables and views in the DBA schema that are modifiable.
ALL_UPDATABLE_VIEWS	Shows all columns in all tables and views that are modifiable.

Table 4 – 1 UPDATABLE_COLUMNS Views

Outer Joins

Views that involve outer joins are modifiable in some cases. For example, the following view:

```
CREATE VIEW emp_dept_oj1 AS
  SELECT empno, ename, e.deptno, dname, loc
  FROM emp e, dept d
  WHERE e.deptno = d.deptno (+);
```

The statement

```
SELECT * FROM emp_dept_oj1;
```

results in:

EMPNO	ENAME	DEPTNO	DNAME	LOC
7369	SMITH	40	OPERATIONS	BOSTON
7499	ALLEN	30	SALES	CHICAGO
7566	JONES	20	RESEARCH	DALLAS
7654	MARTIN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7782	CLARK	10	ACCOUNTING	NEW YORK
7788	SCOTT	20	RESEARCH	DALLAS
7839	KING	10	ACCOUNTING	NEW YORK
7844	TURNER	30	SALES	CHICAGO
7876	ADAMS	20	RESEARCH	DALLAS
7900	JAMES	30	SALES	CHICAGO
7902	FORD	20	RESEARCH	DALLAS
7934	MILLER	10	ACCOUNTING	NEW YORK
7521	WARD	30	SALES	CHICAGO

14 rows selected.

Columns in the base EMP table of EMP_DEPT_OJ1 are modifiable through the view, because EMP is a key-preserved table in the join.

The following view also contains an outer join:

```
CREATE VIEW emp_dept_oj2 AS
  SELECT e.empno, e.ename, e.deptno, d.dname, d.loc
  FROM emp e, dept d
  WHERE e.deptno (+) = d.deptno;
```

The statement

```
SELECT * FROM emp_dept_oj2;
```

results in:

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK
7369	SMITH	20	RESEARCH	DALLAS
7876	ADAMS	20	RESEARCH	DALLAS
7902	FORD	20	RESEARCH	DALLAS
7788	SCOTT	20	RESEARCH	DALLAS
7566	JONES	20	RESEARCH	DALLAS
7499	ALLEN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7654	MARTIN	30	SALES	CHICAGO
7900	JAMES	30	SALES	CHICAGO
7844	TURNER	30	SALES	CHICAGO
7521	WARD	30	SALES	CHICAGO
			OPERATIONS	BOSTON

15 rows selected.

In this view, EMP is no longer a key-preserved table, because the EMPNO column in the result of the join can have nulls (the last row in the SELECT above). So, UPDATE, DELETE, and INSERT operations cannot be performed on this view.

In the case of views containing an outer join on other nested views, a table is key preserved if the view or views containing the table are merged into their outer views, all the way to the top. A view which is being outer-joined is currently merged only if it is “simple.” For example:

```
SELECT col1, col2, ... FROM T;
```

that is, the select list of the view has no expressions, and there is no WHERE clause.

Consider the following set of views:

```
CREATE emp_v AS
  SELECT empno, ename, deptno
  FROM emp;

CREATE VIEW emp_dept_oj1 AS
  SELECT e.*, loc, d.dname
  FROM emp_v e, dept d
  WHERE e.deptno = d.deptno (+);
```

In these examples, EMP_V is merged into EMP_DEPT_OJ1 because EMP_V is a simple view, and so EMP is a key-preserved table. But if EMP_V is changed as follows:

```
CREATE emp_v_2 AS
  SELECT empno, ename, deptno
  FROM emp
  WHERE sal > 1000;
```

then, because of the presence of the WHERE clause, EMP_V_2 cannot be merged into EMP_DEPT_OJ1, and hence EMP is no longer a key-preserved table.

If you are in doubt whether a view is modifiable, you can SELECT from the view USER_UPDATABLE_COLUMNS to see if it is. For example:

```
SELECT * FROM USER_UPDATABLE_COLUMNS WHERE TABLE_NAME =
'EMP_DEPT_VIEW' ;
```

might return:

OWNER	TABLE_NAME	COLUMN_NAM	UPD
-----	-----	-----	---
SCOTT	EMP_DEPT_V	EMPNO	NO
SCOTT	EMP_DEPT_V	ENAME	NO
SCOTT	EMP_DEPT_V	DEPTNO	NO
SCOTT	EMP_DEPT_V	DNAME	NO
SCOTT	EMP_DEPT_V	LOC	NO
5 rows selected.			

Managing Sequences

The sequence generator generates sequential numbers. Sequence number generation is useful to generate unique primary keys for your data automatically, and to coordinate keys across multiple rows or tables.

Without sequences, sequential values can only be produced programmatically. A new primary key value can be obtained by selecting the most recently produced value and incrementing it. This method requires a lock during the transaction and causes multiple users to wait for the next value of the primary key; this waiting is known as *serialization*. If you have such constructs in your applications, you should replace them with access to sequences. Sequences eliminate serialization and improve the concurrency of your application.

The following sections explain how to create, alter, use, and drop sequences using SQL commands. For instructions on tuning your sequences, see pages 4 – 29 and following.

Creating Sequences

Use the SQL command `CREATE SEQUENCE` to create a sequence. The following statement creates a sequence used to generate employee numbers for the `EMPNO` column of the `EMP` table:

```
CREATE SEQUENCE emp_sequence
    INCREMENT BY 1
    START WITH 1
    NOMAXVALUE
    NOCYCLE
    CACHE 10;
```

Notice that several parameters can be specified to control the function of sequences. You can use these parameters to indicate whether the sequence is ascending or descending, the starting point of the sequence, the minimum and maximum values, and the interval between sequence values. The `NOCYCLE` option indicates that the sequence cannot generate more values after reaching its maximum or minimum value.

The `CACHE` option of the `CREATE SEQUENCE` command pre-allocates a set of sequence numbers and keeps them in memory so that they can be accessed faster. When the last of the sequence numbers in the cache have been used, another set of numbers is read into the cache.

For additional implications for caching sequence numbers when using the Oracle Parallel Server, see the *Oracle7 Parallel Server Administrator's Guide*. General information about caching sequence numbers is included on page 4 – 29.

Privileges Required to Create a Sequence

To create a sequence in your schema, you must have the CREATE SEQUENCE system privilege. To create a sequence in another user's schema, you must have the CREATE ANY SEQUENCE privilege.

Altering Sequences

You can change any of the parameters that define how corresponding sequence numbers are generated; however, you cannot alter a sequence to change the starting number of a sequence. To do this, the sequence must be dropped and re-created.

Use the SQL command ALTER SEQUENCE to alter a sequence, as in

```
ALTER SEQUENCE emp_sequence
  INCREMENT BY 10
  MAXVALUE 10000
  CYCLE
  CACHE 20;
```

Privileges Required to Alter a Sequence To alter a sequence, your schema must contain the sequence, or you must have the ALTER ANY SEQUENCE system privilege.

Using Sequences

The following sections provide some information on how to use a sequence once it has been defined. Once defined, a sequence can be made available to many users. A sequence can be accessed and incremented by multiple users with no waiting. Oracle does not wait for a transaction that has incremented a sequence to complete before that sequence can be incremented again.

The examples outlined in the following sections show how sequences can be used in master/detail table relationships. Assume an order entry system is partially comprised of two tables, ORDERS (master table) and LINE_ITEMS (detail table), that hold information about customer orders. A sequence named ORDER_SEQ is defined by the following statement:

```
CREATE SEQUENCE order_seq
  START WITH 1
  INCREMENT BY 1
  NOMAXVALUE
  NOCYCLE
  CACHE 20;
```

Referencing a Sequence

A sequence is referenced in SQL statements with the NEXTVAL and CURRVAL pseudocolumns; each new sequence number is generated by a reference to the sequence's pseudocolumn NEXTVAL, while the current sequence number can be repeatedly referenced using the pseudo-column CURRVAL.

NEXTVAL and CURRVAL are not reserved words or keywords and can be used as pseudo-column names in SQL statements such as SELECTs, INSERTs, or UPDATEs.

Generating Sequence Numbers with NEXTVAL To generate and use a sequence number, reference *seq_name*.NEXTVAL. For example, assume a customer places an order. The sequence number can be referenced in a values list, as in

```
INSERT INTO orders (orderno, custno)
VALUES (order_seq.NEXTVAL, 1032);
```

or in the SET clause of an UPDATE statement, as in

```
UPDATE orders
  SET orderno = order_seq.NEXTVAL
  WHERE orderno = 10112;
```

or the outermost SELECT of a query or subquery, as in

```
SELECT order_seq.NEXTVAL FROM dual;
```

As defined, the first reference to ORDER_SEQ.NEXTVAL returns the value 1. Each subsequent statement that references ORDER_SEQ.NEXTVAL generates the next sequence number (2, 3, 4, . . .). The pseudo-column NEXTVAL can be used to generate as many new sequence numbers as necessary. However, only a single sequence number can be generated per row; that is, if NEXTVAL is referenced more than once in a single statement, the first reference generates the next number and all subsequent references in the statement return the same number.

Once a sequence number is generated, the sequence number is available only to the session that generated the number. Independent of transactions committing or rolling back, other users referencing `ORDER_SEQ.NEXTVAL` obtain unique values. If two users are accessing the same sequence concurrently, the sequence numbers each user receives might have gaps because sequence numbers are also being generated by the other user.

Using Sequence Numbers with CURRVAL To use or refer to the current sequence value of your session, reference `seq_name.CURRVAL`. `CURRVAL` can only be used if `seq_name.NEXTVAL` has been referenced in the current user session (in the current or a previous transaction). `CURRVAL` can be referenced as many times as necessary, including multiple times within the same statement. The next sequence number is not generated until `NEXTVAL` is referenced. Continuing with the previous example, you would finish placing the customer's order by inserting the line items for the order:

```
INSERT INTO line_items (orderno, partno, quantity)
VALUES (order_seq.CURRVAL, 20321, 3);

INSERT INTO line_items (orderno, partno, quantity)
VALUES (order_seq.CURRVAL, 29374, 1);
```

Assuming the `INSERT` statement given in the previous section generated a new sequence number of 347, both rows inserted by the statements in this section insert rows with order numbers of 347.

Uses and Restrictions of NEXTVAL and CURRVAL `CURRVAL` and `NEXTVAL` can be used in the following places:

- `VALUES` clause of `INSERT` statements
- the `SELECT` list of a `SELECT` statement
- the `SET` clause of an `UPDATE` statement

CURRVAL and NEXTVAL cannot be used in these places:

- a subquery
- a view's query or snapshot's query
- a SELECT statement with the DISTINCT operator
- a SELECT statement with a GROUP BY or ORDER BY clause
- a SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- the WHERE clause of a SELECT statement
- DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- the condition of a CHECK constraint

Caching Sequence Numbers

Sequence numbers can be kept in the sequence cache in the System Global Area (SGA). Sequence numbers can be accessed more quickly in the sequence cache than they can be read from disk.

The sequence cache consists of entries. Each entry can hold many sequence numbers for a single sequence.

Follow these guidelines for fast access to all sequence numbers:

- Be sure the sequence cache can hold all the sequences used concurrently by your applications.
- Increase the number of values for each sequence held in the sequence cache.

The Number of Entries in the Sequence Cache When an application accesses a sequence in the sequence cache, the sequence numbers are read quickly. However, if an application accesses a sequence that is not in the cache, the sequence must be read from disk to the cache before the sequence numbers are used.

If your applications use many sequences concurrently, your sequence cache might not be large enough to hold all the sequences. In this case, access to sequence numbers might often require disk reads. For fast access to all sequences, be sure your cache has enough entries to hold all the sequences used concurrently by your applications.

The number of entries in the sequence cache is determined by the initialization parameter `SEQUENCE_CACHE_ENTRIES`. The default value for this parameter is 10 entries. Oracle creates and uses sequences internally for auditing, grants of system privileges, grants of object privileges, profiles, debugging stored procedures, and labels. Be sure your sequence cache has enough entries to hold these sequences as well as sequences used by your applications.

If the value for your `SEQUENCE_CACHE_ENTRIES` parameter is too low, it is possible to skip sequence values. For example, assume that this parameter is set to 4, and that you currently have four cached sequences. If you create a fifth sequence, it will replace the least recently used sequence in the cache. All of the remaining values in this displaced sequence are lost. That is, if the displaced sequence originally held 10 cached sequence values, and only one had been used, nine would be lost when the sequence was displaced.

The Number of Values in Each Sequence Cache Entry When a sequence is read into the sequence cache, sequence values are generated and stored in a cache entry. These values can then be accessed quickly. The number of sequence values stored in the cache is determined by the `CACHE` parameter in the `CREATE SEQUENCE` statement. The default value for this parameter is 20.

This `CREATE SEQUENCE` statement creates the `SEQ2` sequence so that 50 values of the sequence are stored in the `SEQUENCE` cache:

```
CREATE SEQUENCE seq2
  CACHE 50
```

The first 50 values of `SEQ2` can then be read from the cache. When the 51st value is accessed, the next 50 values will be read from disk.

Choosing a high value for `CACHE` allows you to access more successive sequence numbers with fewer reads from disk to the sequence cache. However, if there is an instance failure, all sequence values in the cache are lost. Cached sequence numbers also could be skipped after an export and import if transactions continue to access the sequence numbers while the export is running.

If you use the `NOCACHE` option in the `CREATE SEQUENCE` statement, the values of the sequence are not stored in the sequence cache. In this case, every access to the sequence requires a disk read. Such disk reads slow access to the sequence. This `CREATE SEQUENCE` statement creates the `SEQ3` sequence so that its values are never stored in the cache:

```
CREATE SEQUENCE seq3
  NOCACHE
```

Privileges Required to Use a Sequence

To use a sequence, your schema must contain the sequence or you must have been granted the SELECT object privilege for another user's sequence.

Dropping Sequences

To drop a sequence, use the SQL command DROP SEQUENCE. For example, the following statement drops the ORDER_SEQ sequence:

```
DROP SEQUENCE order_seq;
```

When you drop a sequence, its definition is removed from the data dictionary. Any synonyms for the sequence remain, but return an error when referenced.

Privileges Required to Drop a Sequence You can drop any sequence in your schema. To drop a sequence in another schema, you must have the DROP ANY SEQUENCE system privilege.

Managing Synonyms

A synonym is an alias for a table, view, snapshot, sequence, procedure, function, or package. The following sections explain how to create, use, and drop synonyms using SQL commands.

Creating Synonyms

Use the SQL command CREATE SYNONYM to create a synonym. The following statement creates a public synonym named PUBLIC_EMP on the EMP table contained in the schema of JWARD:

```
CREATE PUBLIC SYNONYM public_emp FOR jward.emp;
```

Privileges Required to Create a Synonym

You must have the CREATE SYNONYM system privilege to create a private synonym in your schema, or the CREATE ANY SYNONYM system privilege to create a private synonym in another user's schema. To create a public synonym, you must have the CREATE PUBLIC SYNONYM system privilege.

Using Synonyms

A synonym can be referenced in a SQL statement the same way that the underlying object of the synonym can be referenced. For example, if a synonym named EMP refers to a table or view, the following statement is valid:

```
INSERT INTO emp (empno, ename, job)
VALUES (emp_sequence.NEXTVAL, 'SMITH', 'CLERK');
```

If the synonym named FIRE_EMP refers to a standalone procedure or package procedure, you could execute it in SQL*Plus or Server Manager with the command

```
EXECUTE fire_emp(7344);
```

Privileges Required to Use a Synonym

You can successfully use any private synonym contained in your schema or any public synonym, assuming that you have the necessary privileges to access the underlying object, either explicitly, from an enabled role, or from PUBLIC. You can also reference any private synonym contained in another schema if you have been granted the necessary object privileges for the private synonym. You can only reference another user's synonym using the object privileges that you have been granted. For example, if you have the SELECT privilege for the JWARD.EMP synonym, you can query the JWARD.EMP synonym, but you cannot insert rows using the synonym for JWARD.EMP.

Dropping Synonyms

To drop a synonym, use the SQL command DROP SYNONYM. To drop a private synonym, omit the PUBLIC keyword; to drop a public synonym, include the PUBLIC keyword. The following statement drops the private synonym named EMP:

```
DROP SYNONYM emp;
```

The following statement drops the public synonym named PUBLIC_EMP:

```
DROP PUBLIC SYNONYM public_emp;
```

When you drop a synonym, its definition is removed from the data dictionary. All objects that reference a dropped synonym remain (for example, views and procedures) but become invalid.

Privileges Required to Drop a Synonym

You can drop any private synonym in your own schema. To drop a private synonym in another user's schema, you must have the DROP ANY SYNONYM system privilege. To drop a public synonym, you must have the DROP PUBLIC SYNONYM system privilege.

Managing Indexes

Indexes are used in Oracle to provide quick access to rows in a table. Indexes provide faster access to data for operations that return a small portion of a table's rows.

Oracle does not limit the number of indexes you can create on a table. However, you should consider the performance benefits of indexes and the needs of your database applications to determine which columns to index.

The following sections explain how to create, alter, and drop indexes using SQL commands. Some simple guidelines to follow when managing indexes are included. See the *Oracle7 Server Tuning* manual for performance implications of index creation.

Create Indexes After Inserting Table Data

With one notable exception, you should usually create indexes after you have inserted or loaded (using SQL*Loader or Import) data into a table. It is more efficient to insert rows of data into a table that has no indexes and then to create the indexes for subsequent queries, etc. If you create indexes before table data is loaded, every index must be updated every time you insert a row into the table. The exception to this rule is that you must create an index for a cluster before you insert any data into the cluster.

When you create an index on a table that already has data, Oracle must use sort space to create the index. Oracle uses the sort space in memory allocated for the creator of the index (the amount per user is determined by the initialization parameter `SORT_AREA_SIZE`), but must also swap sort information to and from temporary segments allocated on behalf of the index creation. If the index is extremely large, it might be beneficial to complete the following steps:

1. Create a new temporary tablespace using the `CREATE TABLESPACE` command.
2. Use the `TEMPORARY TABLESPACE` option of the `ALTER USER` command to make this your new temporary tablespace.
3. Create the index using the `CREATE INDEX` command.
4. Drop this tablespace using the `DROP TABLESPACE` command. Then use the `ALTER USER` command to reset your temporary tablespace to your original temporary tablespace.

Under certain conditions, you can load data into a table with the SQL*Loader “direct path load”, and an index can be created as data is loaded; refer to the *Oracle7 Server Utilities* manual for more information.

Index the Correct Tables and Columns Use the following guidelines for determining when to create an index:

- Create an index if you frequently want to retrieve less than 15% of the rows in a large table. The percentage varies greatly according to the relative speed of a table scan and how clustered the row data is about the index key. The faster the table scan, the lower the percentage; the more clustered the row data, the higher the percentage.
- Index columns used for joins to improve performance on joins of multiple tables.

Note: Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key; see “Concurrency Control, Indexes, and Foreign Keys” on page 6 – 10 for more information.

- Small tables do not require indexes; if a query is taking too long, the table might have grown from small to large.

Some columns are strong candidates for indexing. Columns with one or more of the following characteristics are candidates for indexing:

- Values are relatively unique in the column.
- There is a wide range of values.
- The column contains many nulls, but queries often select all rows having a value. In this case, the phrase

```
WHERE COL_X > -9.99 x 10^125
```

is preferable to

```
WHERE COL_X IS NOT NULL
```

because the first uses an index on COL_X (assuming that COL_X is a numeric column).

Columns with the following characteristics are less suitable for indexing:

- The column has few distinct values (for example, a column for the sex of employees).
- There are many nulls in the column and you do not search on the non-null values.

LONG and LONG RAW columns cannot be indexed.

The size of a single index entry cannot exceed roughly one-half (minus some overhead) of the available space in the data block. Consult with the database administrator for assistance in determining the space required by an index.

Limit the Number of Indexes per Table A table can have any number of indexes. However, the more indexes, the more overhead is incurred as the table is altered. When rows are inserted or deleted, all indexes on the table must be updated. When a column is updated, all indexes on the column must be updated.

Thus, there is a tradeoff between speed of retrieval for queries on a table and speed of accomplishing updates on the table. For example, if a table is primarily read-only, more indexes might be useful, but if a table is heavily updated, fewer indexes might be preferable.

Order Index Columns for Performance The order in which columns are named in the CREATE INDEX command need not correspond to the order in which they appear in the table. However, the order of columns in the CREATE INDEX statement is significant because query performance can be affected by the order chosen. In general, you should put the column expected to be used most often first in the index.

For example, assume the columns of the `VENDOR_PARTS` table are as shown in Figure 4 – 1.

Table <code>VENDOR_PARTS</code>		
VEND ID	PART NO	UNIT COST
1012	10-440	.25
1012	10-441	.39
1012	457	4.95
1010	10-440	.27
1010	457	5.10
1220	08-300	1.33
1012	08-300	1.19
1292	457	5.28

Figure 4 – 1 The `VENDOR_PARTS` Table

Assume that there are five vendors, and each vendor has about 1000 parts.

Suppose that the `VENDOR_PARTS` table is commonly queried by SQL statements such as the following:

```
SELECT * FROM vendor_parts
      WHERE part_no = 457 AND vendor_id = 1012;
```

To increase the performance of such queries, you might create a composite index putting the most selective column first; that is, the column with the *most* values:

```
CREATE INDEX ind_vendor_id
      ON vendor_parts (part_no, vendor_id);
```

Indexes speed retrieval on any query using the *leading portion* of the index. So in the above example, queries with WHERE clauses using only the `PART_NO` column also note a performance gain. Because there are only five distinct values, placing a separate index on `VENDOR_ID` would serve no purpose.

Creating Indexes

You can create an index for a table to improve the performance of queries issued against the corresponding table. You can also create an index for a cluster. You can create a *composite* index on multiple columns up to a maximum of 16 columns. A composite index key cannot exceed roughly one-half (minus some overhead) of the available space in the data block.

Oracle automatically creates an index to enforce a UNIQUE or PRIMARY KEY integrity constraint. In general, it is better to create such constraints to enforce uniqueness and not explicitly use the obsolete CREATE UNIQUE INDEX syntax.

Use the SQL command CREATE INDEX to create an index. The following statement creates an index named EMP_ENAME for the ENAME column of the EMP table:

```
CREATE INDEX emp_ename ON emp(ename)
    TABLESPACE users
    STORAGE (INITIAL 20K
             NEXT 20k
             PCTINCREASE 75)
    PCTFREE 0;
```

Notice that several storage settings are explicitly specified for the index.

Privileges Required to Create an Index

To create a new index, you must own, or have the INDEX object privilege for, the corresponding table. The schema that contains the index must also have a quota for the tablespace intended to contain the index, or the UNLIMITED TABLESPACE system privilege. To create an index in another user's schema, you must have the CREATE ANY INDEX system privilege.

Dropping Indexes

You might drop an index for the following reasons:

- The index is not providing anticipated performance improvements for queries issued against the associated table (the table is very small, or there are many rows in the table but very few index entries, etc.).
- Applications do not contain queries that use the index.
- The index is no longer needed and must be dropped before being rebuilt.

When you drop an index, all extents of the index's segment are returned to the containing tablespace and become available for other objects in the tablespace.

Use the SQL command DROP INDEX to drop an index. For example, to drop the EMP_ENAME index, enter the following statement:

```
DROP INDEX emp_ename;
```

If you drop a table, all associated indexes are dropped.

Privileges Required to Drop an Index To drop an index, the index must be contained in your schema or you must have the DROP ANY INDEX system privilege.

Managing Clusters, Clustered Tables, and Cluster Indexes

Because clusters store related rows of different tables together in the same data blocks, two primary benefits are achieved when clusters are properly used:

- Disk I/O is reduced and access time improves for joins of clustered tables.
- In a cluster, a *cluster key value* (that is, the related value) is only stored once, no matter how many rows of different tables contain the value. Therefore, less storage may be required to store related table data in a cluster than is necessary in non-clustered table format.

Guidelines for Creating Clusters

Some guidelines for creating clusters are outlined below. For performance characteristics, see the *Oracle7 Server Tuning* manual.

Choose Appropriate Tables to Cluster Use clusters to store one or more tables that are primarily queried (not predominantly inserted into or updated), and for which queries often join data of multiple tables in the cluster or retrieve related data from a single table.

Choose Appropriate Columns for the Cluster Key Choose cluster key columns carefully. If multiple columns are used in queries that join the tables, make the cluster key a composite key. In general, the same column characteristics that make a good index apply for cluster indexes; see the section “Index the Correct Tables and Columns” on page 4 – 34 for more information about these guidelines.

A good cluster key has enough unique values so that the group of rows corresponding to each key value fills approximately one data block. Too few rows per cluster key value can waste space and result in negligible performance gains. Cluster keys that are so specific that only a few rows share a common value can cause wasted space in blocks, unless a small SIZE was specified at cluster creation time.

Too many rows per cluster key value can cause extra searching to find rows for that key. Cluster keys on values that are too general (for example, MALE and FEMALE) result in excessive searching and can result in worse performance than with no clustering.

A cluster index cannot be unique or include a column defined as LONG.

Performance Considerations

Also note that clusters can reduce the performance of DML statements (INSERTs, UPDATEs, and DELETEs) as compared to storing a table separately with its own index. These disadvantages relate to the use of space and the number of blocks that must be visited to scan a table. Because multiple tables share each block, more blocks must be used to store a clustered table than if that same table were stored non-clustered. You should decide about using clusters with these tradeoffs in mind.

To identify data that would be better stored in clustered form than non-clustered, look for tables that are related via referential integrity constraints and tables that are frequently accessed together using SELECT statements that join data from two or more tables. If you cluster tables on the columns used to join table data, you reduce the number of data blocks that must be accessed to process the query; all the rows needed for a join on a cluster key are in the same block. Therefore, query performance for joins is improved. Similarly, it may be useful to cluster an individual table. For example, the EMP table could be clustered on the DEPTNO column to cluster the rows for employees in the same department. This would be advantageous if applications commonly process rows, department by department.

Like indexes, clusters do not affect application design. The existence of a cluster is transparent to users and to applications. Data stored in a clustered table is accessed via SQL just like data stored in a non-clustered table.

Creating Clusters, Clustered Tables, and Cluster Indexes

Use a cluster to store one or more tables that are frequently joined in queries. Do not use a cluster to cluster tables that are frequently accessed individually.

Once you create a cluster, tables can be created in the cluster. However, before you can insert any rows into the clustered tables, you must create a cluster index. The use of clusters does not affect the creation of additional indexes on the clustered tables; you can create and drop them as usual.

Use the SQL command `CREATE CLUSTER` to create a cluster. The following statement creates a cluster named `EMP_DEPT`, which stores the `EMP` and `DEPT` tables, clustered by the `DEPTNO` column:

```
CREATE CLUSTER emp_dept (deptno NUMBER(3))
    PCTUSED 80
    PCTFREE 5;
```

Create a table in a cluster using the SQL command `CREATE TABLE` with the `CLUSTER` option. For example, the `EMP` and `DEPT` tables can be created in the `EMP_DEPT` cluster using the following statements:

```
CREATE TABLE dept (
    deptno NUMBER(3) PRIMARY KEY,
    . . . )
    CLUSTER emp_dept (deptno);

CREATE TABLE emp (
    empno NUMBER(5) PRIMARY KEY,
    ename VARCHAR2(15) NOT NULL,
    . . .
    deptno NUMBER(3) REFERENCES dept)
    CLUSTER emp_dept (deptno);
```

A table created in a cluster is contained in the schema specified in the `CREATE TABLE` statement; a clustered table might not be in the same schema that contains the cluster.

You must create a cluster index before any rows can be inserted into any clustered table. For example, the following statement creates a cluster index for the `EMP_DEPT` cluster:

```
CREATE INDEX emp_dept_index
    ON CLUSTER emp_dept
    INITTRANS 2
    MAXTRANS 5
    PCTFREE 5;
```

Note: A cluster index cannot be unique. Furthermore, Oracle is not guaranteed to enforce uniqueness of columns in the cluster key if they have `UNIQUE` or `PRIMARY KEY` constraints.

The cluster key establishes the relationship of the tables in the cluster.

Privileges Required to Create a Cluster, Clustered Table, and Cluster Index

To create a cluster in your schema, you must have the CREATE CLUSTER system privilege and a quota for the tablespace intended to contain the cluster or the UNLIMITED TABLESPACE system privilege. To create a cluster in another user's schema, you must have the CREATE ANY CLUSTER system privilege, and the owner must have a quota for the tablespace intended to contain the cluster or the UNLIMITED TABLESPACE system privilege.

To create a table in a cluster, you must have either the CREATE TABLE or CREATE ANY TABLE system privilege. You do not need a tablespace quota or the UNLIMITED TABLESPACE system privilege to create a table in a cluster.

To create a cluster index, your schema must contain the cluster, and you must have the following privileges:

- the CREATE ANY INDEX system privilege or, if you own the cluster, the CREATE INDEX privilege
- a quota for the tablespace intended to contain the cluster index, or the UNLIMITED TABLESPACE system privilege

Manually Allocating Storage for a Cluster

Oracle dynamically allocates additional extents for the data segment of a cluster, as required. In some circumstances, you might want to explicitly allocate an additional extent for a cluster. For example, when using the Oracle Parallel Server, an extent of a cluster can be allocated explicitly for a specific instance.

You can allocate a new extent for a cluster using the SQL command ALTER CLUSTER with the ALLOCATE EXTENT option; see the *Oracle7 Parallel Server Concepts & Administration* manual for more information.

Dropping Clusters, Clustered Tables, and Cluster Indexes

Drop a cluster if the tables currently within the cluster are no longer necessary. When you drop a cluster, the tables within the cluster and the corresponding cluster index are dropped; all extents belonging to both the cluster's data segment and the index segment of the cluster index are returned to the containing tablespace and become available for other segments within the tablespace.

You can individually drop clustered tables without affecting the table's cluster, other clustered tables, or the cluster index. Drop a clustered table in the same manner as a non-clustered table—use the SQL command `DROP TABLE`. See page 4 – 9 for more information about individually dropping tables.

Note: When you drop a single clustered table from a cluster, each row of the table must be deleted from the cluster. To maximize efficiency, if you intend to drop the entire cluster including all tables, use the `DROP CLUSTER` command with the `INCLUDING TABLES` option. You should only use the `DROP TABLE` command to drop an individual table from a cluster when the rest of the cluster is going to remain.

You can drop a cluster index without affecting the cluster or its clustered tables. However, you cannot use a clustered table if it does not have a cluster index. Cluster indexes are sometimes dropped as part of the procedure to rebuild a fragmented cluster index. See page 4 – 37 for more information about dropping indexes.

To drop a cluster that contains no tables, as well as its cluster index, if present, use the SQL command `DROP CLUSTER`. For example, the following statement drops the empty cluster named `EMP_DEPT`:

```
DROP CLUSTER emp_dept;
```

If the cluster contains one or more clustered tables and you intend to drop the tables as well, add the `INCLUDING TABLES` option of the `DROP CLUSTER` command, as in

```
DROP CLUSTER emp_dept INCLUDING TABLES;
```

If you do not include the `INCLUDING TABLES` option, and the cluster contains tables, an error is returned.

If one or more tables in a cluster contain primary or unique keys that are referenced by `FOREIGN KEY` constraints of tables outside the cluster, you cannot drop the cluster unless you also drop the dependent `FOREIGN KEY` constraints. Use the `CASCADE CONSTRAINTS` option of the `DROP CLUSTER` command, as in

```
DROP CLUSTER emp_dept INCLUDING TABLES CASCADE CONSTRAINTS;
```

An error is returned if the above option is not used in the appropriate situation.

Privileges Required to Drop a Cluster

To drop a cluster, your schema must contain the cluster, or you must have the DROP ANY CLUSTER system privilege. You do not have to have any special privileges to drop a cluster that contains tables, even if the clustered tables are not owned by the owner of the cluster.

Managing Hash Clusters and Clustered Tables

The following sections explain how to create, alter, and drop hash clusters and clustered tables using SQL commands.

Creating Hash Clusters and Clustered Tables

A hash cluster is used to store individual tables or a group of clustered tables that are static and often queried by equality queries. Once you create a hash cluster, you can create tables. To create a hash cluster, use the SQL command CREATE CLUSTER. The following statement creates a cluster named TRIAL_CLUSTER that is used to store the TRIAL table, clustered by the TRIALNO column:

```
CREATE CLUSTER trial_cluster (trialno NUMBER(5,0))
    PCTUSED 80      PCTFREE 5
    SIZE 2K
    HASH IS trialno HASHKEYS 100000;

CREATE TABLE trial (
    trialno NUMBER(5) PRIMARY KEY,
    ...)
    CLUSTER trial_cluster (trialno);
```

Controlling Space Usage Within a Hash Cluster

When you create a hash cluster, it is important that you correctly choose the cluster key and set the HASH IS, SIZE, and HASHKEYS parameters to achieve the desired performance and space usage for the cluster. The following sections provide guidance, as well as examples of setting these parameters.

Choosing the Key

Choosing the correct cluster key is dependent on the most common types of queries issued against the clustered tables. For example, consider the EMP table in a hash cluster. If queries often select rows by employee number, the EMPNO column should be the cluster key; if queries often select rows by department number, the DEPTNO column should be the cluster key. For hash clusters that contain a single table, the cluster key is typically the entire primary key of the contained table. A hash cluster with a composite key must use Oracle's internal hash function.

Setting HASH IS

Only specify the HASH IS parameter if the cluster key is a single column of the NUMBER datatype, and contains uniformly distributed integers. If the above conditions apply, you can distribute rows in the cluster such that each unique cluster key value hashes to a unique hash value (with no collisions). If the above conditions do not apply, you should use the internal hash function.

Dropping Hash Clusters

Drop a hash cluster using the SQL command DROP CLUSTER:

```
DROP CLUSTER emp_dept;
```

Drop a table in a hash cluster using the SQL command DROP TABLE. The implications of dropping hash clusters and tables in hash clusters are the same as for index clusters. See page 4 – 41 for more information about dropping clusters and the required privileges.

When to Use Hashing

Storing a table in a hash cluster is an alternative to storing the same table with an index. Hashing is useful in the following situations:

- Most queries are equality queries on the cluster key. For example:

```
SELECT . . . WHERE cluster_key = . . . ;
```

In such cases, the cluster key in the equality condition is hashed, and the corresponding hash key is usually found with a single read. With an indexed table, the key value must first be found in the index (usually several reads), and then the row is read from the table (another read).

- The table or tables in the hash cluster are primarily static in size such that you can determine the number of rows and amount of space required for the tables in the cluster. If tables in a hash cluster require more space than the initial allocation for the cluster, performance degradation can be substantial because overflow blocks are required.
- A hash cluster with the `HASH IS col`, `HASHKEYS n`, and `SIZE m` clauses is an ideal representation for an array (table) of n items (rows) where each item consists of m bytes of data. For example:

```
ARRAY X[100] OF NUMBER(8)
```

could be represented as

```
CREATE CLUSTER c(subscript INTEGER)
    HASH IS subscript HASHKEYS 100 SIZE 10;
CREATE TABLE x(subscript NUMBER(2), value NUMBER(8))
    CLUSTER c(subscript);
```

Alternatively, hashing is not advantageous in the following situations:

- Most queries on the table retrieve rows over a range of cluster key values. For example, in full table scans, or queries such as

```
SELECT . . . WHERE cluster_key < . . . ;
```

A hash function cannot be used to determine the location of specific hash keys; instead, the equivalent of a full table scan must be done to fetch the rows for the query. With an index, key values are ordered in the index, so cluster key values that satisfy the WHERE clause of a query can be found with relatively few I/Os.

- A table is not static, but is continually growing. If a table grows without limit, the space required over the life of the table (thus, of its cluster) cannot be predetermined.
- Applications frequently perform full table scans on the table and the table is sparsely populated. A full table scan in this situation takes longer under hashing.
- You cannot afford to preallocate the space the hash cluster will eventually need.

In most cases, you should decide (based on the above information) whether to use hashing or indexing. If you use indexing, consider whether it is best to store a table individually or as part of a cluster; see page 4 – 38 for guidance.

If you decide to use hashing, a table can still have separate indexes on any columns, including the cluster key. For additional guidelines on the performance characteristics of hash clusters, see the *Oracle7 Server Tuning* manual.

Miscellaneous Management Topics for Schema Objects

The following sections explain miscellaneous topics regarding the management of the various schema objects discussed in this chapter.

Creating Multiple Tables and Views in One Operation

You can create several tables and views and grant privileges in one operation using the SQL command `CREATE SCHEMA`. The `CREATE SCHEMA` command is useful if you want to guarantee the creation of several tables and views and grants in one operation; if an individual table or view creation fails or a grant fails, the entire statement is rolled back and none of the objects are created or the privileges granted.

For example, the following statement creates two tables and a view that joins data from the two tables:

```
CREATE SCHEMA AUTHORIZATION scott
  CREATE VIEW sales_staff AS
    SELECT empno, ename, sal, comm
    FROM emp
    WHERE deptno = 30 WITH CHECK OPTION CONSTRAINT
                      sales_staff_cnst

  CREATE TABLE emp (
    empno      NUMBER(5) PRIMARY KEY,
    ename      VARCHAR2(15) NOT NULL,
    job        VARCHAR2(10),
    mgr        NUMBER(5),
    hiredate   DATE DEFAULT (sysdate),
    sal        NUMBER(7,2),
    comm       NUMBER(7,2),
    deptno     NUMBER(3) NOT NULL
              CONSTRAINT dept_fkey REFERENCES dept)

  CREATE TABLE dept (
    deptno     NUMBER(3) PRIMARY KEY,
    dname      VARCHAR2(15),
    loc        VARCHAR2(25))

  GRANT SELECT ON sales_staff TO human_resources;
```

The `CREATE SCHEMA` command does not support Oracle extensions to the `ANSI CREATE TABLE` and `CREATE VIEW` commands (for example, the `STORAGE` clause).

Privileges Required to Create Multiple Schema Objects

To create schema objects, such as multiple tables, using the `CREATE SCHEMA` command, you must have the required privileges for any included operation.

Naming Objects

You should decide when you want to use partial and complete global object names in the definition of views, synonyms, and procedures. Keep in mind that database names should be stable and databases should not be unnecessarily moved within a network.

In a distributed database system, each database should have a unique global name. The global name is composed of the database name and the network domain that contains the database. Each object in the database then has a global object name consisting of the object name and the global database name. Because Oracle ensures that the object name is unique within a database, you can ensure that it is unique across all databases by assigning unique global database names. You should coordinate with your database administrator on this task, as it is usually the DBA who is responsible for assigning database names.

Name Resolution in SQL Statements

An object name takes the form

```
[ schema. ] name [ @database ]
```

Some examples include the following:

```
emp  
scott.emp  
scott.emp@personnel
```

A session is established when a user logs onto a database. Object names are resolved relative to the current user session. The username of the current user is the default schema. The database to which the user has directly logged-on is the default database.

Oracle has separate namespaces for different classes of objects. All objects in the same namespace must have distinct names, but two objects in different namespaces can have the same name. Tables, views, snapshots, sequences, synonyms, procedures, functions, and packages are in a single namespace. Triggers, indexes, and clusters each have their own individual namespace. For example, there can be a table, trigger, and index all named SCOTT.EMP.

Based on the context of an object name, Oracle searches the appropriate namespace when resolving the name to an object. For example, in the statement

```
DROP CLUSTER test
```

Oracle looks up TEST in the cluster namespace.

Rather than supplying an object name directly, you can also refer to an object using a synonym. A private synonym name has the same syntax as an ordinary object name. A public synonym is implicitly in the PUBLIC schema, but users cannot explicitly qualify a synonym with the schema PUBLIC. Synonyms can only be used to reference objects in the same namespace as tables. Due to the possibility of synonyms, the following rules are used to resolve a name in a context that requires an object in the table namespace:

1. Look up the name in the table namespace.
2. If the name resolves to an object that is not a synonym, no further work is needed.
3. If the name resolves to a private synonym, replace the name with the definition of the synonym and return to step 1.
4. If the name was originally qualified with a schema, return an error; otherwise, check if the name is a public synonym.
5. If the name is not a public synonym, return an error; otherwise, replace the name with the definition of the public synonym and return to step 1. When global object names are used in a distributed database (either explicitly or indirectly within a synonym), the local Oracle session resolves the reference as is locally required (for example, resolving a synonym to a remote table's global object name). After the partially resolved statement is shipped to the remote database, the remote Oracle session completes the resolution of the object as above. See the *Oracle7 Server Concepts* manual for a complete description of name resolution in a distributed database.

Renaming Schema Objects

If necessary, you can rename some schema objects using two different methods: drop and re-create the object, or rename the object using the SQL command RENAME.



Attention: If you drop an object and re-create it, all privilege grants for the object are lost when the object is dropped. Privileges must be granted again when the object is re-created.

Alternatively, a table, view, sequence, or a private synonym of a table, view, or sequence can be renamed using the RENAME command. When using the RENAME command, grants made for the object are carried forward for the new name. For example, the following statement renames the SALES_STAFF view:

```
RENAME sales_staff TO dept_30;
```


You cannot rename a stored PL/SQL program unit, public synonym, index, or cluster. To rename such an object, you must drop and re-create it.

Renaming a schema object has the following effects:

- All views and PL/SQL program units dependent on a renamed object become invalid (must be recompiled before next use).
- All synonyms for a renamed object return an error when used.

Privileges Required to Rename an Object

To rename an object, you must be the owner of the object.

Listing Information about Schema Objects

The data dictionary provides many views that provide information about the schema objects described in this chapter. The following is a summary of the views associated with schema objects:

- ALL_OBJECTS, USER_OBJECTS
- ALL_CATALOG, USER_CATALOG
- ALL_TABLES, USER_TABLES
- ALL_TAB_COLUMNS, USER_TAB_COLUMNS
- ALL_TAB_COMMENTS, USER_TAB_COMMENTS
- ALL_COL_COMMENTS, USER_COL_COMMENTS
- ALL_VIEWS, USER_VIEWS
- ALL_INDEXES, USER_INDEXES
- ALL_IND_COLUMNS, USER_IND_COLUMNS
- USER_CLUSTERS
- USER_CLU_COLUMNS
- ALL_SEQUENCES, USER_SEQUENCES
- ALL_SYNONYMS, USER_SYNONYMS
- ALL_DEPENDENCIES, USER_DEPENDENCIES

The following sections provide examples of using some of the above views.

Example 1
Listing Different
Schema Objects by
Type

The following query lists all of the objects owned by the user issuing the query:

```
SELECT object_name, object_type FROM user_objects;
```

The query above might return results similar to the following:

OBJECT_NAME	OBJECT_TYPE
-----	-----
EMP_DEPT	CLUSTER
EMP	TABLE
DEPT	TABLE
EMP_DEPT_INDEX	INDEX
PUBLIC_EMP	SYNONYM
EMP_MGR	VIEW

Example 2
Listing Column
Information

Column information, such as name, datatype, length, precision, scale, and default data values, can be listed using one of the views ending with the _COLUMNS suffix. For example, the following query lists all of the default column values for the EMP and DEPT tables:

```
SELECT table_name, column_name, data_default
FROM user_tab_columns
WHERE table_name = 'DEPT' OR table_name = 'EMP';
```

Considering the example statements at the beginning of this section, a display similar to the one below is displayed:

TABLE_NAME	COLUMN_NAME	DATA_DEFAULT
-----	-----	-----
DEPT	DEPTNO	
DEPT	DNAME	
DEPT	LOC	('NEW YORK')
EMP	EMPNO	
EMP	ENAME	
EMP	JOB	
EMP	MGR	
EMP	HIREDATE	(sysdate)
EMP	SAL	
EMP	COMM	
EMP	DEPTNO	

Notice that not all columns have a user-specified default. These columns assume NULL when rows that do not specify values for these columns are inserted.

Example 3 Listing Dependencies of Views and Synonyms

When you create a view or a synonym, the view or synonym is based on its underlying base object. The `_DEPENDENCIES` data dictionary views can be used to reveal the dependencies for a view and the `_SYNONYMS` data dictionary views can be used to list the base object of a synonym. For example, the following query lists the base objects for the synonyms created by the user JWARD:

```
SELECT table_owner, table_name
       FROM all_synonyms
       WHERE owner = 'JWARD';
```

This query might return information similar to the following:

TABLE_OWNER	TABLE_NAME
SCOTT	DEPT
SCOTT	EMP

Selecting a Datatype

This chapter discusses the Oracle datatypes, their properties, and mapping to non-Oracle datatypes. Topics discussed include the following:

- the Oracle datatypes: CHAR, VARCHAR2, NUMBER, DATE, LONG, RAW, and LONG RAW
- ROWIDs and the ROWID datatype
- the Trusted Oracle MLSLABEL datatype
- ANSI, SQL/DS, and DB2 datatypes
- data conversions

Oracle Datatypes

The following sections describe the Oracle datatypes that can be used for column definitions.

Character Datatypes

The CHAR and VARCHAR2 datatypes are used to store alphanumeric data; any character can be stored in a column of these datatypes. Character data is stored in strings with byte values corresponding to the character encoding scheme (usually called a character set or code page) defined for the database when it was created; for example, 7-bit ASCII, EBCDIC Code Page 500, or Japan Extended UNIX. Oracle supports both single-byte and multi-byte encoding schemes. See the *Oracle7 Server Reference* manual for more information about National Language Support features of Oracle and support for different character encoding schemes.

Because Oracle blank-pads values stored in CHAR columns but not in VARCHAR2 columns, a value stored in a VARCHAR2 column may take up less space than if it were stored in a CHAR column. For this reason, a full table scan on a large table containing VARCHAR2 columns may read fewer data blocks than a full table scan on a table containing the same data stored in CHAR columns. If your application often performs full table scans on large tables containing character data, you might be able to improve performance by storing this data in VARCHAR2 columns rather than in CHAR columns.

However, performance is not the only factor to consider when deciding which of these datatypes to use. Oracle uses different semantics to compare values of each datatype. You might choose one datatype over the other if your application is sensitive to the differences between these semantics. For example, if you want Oracle to ignore trailing blanks when comparing character values, you must store these values in CHAR columns. For information on the comparison semantics for these datatypes, see the *Oracle7 Server SQL Reference* manual.

CHAR Datatype

The CHAR datatype is used to store **fixed**-length character strings. When you create a table with a CHAR column, specify a column length (in bytes, not characters) between 1 and 255 (default is 1). Oracle then guarantees the following:

- If a shorter value is given, the value is blank-padded to the fixed length.
- If a value is too large, Oracle returns an error.

Oracle compares CHAR values using **blank-padded comparison semantics**. If two values have different lengths, Oracle adds blanks to the shorter value, until the two values are the same length. Two values that differ only in the number of trailing blanks are considered equal.

VARCHAR2 Datatype

Use the VARCHAR2 datatype to store variable-length character strings. When you create a table with a VARCHAR2 column, specify a maximum column length (in bytes, not characters) between 1 and 2000. For each row, each value in the column is stored as a variable-length field. For example, assume a column is declared VARCHAR2 with a maximum size of 50 characters. If only 10 single-byte characters are given for the VARCHAR2 column value in a particular row, the column in the row's row piece only stores the 10 characters (10 bytes), not 50.

Oracle compares VARCHAR2 values using **non-padded comparison semantics**. Two values are only considered equal if they have the same characters and are of equal length.

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. However, in a future version of Oracle, the VARCHAR datatype might be changed to use different comparison semantics. Therefore, you should use the VARCHAR2 datatype to store variable-length character strings.

Column Lengths for Character Datatypes

The lengths of CHAR and VARCHAR2 columns are specified in bytes rather than characters, and are constrained as such. If the database character encoding scheme is single-byte, the number of bytes and the number of characters in a column is the same. If it is multi-byte, there generally is no such correspondence. A character might be comprised of one or more bytes depending upon the specific multi-byte encoding scheme, and whether shift-in/shift-out control codes are present. When using a multi-byte database character encoding scheme, consider carefully the space required for tables with character columns.

NUMBER Datatype

Use the NUMBER datatype to store real numbers in a fixed-point format. Numbers using this datatype are guaranteed to be portable among different Oracle platforms, and offer up to 38 decimal digits of precision. You can store positive and negative numbers of magnitude 1×10^{-130} to $9.99... \times 10^{125}$, as well as zero, in a NUMBER column.

For numeric columns you can specify the column as

```
column_name NUMBER
```

or you can specify a *precision* (total number of digits) and *scale* (number of digits to right of decimal point):

```
column_name NUMBER (precision, scale)
```

If a precision is not specified, the column stores values as given. If no scale is specified, it is implied that the scale is zero. The scale can range from -84 to 127.

Although not required, specifying the precision and scale for numeric fields provides extra integrity checking on input. Table 5 – 1 shows examples of how data would be stored using different scale factors.

Notice that if a negative scale is specified, the actual data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (7,-2) means to round to the nearest hundreds, as shown in Table 5 – 1.

Input Data	Specified As	Stored As
7,456,123.89	NUMBER	7456123.89
7,456,123.89	NUMBER (9)	7456124
7,456,123.89	NUMBER (9,2)	7456123.89
7,456,123.89	NUMBER (9,1)	7456123.9
7,456,123.89	NUMBER (6)	(not accepted, exceeds precision)
7,456,123.89	NUMBER (7, -2)	7456100

Table 5 – 1 How Scale Factors Affect Numeric Data Storage

DATE Datatype

Use the DATE datatype to store point-in-time values (dates and times) in a table. The DATE datatype stores the century, year, month, day, hours, minutes, and seconds. Oracle can store dates in the Julian era, ranging from January 1, 4712 BCE through December 31, 4712 CE (Common Era). Unless BCE ('BC' in the format mask) is specifically used, CE date entries are the default.

Oracle uses its own internal format to store dates. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second. See Chapter 3 in the *Programmer's Guide to the Oracle Call Interface* for a complete description of the Oracle internal date format.

For input and output of dates, the standard Oracle default date format is DD-MON-YY, as in

```
'13-NOV-92'
```

To change this default date format on an instance-wide basis, use the NLS_DATE_FORMAT parameter. To change the format during a session, use the ALTER SESSION statement. To enter dates that are not in the current default date format, use the TO_DATE function with a format mask, as in

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

If the date format DD-MON-YY is used, YY indicates the year in the 20th century (for example, 31-DEC-92 is December 31, 1992). If you want to indicate years in any century other than the 20th century, use a different format mask, as shown above.

Time is stored in 24-hour format—HH:MM:SS. By default, the time in a date field is 12:00:00 A.M. if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO_DATE function with a format mask indicating the time portion, as in

```
INSERT INTO birthdays (bname, bday) VALUES
('ANNIE',TO_DATE('13-NOV-92 10:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

To compare dates that have time data, use the SQL function TRUNC if you want to ignore the time component. Use the SQL function SYSDATE to return the system date and time. The FIXED_DATE initialization parameter allows you to set SYSDATE to a constant; this can be useful for testing.

Using Julian Dates

Julian dates allow continuous dating from a common reference. (The epoch is 1 January 4712 BCE, so current dates are somewhere in the 2.4 million range.) A Julian date is nominally a non-integer, the fractional part being a portion of a day. Oracle uses a simplified approach that results in integer values. Julian dates can be calculated and interpreted differently; the calculation method used by Oracle results in a seven-digit number for dates most often used, as in 2448355 for 08-APR-1991.

Use the format mask 'J' with date functions (TO_DATE or TO_CHAR, but not TO_NUMBER) to convert date data into Julian dates. For example, the following query returns all dates in Julian date format:

```
SELECT TO_CHAR (hiredate, 'J') FROM emp;
```

Use the TO_NUMBER function to use Julian dates in calculations. You can use the TO_DATE function to enter Julian dates:

```
INSERT INTO emp (hiredate) VALUES (TO_DATE(2448921, 'J'));
```

Note: Oracle Julian dates might not be compatible with Julian dates generated by other date algorithms.

Date Arithmetic

Oracle date arithmetic takes into account the anomalies of the Western calendars used throughout history. For example, the switch from the Julian to the Gregorian calendar, 15 October 1582, eliminated the previous 10 days (05 October – 14 October). Additionally, the year 0 does not exist.

Missing dates can be entered into the database, but are ignored in date arithmetic and treated as the next date. For example, the next day after 04 October 1582 is 15 October 1582 and the day following 05 October 1582 is 16 Oct 1582.

Note: This discussion of date arithmetic may not apply to all countries' date standards (for example, Asian countries).

LONG Datatype

Columns defined as LONG can store variable-length character data containing up to 2 gigabytes of information. LONG columns have many of the characteristics of VARCHAR2 columns. The length of LONG values might be limited by the memory available on your computer.

Uses of LONG Data

Columns defined with the LONG datatype are used in the data dictionary to store the text of view definitions. You can use columns defined as LONG in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements.

Restrictions on LONG and LONG RAW Data

Though LONG (and LONG RAW) columns have many uses, there are some restrictions on their use:

- Only one LONG column is allowed per table.
- LONG columns cannot be indexed.
- LONG columns cannot appear in integrity constraints.
- LONG columns cannot be used in WHERE, GROUP BY, ORDER BY, or CONNECT BY clauses, or with the DISTINCT operator in SELECT statements.
- LONG columns cannot be referenced by SQL functions (such as SUBSTR or INSTR).
- LONG columns cannot be used in the SELECT list of a subquery or queries combined by set operators (UNION, UNION ALL, INTERSECT, or MINUS).
- LONG columns cannot be used in SQL expressions.

- LONG columns cannot be referenced when creating a table with a query (CREATE TABLE ... AS SELECT ...) or when inserting into a table (or view) with a query (INSERT INTO ... SELECT ...).
- A variable or argument of a PL/SQL program unit cannot be declared using the LONG datatype.



Suggestion: When you design tables containing LONG or LONG RAW data, you should place each LONG or LONG RAW column in a table separate from any other data associated with it, rather than storing the LONG or LONG RAW column and its associated data together in the same table. You can then relate the two tables with a referential integrity constraint. This design allows SQL statements that access only the associated data to avoid reading through LONG or LONG RAW data.

Example To store information on magazine articles, including the texts of each article, create two tables:

```
CREATE TABLE article_header (id          NUMBER
                                PRIMARY KEY,
                                title       VARCHAR2(200),
                                first_author VARCHAR2(30),
                                journal      VARCHAR2(50),
                                pub_date    DATE)
CREATE TABLE article_text  (id          NUMBER
                                REFERENCES
                                article_header,
                                text       LONG)
```

The ARTICLE_TEXT table stores only the text of each article. The ARTICLE_HEADER table stores all other information about the article, including the title, first author, and journal and date of publication. The tables are related by the referential integrity constraint on the ID column of each table.

This design allows SQL statements to query data other than the text of an article without reading through the text. If you want to select all first authors published in *Nature* magazine during July 1991, you can issue this statement that queries the ARTICLE_HEADER table:

```
SELECT first_author
FROM article_header
WHERE journal = 'NATURE'
      AND TO_CHAR(pub_date, 'MM YYYY') = '07 1991')
```

If the text of each article were stored in the same table with the first author, publication, and publication date, Oracle would have to read through the text to perform this query.

RAW and LONG RAW Datatypes

Use the RAW and LONG RAW datatypes for data that is not to be interpreted (not to be converted when moving data between different systems) by Oracle. These datatypes are intended for binary data and byte strings. For example, LONG RAW can be used to store graphics, sound, documents, and arrays of binary data; the interpretation is dependent on the use.

Like the character datatype VARCHAR2, RAW and LONG RAW are variable length; however, the length of RAW cannot exceed 255 bytes, and there is no character set conversion done for RAW or LONG RAW data. CHAR, VARCHAR2, and LONG data is automatically converted from the database character set to the character set defined for the user session by the NLS_LANG parameter, where these are different.

When Oracle automatically converts RAW or LONG RAW data to and from CHAR data (as is the case when entering RAW data as a literal in an INSERT statement), the data is represented as one hexadecimal character representing the bit pattern for every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

LONG RAW data cannot be indexed, but RAW data can be indexed.

ROWIDs and the ROWID Datatype

Every row in a non-clustered table of an Oracle database is assigned a unique *ROWID* that corresponds to the physical address of a row's row piece (initial row piece if the row is chained among multiple row pieces). In the case of clustered tables, rows in different tables that are in the same data block can have the same ROWID.

Each table in an Oracle database internally has a *pseudo-column* named ROWID; this pseudo-column is not evident when listing the structure of a table by executing a SELECT * FROM . . . statement, or a DESCRIBE . . . statement using SQL*Plus. However, each row's address can be retrieved with a SQL query using the reserved word ROWID as a column name, as in

```
SELECT ROWID, ename FROM emp;
```

ROWIDs use a binary representation of the physical address for each row selected. When queried using SQL*Plus or SQL*DBA, the binary representation is converted to a VARCHAR2/hexadecimal representation. The above query might return the following row information:

ROWID	ENAME
00000DD5.0000.0001	SMITH
00000DD5.0001.0001	ALLEN
00000DD5.0002.0001	WARD

As shown above, a ROWID's VARCHAR2/hexadecimal representation is divided into three pieces: *block.row.file*.

- The *data block* that contains the row (block DD5 in the example). Block numbers are relative to their datafile, *not* tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- The *row* in the block that contains the row (rows 0, 1, 2 in the example). Row numbers of a given block always start with 0.
- The *datafile* that contains the row (file 1 in the example). The first datafile of every database is always 1, and file numbers are distinct within a database.

A row's assigned ROWID remains unchanged unless the row is exported and imported (using the IMPORT and EXPORT utilities). When you delete a row from a table (and commit the encompassing transaction), the deleted row's associated ROWID can be assigned to a row inserted in a subsequent transaction.

You cannot set the value of the pseudo-column ROWID in INSERT or UPDATE statements. The ROWIDs in the pseudo-column ROWID are used internally by Oracle for various operations (see the next section). Though you can reference ROWIDs in the pseudo-column ROWID like other table columns (used in SELECT lists and WHERE clauses), ROWIDs in this pseudo-column are not stored in the database, nor are they database data.

ROWIDs and Non-Oracle Databases To execute Oracle database applications against non-Oracle database servers, use the Oracle Open Gateway. In such cases, the binary format of ROWIDs varies according to the characteristics of the non-Oracle system. Furthermore, no standard translation to VARCHAR2/hexadecimal format is available. Programs can still use the ROWID datatype; however, they must use a non-standard translation to hexadecimal format of length up to 256 bytes. Refer to the relevant manual for OCIs or Precompilers for further details on handling ROWIDs with non-Oracle systems.

How ROWIDs Are Used

ROWIDs are used internally by Oracle in the construction of indexes. Each key in an index is associated with a ROWID that points to the associated row's address for fast access. Some of the characteristics of ROWIDs include the following:

- ROWIDs are the fastest means of accessing particular rows.
- ROWIDs can be used to see how a table is organized.
- ROWIDs are unique identifiers for rows in a given table.

Before using ROWIDs in DML statements, ROWIDs should be verified and guaranteed not to change; that is, the intended rows should be locked so they cannot be deleted. Attempting to retrieve a row using an invalid ROWID results in either no row being returned or error 1410, invalid ROWID.

You can also create tables with columns defined using the ROWID datatype; for example, you might define an exception table with a column of datatype ROWID to store the ROWIDs of rows in the database that violate integrity constraints. Such columns defined using the ROWID datatype behave like other table columns; that is, values can be updated, etc. All values in a column defined as datatype ROWID require six bytes to store pertinent column data.

Examples of Using ROWIDs

Using some group functions with ROWID, you can see how data is internally stored in an Oracle database. The function SUBSTR can be used to break the data in ROWID into its three components (file, block, and row). For example, the query

```
SELECT ROWID, SUBSTR(ROWID,15,4) "FILE",
       SUBSTR(ROWID,1,8) "BLOCK",
       SUBSTR(ROWID,10,4) "ROW"
FROM emp;
```

might return the following data:

ROWID	FILE	BLOCK	ROW
-----	----	-----	----
00000DD5.0000.0001	0001	00000DD5	0000
00000DD5.0001.0001	0001	00000DD5	0001
00000DD5.0002.0001	0001	00000DD5	0002

ROWIDs can be useful to reveal information about the physical storage of a table's data. For example, if you are interested in the physical location of a table's rows (for example, for table striping), the following query tells how many datafiles contain rows of a given table:

```
SELECT COUNT(DISTINCT(SUBSTR(ROWID,15,4))) "FILES" FROM tablename;
```

which might return

FILES

Summary of Oracle Datatype Information

For quick reference, Table 5 – 2 summarizes the important information about each Oracle datatype.

Datatype	Description
CHAR (size)	Fixed-length character data of length <i>size</i> . Maximum size is 255 bytes. Default size is 1 byte.
VARCHAR2 (size)	Variable-length character data. Maximum <i>size</i> is 2000 bytes.
FLOAT (p)	A floating-point number with binary precision <i>p</i> . FLOAT with no precision is the same as FLOAT (126).
NUMBER (p, s)	Variable-length numeric data. The precision <i>p</i> (total number of digits) can range from 1 to 38. The scale is <i>s</i> (number of decimal places) and can range from –84 to 127.
DATE	Fixed-length date and time data, ranging from January 1, 4712 BC to December 31, 4712 AD. Default format is the value of the NLS_DATE_FORMAT or ALTER SESSION parameter.
LONG	Variable-length character data up to 2 ³¹ – 1 bytes – 1, or 2 gigabytes – 1 byte.
RAW (size)	Variable-length raw binary data. A maximum size must be specified, up to 255.
LONG RAW	Variable-length raw binary data up to 2 ³¹ – 1 bytes – 1, or 2 gigabytes – 1 byte.
ROWID	Binary data representing row addresses.
MLSLABEL	Variable-length tag (2 – 5 bytes) that maps to a binary operating system label. For use with Trusted Oracle; see page 5 – 13 for details.

Table 5 – 2 Summary of Oracle Datatype Information

Trusted Oracle MLSLABEL Datatype

Trusted Oracle provides one new datatype: the MLSLABEL datatype. You can declare columns of the MLSLABEL datatype in standard Oracle, as well as Trusted Oracle, for compatibility with Trusted Oracle applications. In Oracle7, these columns contain nulls.

The MLSLABEL datatype is used to store the binary format of an operating system label. The maximum width of a column declared as MLSLABEL is 255 bytes.

Any labels that are valid on your operating system can be inserted into an MLSLABEL column. When you insert a label into an MLSLABEL column, Trusted Oracle implicitly converts the data into the binary format of the label.

The ALL_LABELS Data Dictionary View

The ALL_LABELS data dictionary view lists all of the labels ever stored in the database, including the values of DBHIGH and DBLOW. Any label ever stored in an MLSLABEL column (including the ROWLABEL column) is automatically added to this view.

Note that this view does not necessarily contain all labels that are valid in the database, since any valid operating system label, in any valid format, is a valid label within Trusted Oracle. Also note that this view may contain labels that are invalid within the database (if those labels were once used in the database, but are no longer valid).

Adding New Labels

If a label is not already in the ALL_LABELS view, and it is a valid operating system label, it is automatically added to the data dictionary as a valid label when you specify it in an INSERT or UPDATE statement.

For example, if TRULY_SENSITIVE:ALPHA, SENSITIVE, and UNCLASSIFIED are the only labels in the ALL_LABELS view, you can enter SENSITIVE:ALPHA into a column declared as MLSLABEL as long as SENSITIVE:ALPHA is a valid label in your operating system. Once inserted, this label automatically becomes a valid label in Trusted Oracle and is displayed in the ALL_LABELS view.

Trusted Oracle ROWLABEL Column

The ROWLABEL column is automatically appended to each Trusted Oracle table at table creation. This column contains a label of the MLSLABEL datatype for every row in the table.

In OS MAC mode, given that a table can contain rows at one label only, the values in this column are always uniform within a table (and within a single database).

In DBMS MAC mode, the values in this column can range within a single table from DBHIGH to DBLOW (within any constraints defined for that table).

Retrieving Row Labels

The ROWLABEL column is not automatically returned as part of a query. To retrieve the label of a row, you must explicitly select the ROWLABEL column.

For example, to retrieve the label of a row from a table or view, specify the ROWLABEL column in the SELECT statement:

```
SELECT rowlabel, ename FROM emp
      WHERE ename = 'JASUJA'
```

which returns

ROWLABEL	ENAME
SENSITIVE	JASUJA

You can also specify the ROWLABEL pseudo-column in the WHERE clause of a SELECT statement:

```
SELECT rowlabel,ename FROM emp
      WHERE rowlabel = 'SENSITIVE'
```

which returns

ROWLABEL	ENAME
SENSITIVE	JASUJA
SENSITIVE	ASHER

Retrieving All Labels From a Table

Note that when you select all columns from a table (SELECT *), the ROWLABEL column is not returned. You must explicitly specify that you want to retrieve the label in order to retrieve it. For example, to retrieve all columns, including the ROWLABEL column, from the DEPT table, enter the following:

```
SELECT rowlabel, dept.*
      FROM dept
```

This provides for compatibility with older applications (for example, Oracle RDBMS Version 6) or those designed to run in single-level and multi-level environments. However, the values for ROWLABEL are always utilized for MAC enforcement, and are always available in Trusted Oracle for retrieval and display when specified.

Retrieving Labels from Multiple Tables

To retrieve the label from more than one table or view, you must preface the ROWLABEL column with the table or view name. For example, to retrieve the ROWLABEL column from the EMP and DEPT tables wherever there is a match in department number, enter:

```
SELECT emp.rowlabel, dept.rowlabel
       FROM emp, dept
       WHERE emp.deptno = dept.deptno
```

If specifying the names of multiple tables becomes too lengthy, you can create synonyms for the table names within the query itself.

For example, the following query is identical to the query above:

```
SELECT e.rowlabel, d.rowlabel
       FROM emp e, dept d
       WHERE e.deptno = d.deptno
```

Modifying Row Labels

You can specify any valid operating system label in the ROWLABEL column, as described in “Adding New Labels” on page 5 – 14.

In OS MAC mode, you can perform inserts and updates to the ROWLABEL column under certain restrictions. In DBMS MAC mode, you can modify the ROWLABEL column if you have the appropriate MAC privileges

For detailed information on modifying the ROWLABEL column, see your *Trusted Oracle7 Server Administrator's Guide*.

Displaying Your DBMS Label

Your DBMS label is the label at which you are connected to Trusted Oracle. In OS MAC mode, this is always equivalent to the label of the database to which you are connected and always equivalent to your operating system session label. In DBMS MAC mode, this is the label with which you are connected to Trusted Oracle; because privileged users can alter their DBMS labels, this does not necessarily equal the label of your operating system session.

You can display your DBMS label in any of the following ways:

- The USERENV function
- The SHOW command in SQL*DBA

Displaying Your Label with the USERENV Function

To retrieve your DBMS label in any of the interactive tools produced by Oracle Corporation (including SQL*Plus and SQL*DBA), use the LABEL parameter of the USERENV function.

For example, assume that you are logged in to the database at SENSITIVE. To determine your DBMS label, enter

```
SELECT USERENV('label') FROM dual
```

which returns the following:

```
ROWLABEL
-----
SENSITIVE
```

Displaying Your Label with the Server Manager SHOW Command

To display your label in Server Manager, enter

```
SVRMGR> SHOW LABEL
```

which returns the following:

```
Label                                SENSITIVE
```

Altering Your DBMS Label

In DBMS MAC mode, you can alter your DBMS label with the SET LABEL parameter of the ALTER SESSION command. See your *Trusted Oracle7 Server Administrator's Guide* for more information about using this command.

The ALTER SESSION SET LABEL command is valid in OS MAC mode; however, as you cannot set your DBMS label to any label other than that of the database to which you are connected, the command is superfluous.

ANSI/ISO, DB2, and SQL/DS Datatypes

In addition to Oracle datatypes, you can define columns of tables in an Oracle database using ANSI/ISO, DB2, and SQL/DS datatypes. However, Oracle internally converts such datatypes to Oracle datatypes. The ANSI datatype conversions to Oracle datatypes are shown in Table 5 – 3; Table 5 – 4 shows the DB2 and SQL/DS conversions.

ANSI SQL Datatype	Oracle Datatype
CHARACTER (n), CHAR (n)	CHAR (n)
NUMERIC (p,s), DECIMAL (p,s), DEC (p,s)	NUMBER (p,s)
INTEGER, INT, SMALLINT	NUMBER (38)
FLOAT (p)	FLOAT (p)
REAL	FLOAT (63)
DOUBLE PRECISION	FLOAT (126)
CHARACTER VARYING(n), CHAR VARYING(n)	VARCHAR2 (n)

Table 5 – 3 ANSI Datatype Conversions to Oracle Datatypes

DB2 or SQL/DS Datatype	Oracle Datatype
CHARACTER (n)	CHAR (n)
VARCHAR (n)	VARCHAR2 (n)
LONG VARCHAR	LONG
DECIMAL (p,s)	NUMBER (p,s)
INTEGER, SMALLINT	NUMBER (38)
FLOAT (p)	FLOAT (p)
DATE	DATE

Table 5 – 4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes

The IBM products SQL/DS, and DB2 datatypes TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC have no corresponding Oracle datatype and cannot be used. The TIME and TIMESTAMP datatypes are subcomponents of the Oracle datatype DATE.

The ANSI/ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, s defaults to 0.

Data Conversion

In some cases, Oracle allows data of one datatype where it expects data of a different datatype. Generally, an expression cannot contain values with different datatypes. However, Oracle can use the following functions to automatically convert data to the expected datatype:

- TO_NUMBER()
- TO_CHAR()
- TO_DATE()
- HEXTORAW()
- RAWTOHEX()
- ROWIDTOCHAR()
- CHARTOROWID()

Implicit datatype conversions work according to the rules explained in the following two sections.

Note: In this discussion, CHAR is used to refer to the class of all character string datatypes.

If using Trusted Oracle, see page 5 – 13 for additional information involving data conversions and the MLSLABEL and RAW MLSLABEL datatypes.

Rule 1: Assignments

For assignments, Oracle can perform the following conversions automatically

- CHAR to NUMBER
- CHAR to DATE
- NUMBER to CHAR
- DATE to CHAR
- CHAR to RAW
- RAW to CHAR
- CHAR to ROWID
- ROWID to CHAR
- RAW to HEX
- HEX to RAW

The assignment succeeds if Oracle can convert the datatype of the value used in the assignment to that of the assignment's target. The four types of assignments are explained below.

For the examples in the following list, assume a package with a public variable declared as

```
var1 CHAR(5);
```

and a table created with the following statement:

```
CREATE TABLE table1 (col1 NUMBER);
```

- `variable := expression`

The datatype of *expression* must either be the same as, or convertible to, the datatype of *variable*. For example, Oracle automatically converts the data provided in the following assignment within the body of a stored procedure:

```
var1 := 0
```

- `INSERT INTO table VALUES (expression1, expression2, ...)`

The datatypes of *expression1*, *expression2*, etc., must either be the same as, or convertible to, the datatypes of corresponding columns in *table*. For example, Oracle automatically converts the data provided in the following INSERT statement for TABLE1 (see table definition above):

```
INSERT INTO table1 VALUES ('19');
```

- `UPDATE table SET column = expression`

The datatype of *expression* must either be the same as, or convertible to, the datatype of *column*. For example, Oracle automatically converts the data provided in the following UPDATE statement issued against TABLE1:

```
UPDATE table1 SET col1 = '30';
```

- `SELECT column INTO variable FROM table`

The datatype of *column* must either be the same as, or convertible to, the datatype of *variable*. For example, Oracle automatically converts data selected from the table before assigning it to the variable in the following statement:

```
SELECT col1 INTO var1 FROM table1 WHERE col1 = 30;
```

Rule 2: Expression Evaluation

For expression evaluation, Oracle can automatically perform the same conversions as for assignments. An expression is converted to a type based on its context. For example, operands to arithmetic operators are converted to NUMBER and operands to string functions are converted to VARCHAR2.

Some common types of expressions follow:

- Simple expressions, such as

```
comm + '500'
```

- Boolean expressions, such as

```
bonus > sal / '10'
```

- Function and procedure calls, such as

```
MOD (counter, '2')
```

- WHERE clause conditions, such as

```
WHERE hiredate = TO_DATE('1993-01-01','yyyy-mm-dd')
```

- WHERE clause conditions, such as

```
WHERE rowid = '00000DC5.000C.0001'
```

In general, Oracle uses the rule for expression evaluation when a datatype conversion is needed in places not covered by the rule for assignment conversions.

In assignments of the form

```
variable := expression
```

Oracle first evaluates *expression* using the conversions covered by Rule 2; *expression* can be as simple or complex as desired. If it succeeds, the evaluation of *expression* results in a single value and datatype. Then, Oracle tries to assign this value to the assignment's target using Rule 1.

CHAR to NUMBER conversions only succeed if the character string represents a valid number. CHAR to DATE conversions only succeed if the character string satisfies the session default format, NLS_DATE_FORMAT.

Data Conversion for Trusted Oracle

In Trusted Oracle, labels are stored internally as compact binary structures. Trusted Oracle provides two functions that allow you to convert a label from external (character or human readable) to internal (binary) format, and vice versa. These functions are described in the following sections.

The TO_CHAR Function

You can use the TO_CHAR function to convert a label from its binary to character representation, as follows:

```
TO_CHAR(label [,format])
```

where label is the binary representation of a label and format is a valid label format (see the following section, “Formatting Labels” on page 5 – 23, for more details about this parameter).

This function is useful because it allows you to retrieve a label in any supported format that suits your needs.

The TO_LABEL Function

You can use the TO_LABEL function to convert a label from a character string to its internal, binary representation.

To convert a label from character string to binary representation, enter

```
TO_LABEL(string [,format])
```

where string is the character representation of a label and format is a valid label format (see the following section, “Formatting Labels” for more details about this parameter).

You must use the TO_LABEL function when converting a label from one external format to another (see “Formatting Labels” on page 5 – 23 for examples of how to use this function).

Formatting Labels

The default format in which Trusted Oracle returns a label may not necessarily be the most appropriate format for a given display, report, or application. To accommodate different formatting needs, Trusted Oracle provides the ability to

- tailor the output of the `TO_CHAR` and `TO_LABEL` functions in a format that best suits your needs
- set a system-wide default label format (the `MLS_LABEL_FORMAT` initialization parameter)
- set a session default label format to override the system-wide default within a given application or session (the `ALTER SESSION SET MLS_LABEL_FORMAT` statement)

The following sections describe how to use these features in more detail.

The `TO_CHAR` and `TO_LABEL` Functions

Depending upon your operating system, a label can have several components: sensitivity, integrity, information, and installation defined. Both the sensitivity and integrity components consists of one classification and zero or more categories.

Using the format parameter of the `TO_CHAR` and `TO_LABEL` functions, you can format the classification and categories of each of these components in one of several ways: numeric, short, or long. You can also specify whether you want the numeric, short, or long representation of a label to be displayed in the format used by the operating system on which Trusted Oracle is running, and you can specify whether you want a label displayed in full operating system format. Note that not all operating systems support separate long and short formats.

Setting the DBMS Label Format for Your Session

If you want to display labels in a format other than the system-wide default, you can alter the default format for your session with the `SET MLS_LABEL_FORMAT` parameter of the `ALTER SESSION` command. You need no special privileges to execute this command.

For example, assume that the system-wide default label format (set in the parameter file) is SEN; this displays the full sensitivity label in short format. However, you want the full sensitivity label to be displayed in numeric format just for your session or a portion of your session. To change the label format for your DBMS session to S, enter

```
ALTER SESSION SET MLS_LABEL_FORMAT = 'S'
```

All labels will then be displayed in this format for the duration of your session, or until you issue another ALTER SESSION SET MLS_LABEL_FORMAT command.

Note that altering the MLS_LABEL_FORMAT parameter not only changes the format in which labels are displayed for your session; it also changes the format in which you must enter labels. For example, after altering the MLS_LABEL_FORMAT parameter as described in the above example, you must enter labels in numeric format for the rest of your session.

Changing the MLS_LABEL_FORMAT parameter does not affect constraint enforcement.

For more information about different label formats, see your *Trusted Oracle7 Server Administrator's Guide*.

Comparing Labels

In addition to formatting labels, you may need to perform comparison operations on labels. Trusted Oracle lets you use standard comparison operators to

- determine which of two or more labels dominates or is dominated by other labels
- determine if two or more labels are equal to each other

You can use the standard Oracle comparison operators to compare labels in Trusted Oracle.

Guidelines for Defining Labels

Labels and label formats vary from operating system to operating system. While your installation currently may not use more than one type of operating system, you should be aware of potential differences between labels and label formats in a distributed database configuration and for future porting considerations, particularly when writing applications that should be easily ported to other operating systems.

To promote portability between labels in databases from one operating system to another, you should consider the following guidelines when defining labels in your database.

Embedded Spaces and Punctuation

Be cautious when using embedded spaces and punctuation in labels, as not all operating systems can interpret them.

For example, a label of SENSITIVE:ALPHA/BETA may be correctly interpreted on one operating system, but not correctly interpreted by other operating systems.

Case Sensitivity

Use care when specifying labels using upper and lowercase, as some operating systems are case sensitive and some are not. You should avoid defining labels that are distinguished only by case (for those operating systems that are not case sensitive); in addition, you should avoid comparing labels for an exact string match, including case (for those operating systems that are).

For example, one operating system may interpret Truly_Sensitive and TRULY_SENSITIVE as different labels; however, an operating system that is not case sensitive would interpret them as identical.

Label Components

Because not all operating systems support the integrity, information, and installation-defined portions of a label, you should use care when defining these in Trusted Oracle or using them in applications.

Note that if you request a label component that is not supported by your operating system, Trusted Oracle will return a null value for that component.

Numeric Format

You should avoid using '0' to represent either a classification or a category in numeric format, as some operating systems begin numbering with '0' and some with '1'.

Number of Classifications and Categories

Be aware that the number of classifications and categories supported by operating systems differs. While most operating systems support between 16 and 256 hierarchical classifications, a few may fall above or below this range.

Maintaining Data Integrity

This chapter explains how to enforce the business rules associated with your database and prevent the entry of invalid information into tables by using integrity constraints. Topics include the following:

- using integrity constraints
- using referential integrity constraints
- referential integrity in a distributed database
- using CHECK integrity constraints
- defining integrity constraints
- enabling and disabling integrity constraints
- altering and dropping integrity constraints
- managing FOREIGN KEY constraints
- listing constraint definitions

See the *Trusted Oracle7 Server Administrator's Guide* for additional information about defining, enabling, disabling, and dropping integrity constraints in Trusted Oracle.

Using Integrity Constraints

You can define integrity constraints to enforce business rules on data in your tables. Once an integrity constraint is enabled, all data in the table must conform to the rule that it specifies. If you subsequently issue a SQL statement that modifies data in the table, Oracle ensures that the resulting data satisfies the integrity constraint. Without integrity constraints, such business rules must be enforced programmatically by your application.

When to Use Integrity Constraints

Enforcing rules with integrity constraints is less costly than enforcing the equivalent rules by issuing SQL statements in your application. The semantics of integrity constraints are very clearly defined, so the internal operations that Oracle performs to enforce them are optimized beneath the level of SQL statements in Oracle. Since your applications use SQL, they cannot achieve this level of optimization.

Enforcing business rules with SQL statements can be even more costly in a networked environment because the SQL statements must be transmitted over a network. In such cases, using integrity constraints eliminates the performance overhead incurred by this transmission.

Example To ensure that each employee in the EMP table works for a department that is listed in the DEPT table, first create a PRIMARY KEY constraint on the DEPTNO column of the DEPT table with this statement:

```
ALTER TABLE dept
  ADD PRIMARY KEY (deptno)
```

Then create a referential integrity constraint on the DEPTNO column of the EMP table that references the primary key of the DEPT table:

```
ALTER TABLE emp
  ADD FOREIGN KEY (deptno) REFERENCES dept(deptno)
```

If you subsequently add a new employee record to the table, Oracle automatically ensures that its department number appears in the department table.

To enforce this rule without integrity constraints, your application must test each new employee record to ensure that its department number belongs to an existing department. This testing involves issuing a SELECT statement to query the DEPT table.

Taking Advantage of Integrity Constraints

For best performance, define and enable integrity constraints and develop your applications to rely on them, rather than on SQL statements in your applications, to enforce business rules.

However, in some cases, you might want to enforce business rules through your application as well as through integrity constraints. Enforcing a business rule in your application might provide faster feedback to the user than an integrity constraint. For example, if your application accepts 20 values from the user and then issues an INSERT statement containing these values, you might want your user to be notified immediately after entering a value that violates a business rule.

Since integrity constraints are enforced only when a SQL statement is issued, an integrity constraint can only notify the user of a bad value after the user has entered all 20 values and the application has issued the INSERT statement. However, you can design your application to verify the integrity of each value as it is entered and notify the user immediately in the event of a bad value.

Using NOT NULL Integrity Constraints

By default, all columns can contain nulls. Only define NOT NULL constraints for columns of a table that absolutely require values at all times.

For example, in the EMP table, it might not be detrimental if an employee's manager or hire date were temporarily omitted. Also, some employees might not have a commission. Therefore, these three columns would not be good candidates for NOT NULL integrity constraints. However, it might not be permitted to have a row that does not have an employee name. Therefore, this column is a good candidate for the use of a NOT NULL integrity constraint.

NOT NULL constraints are often combined with other types of integrity constraints to further restrict the values that can exist in specific columns of a table. Use the combination of NOT NULL and UNIQUE key integrity constraints to force the input of values in the UNIQUE key; this combination of data integrity rules eliminates the possibility that any new row's data will ever attempt to conflict with an existing row's data. For more information about such combinations, see "Relationships Between Parent and Child Tables" on page 6 – 8.

Table EMP							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17-DEC-85	9,000.00		20
7499	ALLEN	VP-SALES	7329	20-FEB-90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22-FEB-90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02-APR-90	2,975.00	400.00	30

NOT NULL Constraint
(no row may contain a null value for this column)

Absence of NOT NULL Constraint
(any row can contain a null for this column)

Figure 6 – 1 NOT NULL Integrity Constraints

Setting Default Column Values

Legal default values include any literal, or any expression that does not refer to a column, LEVEL, ROWNUM, or PRIOR. Default values can include the expressions SYSDATE, USER, USERENV, and UID. The datatype of the default literal or expression must match or be convertible to the column datatype.

If you do not explicitly define a default value for a column, the default for the column is implicitly set to NULL.

When to Use Default Values

Only assign default values to columns that contain a typical value. For example, in the DEPT table, if most departments are located at one site, the default value for the LOC column can be set to this value (such as NEW YORK).

Defaults are also useful when you use a view to make a subset of a table’s columns visible. For example, you might allow users to insert rows into a table through a view. The view is defined to show all columns pertinent to end–user operations; however, the base table might also have a column named INSERTER, not included in the definition of the view, which logs the user that originally inserts each row of the table. The column named INSERTER can record the name of the user that inserts a row by defining the column with the USER function:

```
. . . , inserter VARCHAR2(30) DEFAULT USER, . . .
```

For another example of assigning a default column value, refer to the section “Creating Tables” on page 4 – 3.

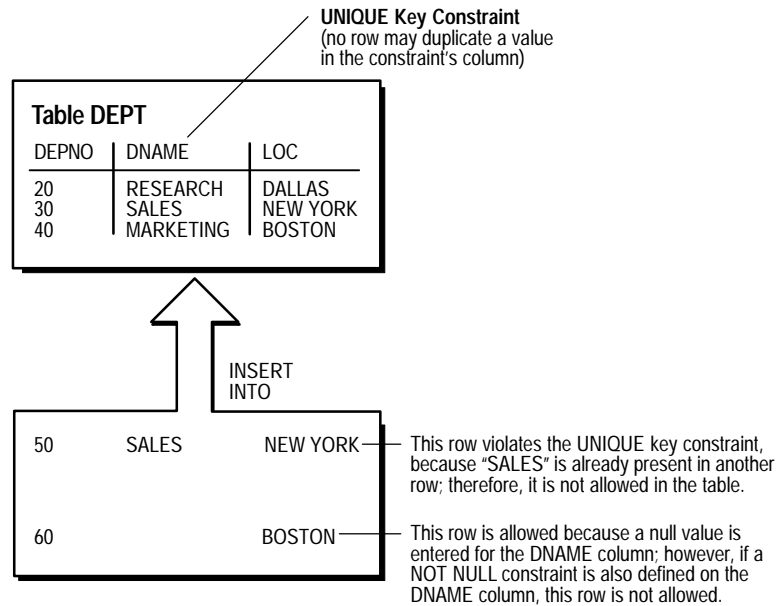


Figure 6 – 2 A UNIQUE Key Constraint

Choosing a Table's Primary Key

Each table can have one primary key. A primary key allows each row in a table to be uniquely identified and ensures that no duplicate rows exist. Use the following guidelines when selecting a primary key:

- Choose a column whose data values are unique.

The purpose of a table's primary key is to uniquely identify each row of the table. Therefore, the column or set of columns in the primary key must contain unique values for each row.

- Choose a column whose data values are never changed.

A primary key value is only used to identify a row in the table; primary key values should never contain any data that is used for any other purpose. Therefore, primary key values should rarely need to be changed.

- Choose a column that does not contain any nulls.

A PRIMARY KEY constraint, by definition, does not allow the input of any row with a null in any column that is part of the primary key.

- Choose a column that is short and numeric.

Short primary keys are easy to type. You can use sequence numbers to easily generate numeric primary keys.

- Avoid choosing composite primary keys.

Although composite primary keys are allowed, they do not satisfy the previous recommendations. For example, composite primary key values are long and cannot be assigned by sequence numbers.

Using UNIQUE Key Integrity Constraints

Choose unique keys carefully. In many situations, unique keys are incorrectly comprised of columns that should be part of the table's primary key (see the previous section for more information about primary keys). When deciding whether to use a UNIQUE key constraint, use the rule that a UNIQUE key constraint is only required to prevent the duplication of the key values within the rows of the table. The data in a unique key is such that it cannot be duplicated in the table.

Note: Although UNIQUE key constraints allow the input of nulls, because of the search mechanism for UNIQUE constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite UNIQUE key constraint.

Do not confuse the concept of a unique key with that of a primary key. Primary keys are used to identify each row of the table uniquely. Therefore, unique keys should not have the purpose of identifying rows in the table.

Some examples of good unique keys include

- an employee's social security number (the primary key is the employee number)
- a truck's license plate number (the primary key is the truck number)
- a customer's phone number, consisting of the two columns AREA and PHONE (the primary key is the customer number)
- a department's name and location (the primary key is the department number)

Using Referential Integrity Constraints

Whenever two tables are related by a common column (or set of columns), define a PRIMARY or UNIQUE key constraint on the column in the parent table, and define a FOREIGN KEY constraint on the column in the child table, to maintain the relationship between the two tables. Depending on this relationship, you may want to define additional integrity constraints including the foreign key, as listed in the section “Relationships Between Parent and Child Tables”, on page 6 – 8.

Figure 6 – 3 shows a foreign key defined on the DEPTNO column of the EMP table. It guarantees that every value in this column must match a value in the primary key of the DEPT table (the DEPTNO column); therefore, no erroneous department numbers can exist in the DEPTNO column of the EMP table.

Foreign keys can be comprised of multiple columns. However, a composite foreign key must reference a composite primary or unique key of the exact same structure; that is, the same number of columns and datatypes. Because composite primary and unique keys are limited to 16 columns, a composite foreign key is also limited to 16 columns.

Nulls and Foreign Keys

By default (that is, without any NOT NULL or CHECK clauses), and in accordance with the ANSI/ISO standard, the FOREIGN KEY constraint enforces the “match none” rule for composite foreign keys. The “full” and “partial” rules can also be enforced by using CHECK and NOT NULL constraints, as follows:

- To enforce the “match full” rule for nulls in composite foreign keys, which requires that all components of the key be null or all be non-null, define a CHECK constraint that allows only all nulls or all non-nulls in the composite foreign key as follows, assuming a composite key comprised of columns A, B, and C:

```
CHECK ((A IS NULL AND B IS NULL AND C IS NULL) OR  
(A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL))
```

- In general, it is not possible to use declarative referential integrity to enforce the “match partial” rule for nulls in composite foreign keys, which requires the non-null portions of the key to appear in the corresponding portions in the primary or unique key of a single row in the referenced table. You can often use triggers to handle this case, as described in Chapter 9.

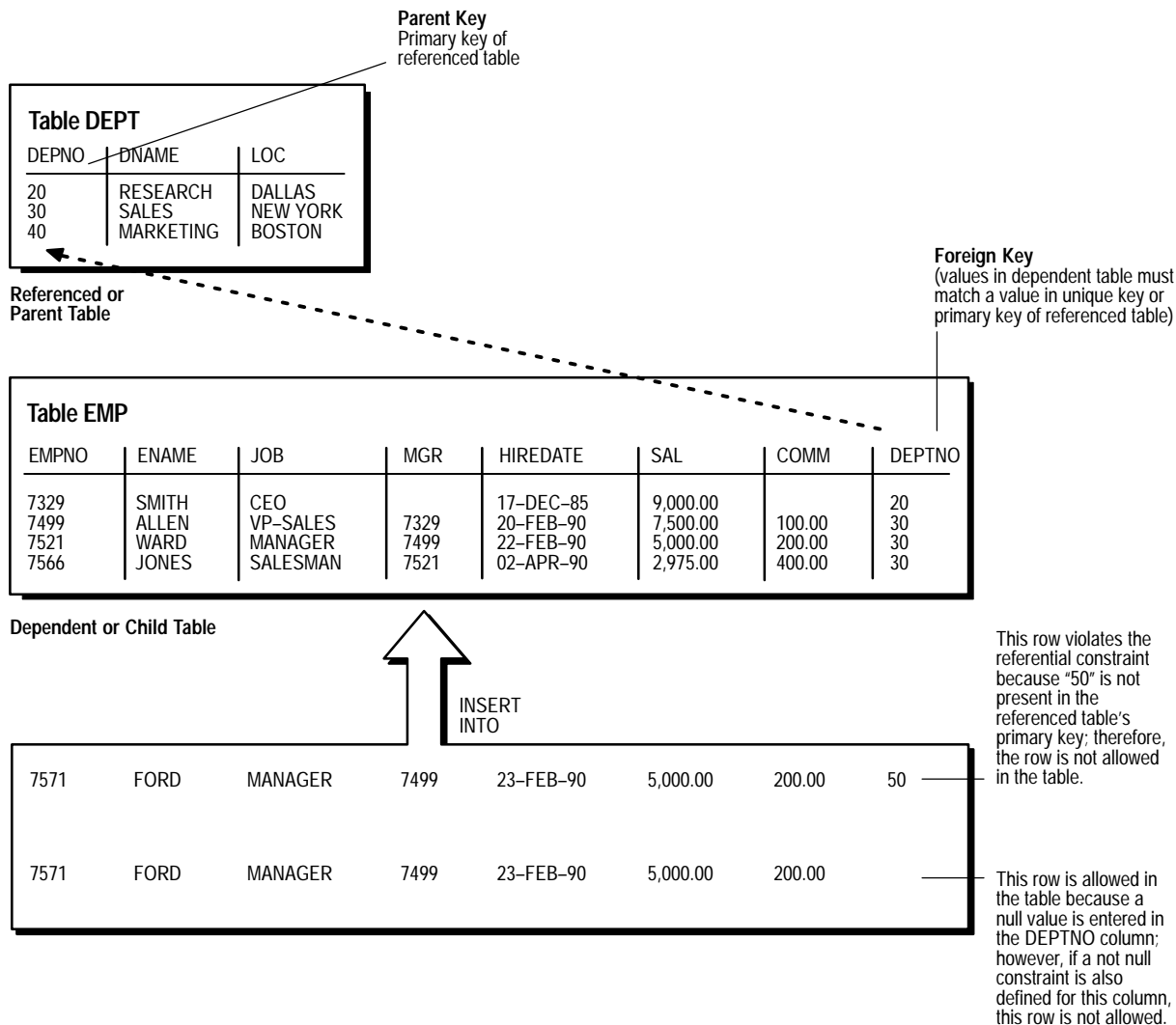


Figure 6 – 3 Referential Integrity Constraints

Relationships Between Parent and Child Tables

Several relationships between parent and child tables can be determined by the other types of integrity constraints defined on the foreign key in the child table.

No Constraints on the Foreign Key When no other constraints are defined on the foreign key, any number of rows in the child table can reference the same parent key value. This model allows nulls in the foreign key.

This model establishes a “one-to-many” relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. An example of such a relationship is shown in Figure 6 – 3 between EMP and DEPT; each department (parent key) has many employees (foreign key), and some employees might not be in a department (nulls in the foreign key).

NOT NULL Constraint on the Foreign Key When nulls are not allowed in a foreign key, each row in the child table must explicitly reference a value in the parent key because nulls are not allowed in the foreign key. However, any number of rows in the child table can reference the same parent key value.

This model establishes a “one-to-many” relationship between the parent and foreign keys. However, each row in the child table must have a reference to a parent key value; the absence of a value (a null) in the foreign key is not allowed. The same example in the previous section can be used to illustrate such a relationship. However, in this case, employees must have a reference to a specific department.

UNIQUE Constraint on the Foreign Key When a UNIQUE constraint is defined on the foreign key, one row in the child table can reference a parent key value. This model allows nulls in the foreign key.

This model establishes a “one-to-one” relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. For example, assume that the EMP table had a column named MEMBERNO, referring to an employee’s membership number in the company’s insurance plan. Also, a table named INSURANCE has a primary key named MEMBERNO, and other columns of the table keep respective information relating to an employee’s insurance policy. The MEMBERNO in the EMP table should be both a foreign key and a unique key:

- to enforce referential integrity rules between the EMP and INSURANCE tables (the FOREIGN KEY constraint)
- to guarantee that each employee has a unique membership number (the UNIQUE key constraint)

UNIQUE and NOT NULL Constraints on the Foreign Key When both UNIQUE and NOT NULL constraints are defined on the foreign key, only one row in the child table can reference a parent key value.

Because nulls are not allowed in the foreign key, each row in the child table must explicitly reference a value in the parent key.

This model establishes a “one-to-one” relationship between the parent and foreign keys that does not allow undetermined values (nulls) in the foreign key. If you expand the previous example by adding a NOT NULL constraint on the MEMBERNO column of the EMP table, in addition to guaranteeing that each employee has a unique membership number, you also ensure that no undetermined values (nulls) are allowed in the MEMBERNO column of the EMP table.

Multiple FOREIGN KEY Constraints

Oracle allows a column to be referenced by multiple FOREIGN KEY constraints; effectively, there is no limit on the number of dependent keys. This situation might be present if a single column is part of two different composite foreign keys.

Concurrency Control, Indexes, and Foreign Keys

Oracle maximizes the concurrency control of parent keys in relation to dependent foreign key values. You can control what concurrency mechanisms are used to maintain these relationships and, depending on the situation, this can be highly beneficial. The following sections explain the possible situations and give recommendations for each.

No Index on the Foreign Key Figure 6 – 4 illustrates the locking mechanisms used by Oracle when no index is defined on the foreign key and when rows are being updated or deleted in the parent table. Inserts into the parent table do not require any locks on the child table.

Notice that a share lock of the entire child table is required until the transaction containing the DELETE statement for the parent table is committed. If the foreign key specifies ON DELETE CASCADE, the DELETE statement results in a table-level share-subexclusive lock on the child table. A share lock of the entire child table is also required for an UPDATE statement on the parent table that affects any columns referenced by the child table. Share locks allow reading only; therefore, no INSERT, UPDATE, or DELETE statements can be issued on the child table until the transaction containing the UPDATE or DELETE is committed. Queries are allowed on the child table.

This situation is tolerable if updates and deletes can be avoided on the parent.

INSERT, UPDATE, and DELETE statements on the child table do not acquire any locks on the parent table; although INSERT and UPDATE statements will wait for a row-lock on the index of the parent table to clear.

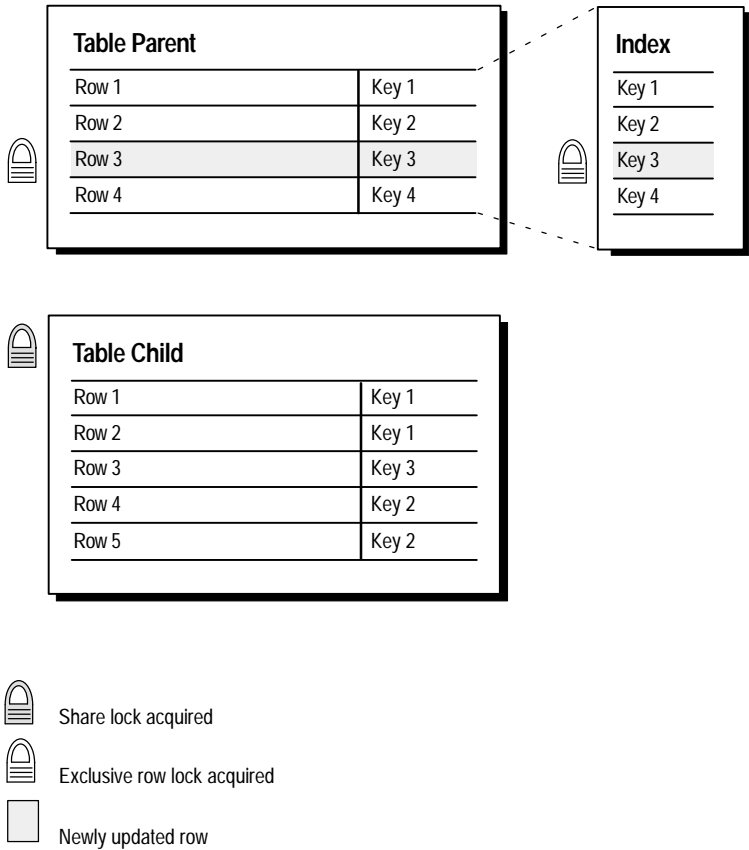


Figure 6 – 4 Locking mechanisms Used When No Index Is Defined on the Foreign Key

Index on the Foreign Key Figure 6 – 5 illustrates the locking mechanisms used by Oracle when an index is defined on the foreign key, and new rows are inserted, updated or deleted in the child table.

Notice that no table locks of any kind are acquired on the parent table or any of its indexes as a result of the insert, update or delete. Therefore, any type of DML statement can be issued on the parent table, including inserts, updates, deletes, and queries.

This situation is preferable if there is any update or delete activity on the parent table while update activity is taking place on the child table. Inserts, updates, and deletes on the parent table do not require any locks on the child table; although updates and deletes will wait for row-level locks on the indexes of the child table to clear.

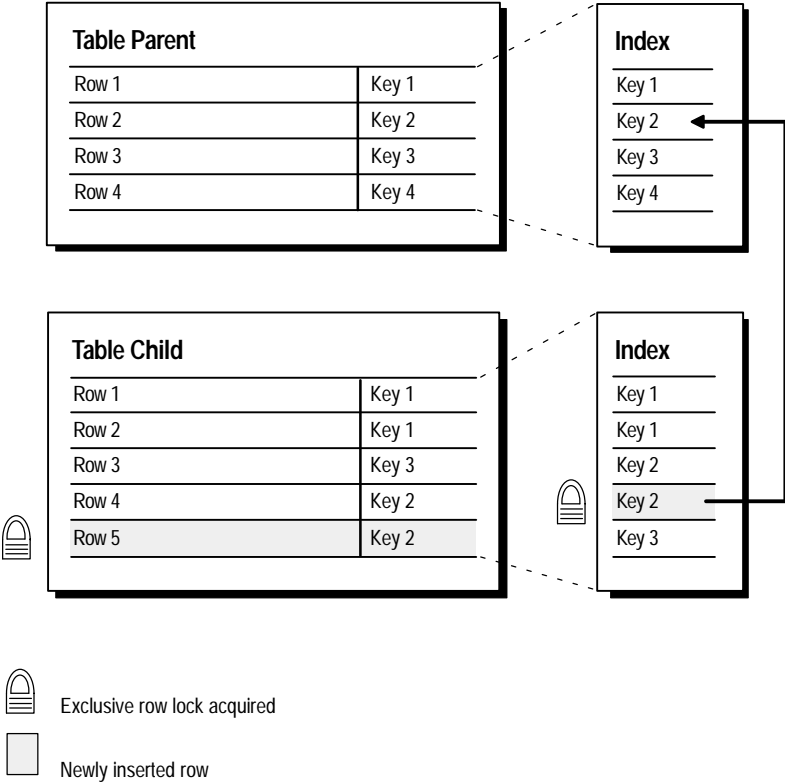


Figure 6 – 5 Locking mechanisms Used When Index Is Defined on the Foreign Key

If the child table specifies ON DELETE CASCADE, deletes from the parent table may result in deletes from the child table. In this case, waiting and locking rules are the same as if you deleted from the child table yourself after performing the delete from the parent table.

Referential Integrity in a Distributed Database

Oracle does not permit declarative referential integrity constraints to be defined across nodes of a distributed database (that is, a declarative referential integrity constraint on one table cannot specify a foreign key that references a primary or unique key of a remote table). However, parent/child table relationships across nodes can be maintained using triggers. For more information about triggers that enforce referential integrity, refer to Chapter 9. Using triggers to maintain referential integrity requires the distributed option; for more information refer to *Oracle7 Server Distributed Systems, Volume I*.

Note: If you decide to define referential integrity across the nodes of a distributed database using triggers, be aware that network failures can limit the accessibility of not only the parent table, but also the child table. For example, assume that the child table is in the SALES database and the parent table is in the HQ database. If the network connection between the two databases fails, some DML statements against the child table (those that insert rows into the child table or update a foreign key value in the child table) cannot proceed because the referential integrity triggers must have access to the parent table in the HQ database.

Using CHECK Integrity Constraints

Use CHECK constraints when you need to enforce integrity rules that can be evaluated based on logical expressions. Never use CHECK constraints when any of the other types of integrity constraints can provide the necessary checking (see the section “CHECK and NOT NULL Integrity Constraints” on page 6 – 15). Examples of appropriate CHECK constraints include the following:

- a CHECK constraint on the SAL column of the EMP table so that no salary value is greater than 10000
- a CHECK constraint on the LOC column of the DEPT table so that only the locations “BOSTON”, “NEW YORK”, and “DALLAS” are allowed
- a CHECK constraint on the SAL and COMM columns to compare the SAL and COMM values of a row and prevent the COMM value from being greater than the SAL value

Restrictions on CHECK Constraints

A CHECK integrity constraint requires that a condition be true or unknown for every row of the table. If a statement causes the condition to evaluate to false, the statement is rolled back. The condition of a CHECK constraint has the following limitations:

- The condition must be a Boolean expression that can be evaluated using the values in the row being inserted or updated.
- The condition cannot contain subqueries or sequences.
- The condition cannot include the SYSDATE, UID, USER, or USERENV SQL functions.
- The condition cannot contain the pseudocolumns LEVEL, PRIOR, or ROWNUM; see the *Oracle7 Server SQL Reference* manual for an explanation of these pseudocolumns.
- The condition cannot contain a user-defined SQL function.

Designing CHECK Constraints

When using CHECK constraints, consider the ANSI/ISO standard, which states that a CHECK constraint is violated only if the condition evaluates to false; true and unknown values do not violate a check condition. Therefore, make sure that a CHECK constraint that you define actually enforces the rule you need enforced.

For example, consider the following CHECK constraint:

```
CHECK (sal > 0 OR comm >= 0)
```

At first glance, this rule may be interpreted as “do not allow a row in the EMP table unless the employee’s salary is greater than zero or the employee’s commission is greater than or equal to zero.” However, note that if a row is inserted with a null salary and a negative commission, the row does not violate the CHECK constraint because the entire check condition is evaluated as unknown. In this particular case, you can account for such violations by placing NOT NULL integrity constraints on both the SAL and COMM columns.

Note: If you are not sure when unknown values result in NULL conditions, review the truth tables for the logical operators AND and OR in the *Oracle7 Server SQL Reference* manual.

Multiple CHECK Constraints

A single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number of CHECK constraints that can be defined that reference a column.

CHECK and NOT NULL Integrity Constraints

According to the ANSI/ISO standard, a NOT NULL integrity constraint is an example of a CHECK integrity constraint, where the condition is

```
CHECK (column_name IS NOT NULL)
```

Therefore, NOT NULL integrity constraints for a single column can, in practice, be written in two forms: using the NOT NULL constraint or a CHECK constraint. For ease of use, you should always choose to define NOT NULL integrity constraints instead of CHECK constraints with the “IS NOT NULL” condition.

In the case where a composite key can allow only all nulls or all values, you must use a CHECK integrity constraint. For example, the following expression of a CHECK integrity constraint allows a key value in the composite key made up of columns C1 and C2 to contain either all nulls or all values:

```
CHECK ((c1 IS NULL AND c2 IS NULL) OR  
       (c1 IS NOT NULL AND c2 IS NOT NULL))
```

Defining Integrity Constraints

Define an integrity constraint using the constraint clause of the SQL commands CREATE TABLE or ALTER TABLE. The next two sections describe how to use these commands to define integrity constraints.

There are additional considerations if you are using Trusted Oracle; see the *Trusted Oracle7 Server Administrator's Guide* for more information.

The CREATE TABLE Command

The following examples of CREATE TABLE statements show the definition of several integrity constraints:

```
CREATE TABLE dept (  
    deptno  NUMBER(3) PRIMARY KEY,  
    dname   VARCHAR2(15),  
    loc     VARCHAR2(15),  
           CONSTRAINT dname_ukey UNIQUE (dname, loc),  
           CONSTRAINT loc_check1  
             CHECK (loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));  
  
CREATE TABLE emp (  
    empno    NUMBER(5) PRIMARY KEY,  
    ename    VARCHAR2(15) NOT NULL,  
    job      VARCHAR2(10),  
    mgr      NUMBER(5) CONSTRAINT mgr_fkey  
           REFERENCES emp,  
    hiredate DATE,  
    sal      NUMBER(7,2),  
    comm     NUMBER(5,2),  
    deptno   NUMBER(3) NOT NULL  
           CONSTRAINT dept_fkey  
           REFERENCES dept ON DELETE CASCADE);
```

The ALTER TABLE Command

You can also define integrity constraints using the constraint clause of the ALTER TABLE command. For example, the following examples of ALTER TABLE statements show the definition of several integrity constraints:

```
ALTER TABLE dept  
    ADD PRIMARY KEY (deptno);  
  
ALTER TABLE emp  
    ADD CONSTRAINT dept_fkey FOREIGN KEY (deptno) REFERENCES dept  
    MODIFY (ename VARCHAR2(15) NOT NULL);
```

Restrictions with the ALTER TABLE Command

Because data is likely to be in the table at the time an ALTER TABLE statement is issued, there are several restrictions to be aware of. Table 6 – 1 lists each type of constraint and the associated restrictions with the ALTER TABLE command.

Type of Constraint	Added to Existing Columns of the Table	Added with New Columns to the Table
NOT NULL	Cannot be defined if any row contains a null value for this column*	Cannot be defined if the table contains any rows
UNIQUE	Cannot be defined if duplicate values exist in the key*	Always OK
PRIMARY KEY	Cannot be defined if duplicate or null values exist in the key*	Cannot be defined if the table contains any rows
FOREIGN KEY	Cannot be defined if the foreign key has values that do not reference a parent key value*	Always OK
CHECK	Cannot be defined if the volume has values that do not comply with the check condition*	Always OK

Table 6 – 1 Restrictions for Defining Integrity Constraints with the ALTER TABLE Command

** Assumes DISABLE clause not included in statement.*

If you attempt to define a constraint with an ALTER TABLE statement and violate one of these restrictions, the statement is rolled back and an informative error is returned explaining the violation.

Required Privileges

The creator of a constraint must have the ability to create tables (that is, the CREATE TABLE or CREATE ANY TABLE system privilege) or the ability to alter the table (that is, the ALTER object privilege for the table or the ALTER ANY TABLE system privilege) with the constraint. Additionally, UNIQUE key and PRIMARY KEY integrity constraints require that the owner of the table have either a quota for the tablespace that contains the associated index or the UNLIMITED TABLESPACE system privilege. FOREIGN KEY integrity constraints also require some additional privileges; see “Privileges Required for FOREIGN KEY Integrity Constraints” on page 6 – 26 for specific information.

Naming Integrity Constraints

Assign names to NOT NULL, UNIQUE KEY, PRIMARY KEY, FOREIGN KEY, and CHECK constraints using the CONSTRAINT option of the constraint clause. This name must be unique with respect to other constraints that you own. If you do not specify a constraint name, one is assigned by Oracle.

See the previous examples of the CREATE TABLE and ALTER TABLE statements for examples of the CONSTRAINT option of the Constraint clause. Note that the name of each constraint is included with other information about the constraint in the data dictionary. Refer to the section “Listing Integrity Constraint Definitions” on page 6 – 28 for examples of data dictionary views.

Enabling and Disabling Constraints Upon Definition

By default, whenever an integrity constraint is defined in a CREATE or ALTER TABLE statement, the constraint is automatically enabled (enforced) by Oracle unless it is specifically created in a disabled state using the DISABLE clause. Refer to the section “Enabling and Disabling Integrity Constraints” on page 6 – 18 for more information about important issues for enabling and disabling constraints.

UNIQUE Key, PRIMARY KEY, and FOREIGN KEY

When defining UNIQUE key, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. For more information about defining and managing FOREIGN KEY constraints, see “Managing FOREIGN KEY Integrity Constraints” on page 6 – 25. UNIQUE key and PRIMARY KEY constraints are usually enabled by the database administrator; see the *Oracle7 Server Administrator’s Guide* for more information.

Enabling and Disabling Integrity Constraints

This section explains the mechanisms and procedures for manually enabling and disabling integrity constraints.

enabled constraint	When a constraint is enabled, the rule defined by the constraint is enforced on the data values in the columns that define the constraint. The definition of the constraint is stored in the data dictionary.
disabled constraint	When a constraint is disabled, the rule defined by the constraint is not enforced on the data values in the columns included in the constraint; however, the definition of the constraint is retained in the data dictionary.

In summary, an integrity constraint can be thought of as a statement about the data in a database. This statement is always true when the constraint is enabled; however, the statement may or may not be true when the constraint is disabled because data in violation of the integrity constraint can be in the database.

Why Enable or Disable Constraints?

To enforce the rules defined by integrity constraints, the constraints should always be enabled; however, in certain situations, it is desirable to disable the integrity constraints of a table temporarily for performance reasons. For example:

- when loading large amounts of data into a table using SQL*Loader
- when performing batch operations that make massive changes to a table (such as changing everyone's employee number by adding 1000 to the existing number)
- when importing or exporting one table at a time

In cases such as these, integrity constraints may be temporarily turned off to improve the performance of the operation.

Integrity Constraint Violations

If a row of a table does not adhere to an integrity constraint, this row is said to be in violation of the constraint and is known as an *exception* to the constraint. If any exceptions exist, **the constraint cannot be enabled**. The rows that violate the constraint must be either updated or deleted in order for the constraint to be enabled.

Exceptions for a specific integrity constraint can be identified while attempting to enable the constraint. This procedure is discussed in the section "Exception Reporting" on page 6 – 23.

On Definition

When you define an integrity constraint in a CREATE TABLE or ALTER TABLE statement, you can enable the constraint by including the ENABLE clause in its definition or disable it by including the DISABLE clause in its definition. If neither the ENABLE nor the DISABLE clause is included in a constraint's definition, Oracle automatically enables the constraint.

Enabling Constraints

The following CREATE TABLE and ALTER TABLE statements both define and enable integrity constraints:

```
CREATE TABLE emp (  
    empno NUMBER(5) PRIMARY KEY,    . . . );  
  
ALTER TABLE emp  
    ADD PRIMARY KEY (empno);
```

An ALTER TABLE statement that defines and attempts to enable an integrity constraint may fail because rows of the table may violate the integrity constraint. In this case, the statement is rolled back and the constraint definition is not stored and not enabled. Refer to the section “Exception Reporting” on page 6 – 23 for more information about rows that violate integrity constraints.

Disabling Constraints

The following CREATE TABLE and ALTER TABLE statements both define and disable integrity constraints:

```
CREATE TABLE emp (  
    empno NUMBER(5) PRIMARY KEY DISABLE,    . . . );  
  
ALTER TABLE emp  
    ADD PRIMARY KEY (empno) DISABLE;
```

An ALTER TABLE statement that defines and disables an integrity constraints never fails. The definition of the constraint is always allowed because its rule is not enforced.

Enabling and Disabling Defined Integrity Constraints

Use the ALTER TABLE command to

- enable a disabled constraint, using the ENABLE clause
- disable an enabled constraint, using the DISABLE clause

Enabling Disabled Constraints

The following statements are examples of statements that enable disabled integrity constraints:

```
ALTER TABLE dept
    ENABLE CONSTRAINT dname_ukey;
```

```
ALTER TABLE dept
    ENABLE PRIMARY KEY,
    ENABLE UNIQUE (dname, loc);
```

An ALTER TABLE statement that attempts to enable an integrity constraint fails when the rows of the table violate the integrity constraint. In this case, the statement is rolled back and the constraint is not enabled. Refer to the section “Exception Reporting” on page 6 – 23 for more information about rows that violate integrity constraints.

Disabling Enabled Constraints

The following statements are examples of statements that disable enabled integrity constraints:

```
ALTER TABLE dept
    DISABLE CONSTRAINT dname_ukey;
```

```
ALTER TABLE dept
    DISABLE PRIMARY KEY,
    DISABLE UNIQUE (dname, loc);
```

Tip: Using the Data Dictionary for Reference

The example statements in the previous sections require that you have some information about a constraint to enable or disable it. For example, the first statement of each section requires that you know the constraint’s name, while the second statement of each section requires that you know the unique key’s column list. If you do not have such information, you can query one of the data dictionary views defined for constraints; for more information about these views, see “Listing Integrity Constraint Definitions” on page 6 – 28 and the *Oracle7 Server Reference* manual.

Enabling and Disabling Key Integrity Constraints

When enabling or disabling UNIQUE key, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. For more information about enabling, disabling, and managing FOREIGN KEY constraints, see “Managing FOREIGN KEY Integrity Constraints” on page 6 – 25. UNIQUE key and PRIMARY KEY constraints are usually managed by the database administrator; see the *Oracle7 Server Administrator's Guide* for more information.

Enabling Constraints after a Parallel Direct Path Load

SQL*Loader permits multiple concurrent sessions to perform a direct path load into the same table. Because each SQL*Loader session can attempt to re-enable constraints on a table after a direct path load, there is a danger that one session may attempt to re-enable a constraint before another session is finished loading data. In this case, the first session to complete the load will be unable to enable the constraint because the remaining sessions possess share locks on the table.

Because there is a danger that some constraints might not be re-enabled after a direct path load, you should check the status of the constraint after completing the load to ensure that it was enabled properly.

PRIMARY and UNIQUE KEY constraints

PRIMARY KEY and UNIQUE key constraints create indexes on a table when they are enabled, and subsequently can take a significantly long time to enable after a direct path loading session if the table is very large.

You should consider enabling these constraints manually after a load (and not specify the automatic enable feature). This allows you to manually create the required indexes in parallel to save time before enabling the constraint. See the *Oracle7 Server Tuning* manual for more information about creating indexes in parallel.

Exception Reporting

If no exceptions are present when you issue a CREATE TABLE . . . ENABLE . . . or ALTER TABLE . . . ENABLE . . . statement, the integrity constraint is enabled and all subsequent DML statements are subject to the enabled integrity constraints.

If exceptions exist when you enable a constraint, an error is returned and the integrity constraint remains disabled. When a statement is not successfully executed because integrity constraint exceptions exist, the statement is rolled back. If exceptions exist, you cannot enable the constraint until all exceptions to the constraint are either updated or deleted.

To determine which rows violate the integrity constraint, include the EXCEPTIONS option in the ENABLE clause of a CREATE TABLE or ALTER TABLE statement. The EXCEPTIONS option places the ROWID, table owner, table name, and constraint name of all exception rows into a specified table. For example, the following statement attempts to enable the primary key of the DEPT table; if exceptions exist, information is inserted into a table named EXCEPTIONS:

```
ALTER TABLE dept ENABLE PRIMARY KEY EXCEPTIONS INTO exceptions;
```

Create an appropriate exceptions report table to accept information from the EXCEPTIONS option of the ENABLE clause. Create an exception table by submitting the script UTLEXCPT.SQL. The script creates a table named EXCEPTIONS. You can create additional exceptions tables with different names by modifying and resubmitting the script.

If duplicate primary key values exist in the DEPT table and the name of the PRIMARY KEY constraint on DEPT is SYS_C00301, the following rows might be placed in the table EXCEPTIONS by the previous statement:

```
SELECT * FROM exceptions;
```

ROWID	OWNER	TABLE_NAME	CONSTRAINT
-----	-----	-----	-----
000003A5.000C.0001	SCOTT	DEPT	SYS_C00301
000003A5.000D.0001	SCOTT	DEPT	SYS_C00301

A more informative query would be to join the rows in an exception report table and the master table to list the actual rows that violate a specific constraint. For example:

```
SELECT deptno, dname, loc FROM dept, exceptions
WHERE exceptions.constraint = 'SYS_C00301'
AND dept.rowid = exceptions.row_id;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
10	RESEARCH	DALLAS

Rows that violate a constraint must be either updated or deleted from the table that contains the constraint. If updating exceptions, you must change the value that violates the constraint to a value consistent with the constraint or a null (if allowed). After updating or deleting a row in the master table, delete the corresponding rows for the exception in the exception report table to avoid confusion with later exception reports. The statements that update the master table and the exception report table should be in the same transaction to ensure transaction consistency.

For example, to correct the exceptions in the previous examples, the following transaction might be issued:

```
UPDATE dept SET deptno = 20 WHERE dname = 'RESEARCH';

DELETE FROM exceptions WHERE constraint = 'SYS_C00301';

COMMIT;
```

When you manage exceptions, your goal should be to eliminate all exceptions in your exception report table. After eliminating all exceptions, you must re-enable the constraint; the constraint is not automatically enabled after the exceptions are handled.

While you are correcting current exceptions for a table with the constraint disabled, other users can issue statements creating new exceptions.

Altering Integrity Constraints

You cannot alter integrity constraints. If you must alter the action defined by a given integrity constraint, drop the existing constraint and create a replacement.

Dropping Integrity Constraints

Drop an integrity constraint if the rule that it enforces is no longer true or if the constraint is no longer needed. Drop an integrity constraint using the ALTER TABLE command and the DROP clause. For example, the following statements drop integrity constraints:

```
ALTER TABLE dept
  DROP UNIQUE (dname, loc);

ALTER TABLE emp
  DROP PRIMARY KEY,
  DROP CONSTRAINT dept_fkey;

DROP TABLE emp CASCADE CONSTRAINTS;
```

When dropping UNIQUE key, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. For more information about dropping FOREIGN KEY constraints, see “Managing FOREIGN KEY Integrity Constraints” on page 6 – 25. UNIQUE key and PRIMARY KEY constraints are usually managed by the database administrator; see the *Oracle7 Server Administrator's Guide* for more information.

Managing FOREIGN KEY Integrity Constraints

General information about defining, enabling, disabling, and dropping all types of integrity constraints is given in the previous sections. The following section supplements this information, focusing specifically on issues regarding FOREIGN KEY integrity constraints.

Defining FOREIGN KEY Integrity Constraints

The following topics are of interest when defining FOREIGN KEY integrity constraints.

Matching of Datatypes

When defining referential integrity constraints, the corresponding column names of the dependent and referenced tables do not need to match. However, they must be of the same datatype.

Composite Foreign Keys

Because foreign keys reference primary and unique keys of the parent table, and PRIMARY KEY and UNIQUE key constraints are enforced using indexes, composite foreign keys are limited to 16 columns.

Implied Referencing of a Primary Key

If the column list is not included in the REFERENCES option when defining a FOREIGN KEY constraint (single column or composite), Oracle assumes that you intend to reference the primary key of the specified table. Alternatively, you can explicitly specify the column(s) to reference in the parent table within parentheses. Oracle automatically checks to verify that this column list references a primary or unique key of the parent table. If it does not, an informative error is returned.

Privileges Required for FOREIGN KEY Integrity Constraints

To create a FOREIGN KEY constraint, the creator of the constraint must have privileged access to both the parent and the child table.

- **The Parent Table** The creator of the referential integrity constraint must own the parent table or have REFERENCES object privileges on the columns that constitute the parent key of the parent table.
- **The Child Table** The creator of the referential integrity constraint must have the ability to create tables (that is, the CREATE TABLE or CREATE ANY TABLE system privilege) or the ability to alter the child table (that is, the ALTER object privilege for the child table or the ALTER ANY TABLE system privilege).

In both cases, necessary privileges **cannot** be obtained via a role; they must be explicitly granted to the creator of the constraint.

These restrictions allow

- the owner of the child table to explicitly decide what constraints are enforced on her or his tables and the other users that can create constraints on her or his tables
- the owner of the parent table to explicitly decide if foreign keys can depend on the primary and unique keys in her tables

Specifying Referential Actions for Foreign Keys

Oracle allows two different types of referential integrity actions to be enforced, as specified with the definition of a FOREIGN KEY constraint:

- **The UPDATE/DELETE No Action Restriction** This action prevents the update or deletion of a parent key if there is a row in the child table that references the key. By default, all FOREIGN KEY constraints enforce the no action restriction; no option needs to be specified when defining the constraint to enforce the no action restriction. For example:

```
CREATE TABLE emp (  
    . . . ,  
    FOREIGN KEY (deptno) REFERENCES dept);
```

- **The ON DELETE CASCADE Action** This action allows referenced data in the parent key to be deleted (but not updated). If referenced data in the parent key is deleted, all rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint. For example:

```
CREATE TABLE emp (  
    . . . ,  
    FOREIGN KEY (deptno) REFERENCES dept  
        ON DELETE CASCADE);
```

Enabling FOREIGN KEY Integrity Constraints

FOREIGN KEY integrity constraints cannot be enabled if the referenced primary or unique key's constraint is not present or not enabled.

Listing Integrity Constraint Definitions

The data dictionary contains the following views that relate to integrity constraints:

- ALL_CONSTRAINTS
- ALL_CONS_COLUMNS
- CONSTRAINT_COLUMNS
- CONSTRAINT_DEFS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- USER_CROSS_REFS
- DBA_CONSTRAINTS
- DBA_CONS_COLUMNS
- DBA_CROSS_REFS

Refer to the *Oracle7 Server Reference* manual for detailed information about each view.

Examples

Consider the following CREATE TABLE statements that define a number of integrity constraints, and the following examples:

```
CREATE TABLE dept (  
    deptno    NUMBER(3) PRIMARY KEY,  
    dname     VARCHAR2(15),      loc     VARCHAR2(15),  
    CONSTRAINT dname_ukey UNIQUE (dname, loc),  
    CONSTRAINT loc_check1  
        CHECK (loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));  
  
CREATE TABLE emp (  
    empno     NUMBER(5) PRIMARY KEY,  
    ename     VARCHAR2(15) NOT NULL,  
    job       VARCHAR2(10),  
    mgr       NUMBER(5) CONSTRAINT mgr_fkey  
        REFERENCES emp ON DELETE CASCADE,  
    hiredate  DATE,  
    sal       NUMBER(7,2),  
    comm      NUMBER(5,2),  
    deptno    NUMBER(3) NOT NULL  
    CONSTRAINT dept_fkey REFERENCES dept);
```

Example 1
Listing All of Your
Accessible Constraints

The following query lists all constraints defined on all tables accessible to you, the user:

```
SELECT constraint_name, constraint_type, table_name,
       r_constraint_name
FROM user_constraints;
```

Considering the example statements at the beginning of this section, a list similar to the one below is returned:

CONSTRAINT_NAME	C	TABLE_NAME	R_CONSTRAINT_NAME
SYS_C00275	P	DEPT	
DNAME_UKEY	U	DEPT	
LOC_CHECK1	C	DEPT	
SYS_C00278	C	EMP	
SYS_C00279	C	EMP	
SYS_C00280	P	EMP	
MGR_FKEY	R	EMP	SYS_C00280
DEPT_FKEY	R	EMP	SYS_C00275

Notice the following:

- Some constraint names are user specified (such as DNAME_UKEY), while others are system specified (such as SYS_C00275).
- Each constraint type is denoted with a different character in the CONSTRAINT_TYPE column. The table below summarizes the characters used for each constraint type.

<i>Constraint Type</i>	<i>Character</i>
PRIMARY KEY	P
UNIQUE KEY	U
FOREIGN KEY	R
CHECK, NOT NULL	C

Note: An additional constraint type is indicated by the character “V” in the CONSTRAINT_TYPE column. This constraint type corresponds to constraints created by the WITH CHECK OPTION for views. See Chapter 4 for more information about views and the WITH CHECK OPTION.

Example 2
Distinguishing NOT
NULL Constraints
from CHECK
Constraints

In the previous example, several constraints are listed with a constraint type of “C”. To distinguish which constraints are NOT NULL constraints and which are CHECK constraints in the EMP and DEPT tables, issue the following query:

```
SELECT constraint_name, search_condition
FROM user_constraints
WHERE (table_name = 'DEPT' OR table_name = 'EMP') AND
      constraint_type = 'C';
```

Considering the example CREATE TABLE statements at the beginning of this section, a list similar to the one below is returned:

CONSTRAINT_NAME	SEARCH_CONDITION
-----	-----
LOC_CHECK1	loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')
SYS_C00278	ENAME IS NOT NULL
SYS_C00279	DEPTNO IS NOT NULL

Notice the following:

- NOT NULL constraints are clearly identified in the SEARCH_CONDITION column.
- The conditions for user-defined CHECK constraints are explicitly listed in the SEARCH_CONDITION column.

Example 3
Listing the Column
Names that Constitute
an Integrity Constraint

The following query lists all columns that constitute the constraints defined on all tables accessible to you, the user:

```
SELECT constraint_name, table_name, column_name
FROM user_cons_columns;
```

Considering the example statements at the beginning of this section, a list similar to the one below is returned:

CONSTRAINT_NAME	TABLE_NAME	COLUMN_NAME
-----	-----	-----
DEPT_FKEY	EMP	DEPTNO
DNAME_UKEY	DEPT	DNAME
DNAME_UKEY	DEPT	LOC
LOC_CHECK1	DEPT	LOC
MGR_FKEY	EMP	MGR
SYS_C00275	DEPT	DEPTNO
SYS_C00278	EMP	ENAME
SYS_C00279	EMP	DEPTNO
SYS_C00280	EMP	EMPNO

Using Procedures and Packages

This chapter discusses the procedural capabilities of Oracle. This chapter includes information on the following topics:

- PL/SQL procedures and functions
- PL/SQL packages
- creating stored procedures and packages
- invoking stored procedures
- debugging stored procedures
- modifying packages and procedures
- cursor variables
- managing remote dependencies
- calling stored functions from SQL statements
- describing stored procedures

Note: If you are using Trusted Oracle, also see the *Trusted Oracle7 Server Administrator's Guide* for additional information.

PL/SQL

PL/SQL is a modern, block-structured programming language. It provides you with a number of features that make developing powerful database applications very convenient. For example, PL/SQL provides procedural constructs, such as loops and conditional statements, that you do not find in standard SQL.

You can directly issue SQL data manipulation language (DML) statements inside PL/SQL blocks, and you can use procedures, supplied by Oracle, to perform data definition language (DDL) statements.

PL/SQL code executes on the server, so using PL/SQL allows you to centralize significant parts of your database applications for increased maintainability and security. It also enables you to achieve a significant reduction of network overhead in client/server applications.

Note: Some Oracle tools, such as Oracle Forms, contain a PL/SQL engine, and can execute PL/SQL locally.

You can even use PL/SQL for some database applications in place of 3GL programs that use embedded SQL or the Oracle Call Interface (OCI).

There are several kinds of PL/SQL program units:

- anonymous PL/SQL blocks
- triggers
- standalone stored procedures and functions
- packages, that can contain stored procedures and functions

For complete information about the PL/SQL language, see the *PL/SQL User's Guide and Reference*.

Anonymous Blocks

An anonymous PL/SQL block consists of an optional *declarative* part, an *executable* part, and one or more optional *exception handlers*.

You use the declarative part to declare PL/SQL variables, exceptions, and cursors. The executable part contains PL/SQL code and SQL statements, and can contain nested blocks. Exception handlers contain code that is called when the exception is raised, either as a predefined PL/SQL exception (such as NO_DATA_FOUND or ZERO_DIVIDE), or as an exception that you define.

The following short example of a PL/SQL anonymous block prints the names of all employees in department 20 in the EMP table, using the DBMS_OUTPUT package (described on page 8 – 21):

```
DECLARE
    emp_name    VARCHAR2(10);
    CURSOR      c1 IS SELECT ename FROM emp
                  WHERE deptno = 20;
BEGIN
    LOOP
        FETCH c1 INTO emp_name;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(emp_name);
    END LOOP;
END;
```

Note: If you try this block out using SQL*Plus make sure to issue the command SET SERVEROUTPUT ON so that output using the DBMS_OUTPUT procedures such as PUT_LINE is activated. Also, terminate the example with a slash (/) to activate it.

Exceptions allow you to handle Oracle error conditions within PL/SQL program logic. This allows your application to prevent the server from issuing an error that could cause the client application to abort. The following anonymous block handles the predefined Oracle exception NO_DATA_FOUND (which would result in an ORA-01403 error if not handled):

```
DECLARE
    emp_number   INTEGER := 9999;
    emp_name     VARCHAR2(10);
BEGIN
    SELECT ename INTO emp_name FROM emp
        WHERE empno = emp_number; -- no such number
    DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No such employee: ' || emp_number);
END;
```

You can also define your own exceptions, declare them in the declaration part of a block, and define them in the exception part of the block. An example follows:

```
DECLARE
    emp_name          VARCHAR2(10);
    emp_number         INTEGER;
    empno_out_of_range EXCEPTION;
BEGIN
    emp_number := 10001;
    IF emp_number > 9999 OR emp_number < 1000 THEN
        RAISE empno_out_of_range;
    ELSE
        SELECT ename INTO emp_name FROM emp
            WHERE empno = emp_number;
        DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
    END IF;
EXCEPTION
    WHEN empno_out_of_range THEN
        DBMS_OUTPUT.PUT_LINE('Employee number ' || emp_number ||
            ' is out of range.');
```

END;

See the *PL/SQL User's Guide and Reference* for a complete treatment of exceptions.

Anonymous blocks are most often used either interactively, from a tool such as SQL*Plus, or in a precompiler, OCI, or SQL*Module application. They are normally used to call stored procedures, or to open cursor variables. (See page 7 – 26 for a description of cursor variables.)

Database Triggers

A database trigger is a special kind of PL/SQL anonymous block. You can define triggers to fire before or after SQL statements, either on a statement level or for each row that is affected. See Chapter 9 in this Guide for information about database triggers.

Stored Procedures and Functions

A stored procedure or function is a PL/SQL program unit that

- has a name
- can take parameters, and return values
- is stored in the data dictionary
- can be invoked by many users

Note: The term stored *procedure* is sometimes used generically in this Guide to cover both stored procedures and stored functions.

Procedure Names

Since a procedure is stored in the database, it must be named, to distinguish it from other stored procedures, and to make it possible for applications to call it. Each publicly-visible procedure in a schema must have a unique name. The name must be a legal PL/SQL identifier.

Note: If you plan to call a stored procedure using a stub generated by SQL*Module, the stored procedure name must also be a legal identifier in the calling host 3GL language such as Ada or C.

Procedure and function names that are part of packages can be overloaded. That is, you can use the same name for different subprograms as long as their formal parameters differ in number, order, or datatype family. See the *PL/SQL User's Guide and Reference* for more information about subprogram name overloading.

Procedure Parameters

Stored procedures and functions can take parameters. The following example shows a stored procedure that is similar to the anonymous block on page 7 – 3:

```
PROCEDURE get_emp_names (dept_num IN NUMBER) IS
    emp_name      VARCHAR2(10);
    CURSOR        c1 (deptno NUMBER) IS
        SELECT ename FROM emp
           WHERE deptno = deptno;
```



```
BEGIN
  OPEN c1(dept_num);
  LOOP
    FETCH c1 INTO emp_name;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_name);
  END LOOP;
  CLOSE c1;
END;
```

In the stored procedure example, the department number is an input parameter, which is used when the parameterized cursor C1 is opened.

The formal parameters of a procedure have three major parts:

name	The name of the parameter, which must be a legal PL/SQL identifier.
mode	The parameter mode, which indicates whether the parameter is an input-only parameter (IN), an output-only parameter (OUT), or is both an input and an output parameter (IN OUT). If the mode is not specified, IN is assumed.
datatype	The parameter datatype is a standard PL/SQL datatype.

Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes, IN (the default), OUT, and IN OUT, can be used with any subprogram. However, avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take zero or more arguments and return a single value. It is poor programming practice to have a function return multiple values. Also, functions should be free from *side effects*, which change the values of variables not local to the subprogram.

Table 7 – 1 summarizes the information about parameter modes. Parameter modes are explained in detail in the *PL/SQL User's Guide and Reference*.

IN	OUT	IN OUT
the default	must be specified	must be specified
passes values to a subprogram	returns values to the caller	passes initial values to a subprogram; returns updated values to the caller
formal parameter acts like a constant	formal parameter acts like an uninitialized variable	formal parameter acts like an initialized variable
formal parameter cannot be assigned a value	formal parameter cannot be used in an expression; must be assigned a value	formal parameter should be assigned a value
actual parameter can be a constant, initialized variable, literal, or expression	actual parameter must be a variable	actual parameter must be a variable

Table 7 – 1 Parameter Modes

Parameter Datatypes

The datatype of a formal parameter consists of one of the following:

- an *unconstrained* type name, such as NUMBER or VARCHAR2
- a type that is constrained using the %TYPE or %ROWTYPE attributes



Attention: Numerically constrained types such as NUMBER(2) or VARCHAR2(20) are not allowed in a parameter list.

%TYPE and %ROWTYPE Attributes

However, you can use the type attributes %TYPE and %ROWTYPE to constrain the parameter. For example, the GET_EMP_NAMES procedure specification on page 7 – 5 could be written as

```
PROCEDURE get_emp_names(dept_num IN emp.deptno%TYPE)
```

to have the DEPT_NUM parameter take the same datatype as the DEPTNO column in the EMP table. The column and table must be available when a declaration using %TYPE (or %ROWTYPE) is elaborated.

Using %TYPE is recommended, since if the type of the column in the table changes, it is not necessary to change the application code.

If the GET_EMP_NAMES procedure is part of a package, then you can use previously-declared public (package) variables to constrain a parameter datatype. For example:

```
dept_number    number(2);
...
PROCEDURE get_emp_names(dept_num IN dept_number%TYPE);
```

You use the %ROWTYPE attribute to create a record that contains all the columns of the specified table. The following example defines the GET_EMP_REC procedure, that returns all the columns of the EMP table in a PL/SQL record, for the given EMPNO:

```
PROCEDURE get_emp_rec (emp_number IN emp.empno%TYPE,
                      emp_ret     OUT emp%ROWTYPE) IS
BEGIN
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
    INTO emp_ret
    FROM emp
    WHERE empno = emp_number;
END;
```

You could call this procedure from a PL/SQL block as follows:

```
DECLARE
    emp_row      emp%ROWTYPE;      -- declare a record matching a
                                   -- row in the EMP table
BEGIN
    get_emp_rec(7499, emp_row);    -- call for emp# 7499
    DBMS_OUTPUT.PUT(emp_row.ename || ' ' || emp_row.empno);
    DBMS_OUTPUT.PUT(' ' || emp_row.job || ' ' || emp_row.mgr);
    DBMS_OUTPUT.PUT(' ' || emp_row.hiredate || ' ' || emp_row.sal);
    DBMS_OUTPUT.PUT(' ' || emp_row.comm || ' ' || emp_row.deptno);
    DBMS_OUTPUT.NEW_LINE;
END;
```

Stored functions can also return values that are declared using %ROWTYPE. For example:

```
FUNCTION get_emp_rec (dept_num IN emp.deptno%TYPE)
RETURN emp%ROWTYPE IS ...
```

Tables and Records

You can pass PL/SQL tables as parameters to stored procedures and functions. You can also pass tables of records as parameters.

Default Parameter Values

Parameters can take default values. You use the **DEFAULT** keyword or the assignment operator to give a parameter a default value. For example, the specification for the **GET_EMP_NAMES** procedure on page 7 – 5 could be written as

```
PROCEDURE get_emp_names (dept_num IN NUMBER DEFAULT 20) IS ...
```

or as

```
PROCEDURE get_emp_names (dept_num IN NUMBER := 20) IS ...
```

When a parameter takes a default value, it can be omitted from the actual parameter list when you call the procedure. When you do specify the parameter value on the call, it overrides the default value.

DECLARE Keyword

Unlike in an anonymous PL/SQL block, you do not use the keyword **DECLARE** before the declarations of variables, cursors, and exceptions in a stored procedure. In fact, it is an error to use it.

Creating Stored Procedures and Functions

Use your normal text editor to write the procedure. At the beginning of the procedure, place the command

```
CREATE PROCEDURE procedure_name AS ...
```

For example, to use the example on page 7 – 8, you can create a text (source) file called *get_emp.sql* containing the following code:

```
CREATE PROCEDURE get_emp_rec (emp_number IN emp.empno%TYPE,  
                             emp_ret OUT emp%ROWTYPE) AS  
BEGIN  
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno  
    INTO emp_ret  
    FROM emp  
    WHERE empno = emp_number;  
END;
```

Then, using an interactive tool such as SQL*Plus, load the text file containing the procedure by entering the command

```
SQLPLUS> @get_emp
```

to load the procedure into the current schema. (.SQL is the default file extension.) Note the slash (/) at the end of the code. This is not part of the code; it just activates the loading of the procedure.

Note: When developing a new procedure, it is usually much more convenient to use the CREATE OR REPLACE . . . PROCEDURE command. This replaces any previous version of that procedure in the same schema with the newer version. ***This is done with no warning.***

You can use either the keyword IS or AS after the procedure parameter list.

Use the CREATE [OR REPLACE] FUNCTION . . . command to store functions. See the *Oracle7 Server SQL Reference* for the complete syntax of the CREATE PROCEDURE and CREATE FUNCTION commands.

Privileges Required to Create Procedures and Functions

To create a stand-alone procedure or function, or package specification or body, you must meet the following prerequisites:

- You must have the CREATE PROCEDURE system privilege to create a procedure or package in your schema, or the CREATE ANY PROCEDURE system privilege to create a procedure or package in another user's schema.



Attention: To create without errors, that is, to compile the procedure or package successfully, requires the following additional privileges:

The **owner** of the procedure or package must have been explicitly granted the necessary object privileges for all objects referenced within the body of the code; **the owner cannot have obtained required privileges through roles.**

If the privileges of a procedure's or package's owner change, the procedure must be reauthenticated before it is executed. If a necessary privilege to a referenced object is revoked from the owner of the procedure (or package), the procedure cannot be executed.

The EXECUTE privilege on a procedure gives a user the right to execute a procedure owned by another user. Privileged users execute the procedure under the security domain of the procedure's owner. Therefore, users never have to be granted the privileges to the objects referenced by a procedure. This allows for more disciplined and efficient security strategies with database applications and their users. Furthermore, all procedures and packages are stored in the data dictionary (in the SYSTEM tablespace). No quota controls the amount of space available to a user who creates procedures and packages.

Altering Stored Procedures and Functions

To alter a stored procedure or stored function, you must first DROP it, using the DROP PROCEDURE or DROP FUNCTION command, then recreate it using the CREATE PROCEDURE or CREATE FUNCTION command. Alternatively, use the CREATE OR REPLACE PROCEDURE or CREATE OR REPLACE FUNCTION command, which first drops the procedure or function if it exists, then recreates it as specified.

The procedure or function is dropped ***with no warning***.

Packages

A *package* is a group of PL/SQL types, objects, and stored procedures and functions. The *specification* part of a package *declares* the public types, variables, constants, and subprograms that are visible outside the immediate scope of the package. The *body* of a package *defines* the objects declared in the specification, as well as private objects that are not visible to applications outside the package.

The following example shows a package specification for a package named EMPLOYEE_MANAGEMENT. The package contains one stored function and two stored procedures.

```
CREATE PACKAGE employee_management AS
    FUNCTION hire_emp (name VARCHAR2, job VARCHAR2,
        mgr NUMBER, hiredate DATE, sal NUMBER, comm NUMBER,
        deptno NUMBER) RETURN NUMBER;
    PROCEDURE fire_emp (emp_id NUMBER);
    PROCEDURE sal_raise (emp_id NUMBER, sal_incr NUMBER);
END employee_management;
```

The body for this package defines the function and the procedures:

```
CREATE PACKAGE BODY employee_management AS
    FUNCTION hire_emp (name VARCHAR2, job VARCHAR2,
        mgr NUMBER, hiredate DATE, sal NUMBER, comm NUMBER,
        deptno NUMBER) RETURN NUMBER IS

-- The function accepts all arguments for the fields in
-- the employee table except for the employee number.
-- A value for this field is supplied by a sequence.
-- The function returns the sequence number generated
-- by the call to this function.

        new_empno    NUMBER(10);
```

```

BEGIN
    SELECT emp_sequence.NEXTVAL INTO new_empno FROM dual;
    INSERT INTO emp VALUES (new_empno, name, job, mgr,
        hiredate, sal, comm, deptno);
    RETURN (new_empno);
END hire_emp;

PROCEDURE fire_emp(emp_id IN NUMBER) AS

-- The procedure deletes the employee with an employee
-- number that corresponds to the argument EMP_ID. If
-- no employee is found, an exception is raised.

BEGIN
    DELETE FROM emp WHERE empno = emp_id;
    IF SQL%NOTFOUND THEN
        raise_application_error(-20011, 'Invalid Employee
            Number: ' || TO_CHAR(emp_id));
    END IF;
END fire_emp;

PROCEDURE sal_raise (emp_id IN NUMBER, sal_incr IN NUMBER) AS

-- The procedure accepts two arguments. EMP_ID is a
-- number that corresponds to an employee number.
-- SAL_INCR is the amount by which to increase the
-- employee's salary.
BEGIN

-- If employee exists, update salary with increase.
UPDATE emp
    SET sal = sal + sal_incr
    WHERE empno = emp_id;
IF SQL%NOTFOUND THEN
    raise_application_error(-20011, 'Invalid Employee
        Number: ' || TO_CHAR(emp_id));
END IF;
END sal_raise;
END employee_management;

```

Note: If you want to try this example, first create the sequence number EMP_SEQUENCE. You can do this using the following SQL*Plus statement:

```

SQL> EXECUTE CREATE SEQUENCE emp_sequence
> START WITH 8000 INCREMENT BY 10;

```

Creating Packages

Each part of a package is created with a different command. Create the package specification using the CREATE PACKAGE command. The CREATE PACKAGE command declares public package objects.

To create a package body, use the CREATE PACKAGE BODY command. The CREATE PACKAGE BODY command defines the procedural code of the public procedures and functions declared in the package specification. (You can also define private (or local) package procedures, functions, and variables within the package body. See “Local Objects” on page 7 – 14.

The OR REPLACE Clause

It is often more convenient to add the OR REPLACE clause in the CREATE PACKAGE or CREATE PACKAGE BODY commands when you are first developing your application. The effect of this option is to drop the package or the package body without warning. The CREATE commands would then be

```
CREATE OR REPLACE PACKAGE package_name AS ...
```

and

```
CREATE OR REPLACE PACKAGE BODY package_name AS ...
```

Privileges Required to Create Packages

The privileges required to create a package specification or package body are the same as those required to create a stand-alone procedure or function; see page 7 – 10.

Creating Packaged Objects

The body of a package can contain

- procedures declared in the package specification
- functions declared in the package specification
- definitions of cursors declared in the package specification
- local procedures and functions, not declared in the package specification
- local variables

Procedures, functions, cursors, and variables that are declared in the package specification are *global*. They can be called, or used, by external users that have execute permission for the package, or that have EXECUTE ANY PROCEDURE privileges.

When you create the package body, make sure that each procedure that you define in the body has the same parameters, *by name, datatype, and mode*, as the declaration in the package specification. For functions in the package body, the parameters as well as the return type must agree in name and type.

Local Objects

You can define local variables, procedures, and functions in a package body. These objects can only be accessed by other procedures and functions in the body of the same package. They are not visible to external users, regardless of the privileges they hold.

Naming Packages and Package Objects

The names of a package and all public objects in the package must be unique within a given schema. The package specification and its body must have the same name. All package constructs must have unique names within the scope of the package, unless overloading of procedure names is desired.

Dropping Packages and Procedures

A standalone procedure, a standalone function, a package body, or an entire package can be dropped using the SQL commands DROP PROCEDURE, DROP FUNCTION, DROP PACKAGE BODY, and DROP PACKAGE, respectively. A DROP PACKAGE statement drops both a package's specification and body.

The following statement drops the OLD_SAL_RAISE procedure in your schema:

```
DROP PROCEDURE old_sal_raise;
```

Privileges Required to Drop Procedures and Packages

To drop a procedure or package, the procedure or package must be in your schema or you must have the DROP ANY PROCEDURE privilege. An individual procedure within a package cannot be dropped; the containing package specification and body must be re-created without the procedures to be dropped.

Package Invalidations and Session State

Each session that references a package object has its own instance of the corresponding package, including persistent state for any public and private variables, cursors, and constants. If any of the session's instantiated packages (specification or body) are subsequently invalidated and recompiled, all other dependent package instantiations (including state) for the session are lost.

For example, assume that session S instantiates packages P1 and P2, and that a procedure in package P1 calls a procedure in package P2. If P1 is invalidated and recompiled (for example, as the result of a DDL operation), the session S instantiations of both P1 and P2 are lost. In such situations, a session receives the following error the first time it attempts to use any object of an invalidated package instantiation:

```
ORA-04068: existing state of packages has been discarded
```

The second time a session makes such a package call, the package is reinstantiated for the session without error.

Note: Oracle has been optimized to not return this message to the session calling the package that it invalidated. Thus, in the example above, session S would receive this message the first time it called package P2, but would not receive it when calling P1.

In most production environments, DDL operations that can cause invalidations are usually performed during inactive working hours; therefore, this situation might not be a problem for end-user applications. However, if package specification or body invalidations are common in your system during working hours, you might want to code your applications to detect for this error when package calls are made. For example, the user-side application might reinitialize any user-side state that depends on any session's package state (that was lost) and reissue the package call.

Timestamps and Signatures

In Oracle7 release 7.2 and earlier, dependencies among PL/SQL library units (packages, stored procedures, and stored functions) were handled in a very consistent, but restrictive, manner. Each time that a library unit or a relevant database object was altered, all dependent units were marked as invalid. Invalid dependent library units then had to be recompiled before they could be executed.

Timestamps

In the release 7.2 dependency model, each library unit carries a timestamp. The timestamp is set by the server when the unit is created or recompiled. Figure 7 – 1 demonstrates this graphically. Procedures P1 and P2 call stored procedure P3. Stored procedure P3 references table T1. In this example, each of the procedures is dependent on table T1. P3 depends upon T1 directly, while P1 and P2 depend upon T1 indirectly.

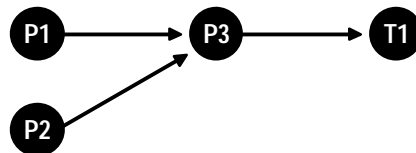


Figure 7 – 1 Dependency Relationships

If P3 is altered, P1 and P2 are marked as invalid immediately if they are on the same server as P3. The compiled states of P1 and P2 contain records of the timestamp of P3. So if the procedure P3 is altered and recompiled, the timestamp on P3 no longer matches the value that was recorded for P3 during the compilation of P1 and P2.

If P1 and P2 are on a client system, or on another Oracle server in a distributed environment, the timestamp information is used to mark them as invalid at runtime.

Disadvantages of the Timestamp Model

The disadvantage of this dependency model is that is unnecessarily restrictive. Recompilation of dependent objects across the network are often performed when not strictly necessary, leading to performance degradation.

Furthermore, on the client side, the timestamp model can lead to situations that block an application from running at all, if the client-side application is built using PL/SQL version 2. (Earlier releases of tools such as Oracle Forms that used PL/SQL version 1 on the client side did not use this dependency model, since PL/SQL version 1 had no support for stored procedures.)

For releases of Oracle Forms that are integrated with PL/SQL version 2 on the client side, the timestamp model can present problems. First of all, during the installation of the application, the application is rendered invalid unless the client-side PL/SQL procedures that it uses are recompiled at the client site. Also, if a client-side procedure depends on a server procedure, and the server procedure is changed or automatically recompiled, the client-side PL/SQL procedure must then be recompiled. Yet in many application environments (such as Forms runtime applications), there is no PL/SQL compiler available on the client. This blocks the application from running at all. The client application developer must then redistribute new versions of the application to all customers.

Signatures

To alleviate some of the problems with the timestamp-only dependency model, Oracle7 release 7.3 (with PL/SQL release 2.3) introduces the additional capability of remote dependencies using *signatures*. The signature capability affects only remote dependencies. Local (same server) dependencies are not affected, as recompilation is always possible in this environment.

The signature of a subprogram contains information about the

- name of the subprogram
- base types of the parameters of the subprogram
- modes of the parameters (IN, OUT, IN OUT)

Note: Only the types and modes of parameters are significant. The name of the parameter does not affect the signature.

The user has control over whether signatures or timestamps govern remote dependencies. See the section “Controlling Remote Dependencies” on page 7 – 23 for more information. If the signature dependency model is in effect, a dependency on a remote library unit causes an invalidation of the dependent unit if the dependent unit contains a call to a subprogram in the parent unit, and the signature of this subprogram has been changed in an incompatible manner.

For example, consider a procedure GET_EMP_NAME stored on a server BOSTON_SERVER. The procedure is defined as

```
CREATE OR REPLACE PROCEDURE get_emp_name (  
    emp_number    IN NUMBER,  
    hire_date     OUT VARCHAR2,  
    emp_name      OUT VARCHAR2) AS  
  
BEGIN  
    SELECT ename, to_char(hiredate, 'DD-MON-YY')  
    INTO emp_name, hire_date  
    FROM emp  
    WHERE empno = emp_number;  
  
END;
```

When GET_EMP_NAME is compiled on the BOSTON_SERVER, its signature as well as its timestamp is recorded.

Now assume that on another server, in California, some PL/SQL code calls GET_EMP_NAME identifying it using a DB link called BOSTON_SERVER, as follows:

```
CREATE OR REPLACE PROCEDURE print_ename (  
    emp_number IN NUMBER) AS  
    hire_date   VARCHAR2(12);  
    ename       VARCHAR2(10);  
  
BEGIN  
    get_emp_name@BOSTON_SERVER(  
        emp_number, hire_date, ename);  
    dbms_output.put_line(ename);  
    dbms_output.put_line(hiredate);  
  
END;
```

When this California server code is compiled, the following actions take place:

- a connection is made to the Boston server
- the signature of GET_EMP_NAME is transferred to the California server
- the signature is recorded in the compiled state of PRINT_ENAME

At runtime, during the remote procedure call from the California server to the Boston server, the recorded signature of GET_EMP_NAME that was saved in the compiled state of PRINT_ENAME gets sent across to the Boston server., regardless of whether there were any changes or not.

If the timestamp dependency mode is in effect, a mismatch in timestamps causes an error status to be returned to the calling procedure.

However, if the signature mode is in effect, any mismatch in timestamps is ignored, and the recorded signature of GET_EMP_NAME in the compiled state of PRINT_ENAME on the California server is compared with the current signature of GET_EMP_NAME on the Boston server. If they match, the call succeeds. If they do not match, an error status is returned to the PRINT_NAME procedure.

Note that the GET_EMP_NAME procedure on the Boston server could have been changed. Or, its timestamp could be different from that recorded in the PRINT_NAME procedure on the California server, due to, for example, the installation of a new release of the server. As long as the signature remote dependency mode is in effect on the California server, a timestamp mismatch does not cause an error when GET_EMP_NAME is called.

What is a Signature?

A signature is associated with each compiled stored library unit. It identifies the unit using the following criteria:

- the name of the unit, that is, the package, procedure or function name
- the types of each of the parameters of the subprogram
- the modes of the parameters
- the number of parameters
- the type of the return value for a function

When Does a Signature Change?

Datatypes:

A signature changes when you change from one class of datatype to another. Within each datatype class, there can be several types. Changing a parameter datatype from one type to another within a class does not cause the signature to change. Table 7 – 2 shows the classes of types.

Varchar Types: VARCHAR2 VARCHAR STRING LONG ROWID	Number Types: NUMBER INTEGER INT SMALLINT DECIMAL DEC REAL FLOAT NUMERIC DOUBLE PRECISION
Character Types: CHARACTER CHAR	
Raw Types: RAW LONG RAW	
Integer Types: BINARY_INTEGER PLS_INTEGER BOOLEAN NATURAL POSITIVE POSITIVEN NATURALN	Date Type: DATE MLS Label Type: MLSLABEL

Table 7 – 2 Datatype Classes

Modes:

Changing to or from an explicit specification of the default parameter mode IN does not change the signature of a subprogram. For example, changing

```
PROCEDURE P1 (param1 NUMBER);
```

to

```
PROCEDURE P1 (param1 IN NUMBER);
```

does not change the signature. Any other change of parameter mode *does* change the signature.

Default Parameter Values:

Changing the specification of a default parameter value does not change the signature. For example, procedure P1 has the same signature in the following two examples:

```
PROCEDURE P1 (param1 IN NUMBER := 100);  
PROCEDURE P1 (param1 IN NUMBER := 200);
```

An application developer who requires that callers get the new default value must recompile the called procedure, but no signature-based invalidation occurs when a default parameter value assignment is changed.

Examples

In the GET_EMP_NAME procedure defined on page 7 – 18, if the procedure body is changed to

```
BEGIN  
-- date format model changes  
    SELECT ename, to_char(hiredate, 'DD/MON/YYYY')  
        INTO emp_name, hire_date  
        FROM emp  
        WHERE empno = emp_number;  
END;
```

then the specification of the procedure has not changed, and so its signature has not changed.

But if the procedure specification is changed to

```
CREATE OR REPLACE PROCEDURE get_emp_name (  
    emp_number  IN NUMBER,  
    hire_date   OUT DATE,  
    emp_name    OUT VARCHAR2) AS
```

and the body is changed accordingly, then the signature changes, because the parameter HIRE_DATE has a different datatype.

However, if the name of that parameter changes to WHEN_HIRED, and the datatype remains VARCHAR2, and the mode remains OUT, then the signature does not change. Changing the *name* of a formal parameter does not change the signature of the unit.

Consider the following example:

```
CREATE OR REPLACE PACKAGE emp_package AS
    TYPE emp_data_type IS RECORD (
        emp_number NUMBER,
        hire_date   VARCHAR2(12),
        emp_name    VARCHAR2(10));
    PROCEDURE get_emp_data
        (emp_data IN OUT emp_data_type);
END;

CREATE OR REPLACE PACKAGE BODY emp_package AS
    PROCEDURE get_emp_data
        (emp_data IN OUT emp_data_type) IS
    BEGIN
        SELECT empno, ename, to_char(hiredate, 'DD/MON/YY')
            INTO emp_data
            FROM emp
            WHERE empno = emp_data.emp_number;
    END;
```

If the package specification is changed so that the record's field names are changed, but the types remain the same, this does not affect the signature. For example, the following package specification has the same signature as the previous package specification example:

```
CREATE OR REPLACE PACKAGE emp_package AS
    TYPE emp_data_type IS RECORD (
        emp_num    NUMBER,           -- was emp_number
        hire_dat   VARCHAR2(12),    --was hire_date
        empname    VARCHAR2(10)); -- was emp_name
    PROCEDURE get_emp_data
        (emp_data IN OUT emp_data_type);
END;
```

Changing the name of the type of a parameter does not cause a change in the signature if the type remains the same as before. For example, the following package specification for EMP_PACKAGE is the same as the first one on page 7 – 22:

```

CREATE OR REPLACE PACKAGE emp_package AS
    TYPE emp_data_record_type IS RECORD (
        emp_number NUMBER,
        hire_date   VARCHAR2(12),
        emp_name    VARCHAR2(10));
    PROCEDURE get_emp_data
        (emp_data IN OUT emp_data_record_type);
END;

```

Controlling Remote Dependencies

Whether the timestamp or the signature dependency model is in effect is controlled by the dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE`.

If the initialization parameter file contains the specification

```
REMOTE_DEPENDENCIES_MODE = TIMESTAMP
```

and this is not explicitly overridden dynamically, then only timestamps are used to resolve dependencies. This is identical to the Oracle7 Server release 7.2 model.

If the initialization parameter file contains the parameter specification

```
REMOTE_DEPENDENCIES_MODE = SIGNATURE
```

and this not explicitly overridden dynamically, then signatures are used to resolve dependencies.

You can alter the mode dynamically by using the DDL commands

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =
    {SIGNATURE | TIMESTAMP}
```

to alter the dependency model for the current session, or

```
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =
    {SIGNATURE | TIMESTAMP}
```

to alter the dependency model on a system-wide basis after startup.

If the `REMOTE_DEPENDENCIES_MODE` parameter is not specified, either in the `INIT.ORA` parameter file, or using the `ALTER SESSION` or `ALTER SYSTEM` DDL commands, `TIMESTAMP` is the default value. So, unless you explicitly use the `REMOTE_DEPENDENCIES_MODE` parameter, or the appropriate DDL command, your server is operating using the release 7.2 timestamp dependency model.

When you use `REMOTE_DEPENDENCIES_MODE=SIGNATURE` you should be aware of the following:

- If you change the default value of a parameter of a remote procedure, the local procedure calling the remote procedure is not invalidated. If the call to the remote procedure does not supply the parameter, the default value is used. In this case, because invalidation/recompilation does not automatically occur, the old default value is used. If you wish to see the new default values, you must recompile the calling procedure manually.
- If you add a new overloaded procedure in a package (a new procedure with the same name as an existing one) , local procedures that call the remote procedure are not invalidated. If it turns out that this overloading ought to result in a rebinding of existing calls from the local procedure under the `TIMESTAMP` mode, this rebinding does not happen under the `SIGNATURE` mode, because the local procedure does not get invalidated. You must recompile the local procedure manually to achieve the new rebinding.
- If the types of parameters of an existing packaged procedure are changed so that the new types have the same shape as the old ones, the local calling procedure is not invalidated/recompiled automatically. You must recompile the calling procedure manually to get the semantics of the new type.

Dependency Resolution

When `REMOTE_DEPENDENCIES_MODE = TIMESTAMP` (the default value), dependencies among library units are handled exactly like in Oracle7 release 7.2 or earlier. If at runtime the timestamp of a called remote procedure does not match the timestamp of the called procedure, the calling (dependent) unit is invalidated, and must be recompiled. In this case, if there is no local PL/SQL compiler, the calling application cannot proceed.

In the timestamp dependency mode, signatures are not compared. If there is a local PL/SQL compiler, recompilation happens automatically when the calling procedure is executed.

When `REMOTE_DEPENDENCIES_MODE = SIGNATURE`, the recorded timestamp in the calling unit is first compared to the current timestamp in the called remote unit. If they match, then the call proceeds normally. If the timestamps do not match, then the signature of the called remote subprogram, as recorded in the calling subprogram, is compared with the current signature of the called subprogram. If they do not match, using the criteria described in the section “What is a Signature” on page 7 – 19, then an error is returned to the calling session.

Suggestions for Managing Dependencies

Oracle recommends that you follow these guidelines for setting the `REMOTE_DEPENDENCIES_MODE` parameter:

- Server-side PL/SQL users can set the parameter to `TIMESTAMP` (or let it default to that) to get 7.2 behavior.
- Server-side PL/SQL users can choose to use the signature dependency mode if they have a distributed system and wish to avoid possible unnecessary recompilations.
- Client-side PL/SQL users should set the parameter to `SIGNATURE`. This allows
 - installation of new applications at client sites, without the need to recompile procedures
 - ability to upgrade the server, without encountering timestamp mismatches.
- When using `SIGNATURE` mode on the server side, make sure to add new procedures to the end of the procedure (or function) declarations in a package spec. Adding a new procedure in the middle of the list of declarations can cause unnecessary invalidation and recompilation of dependent procedures.

Cursor Variables

Cursor variables are references to cursors. A cursor is a static object; a cursor variable is a pointer to a cursor. Since cursor variables are pointers, they can be passed and returned as parameters to procedures and functions. A cursor variable can also refer to (“point to”) different cursors in its lifetime.

Some additional advantages of cursor variables are

- *Encapsulation*: queries are centralized in the stored procedure that opens the cursor variable.
- *Ease of maintenance*: if you need to change the cursor, you only need to make the change in one place: the stored procedure. There is no need to change each application.
- *Convenient security*: the user of the application is the username used when the application connects to the server. The user must have execute permission on the stored procedure that opens the cursor. But the user does not need to have read permission on the tables used in the query. This capability can be used to limit access to the columns in the table, as well as access to other stored procedures.

See the *PL/SQL User's Guide and Reference* for a complete discussion of cursor variables.

Declaring and Opening Cursor Variables

You normally allocate memory for a cursor variable in the client application, using the appropriate ALLOCATE command. In Pro*C, you use the EXEC SQL ALLOCATE <cursor_name> command. In the OCI, you use the Cursor Data Area.

Starting with Oracle7 release 7.3, you can use cursor variables in applications that run entirely in a single server session. You can declare cursor variables in PL/SQL subprograms, open them, and use them as parameters for other PL/SQL subprograms.

Examples

This section includes several examples of cursor variable usage in PL/SQL. For additional cursor variable examples that use the programmatic interfaces, see the following manuals:

- *Programmer's Guide to the Oracle Pro*C/C++ Precompiler*
- *Programmer's Guide to the Oracle Precompilers*
- *Programmer's Guide to the Oracle Call Interface*
- *SQL*Module User's Guide and Reference*

Fetching Data

The following package defines a PL/SQL cursor variable type EMP_VAL_CV_TYPE, and two procedures. The first procedure opens the cursor variable, using a bind variable in the WHERE clause. The second procedure (FETCH_EMP_DATA) fetches rows from the EMP table using the cursor variable.

```
CREATE OR REPLACE PACKAGE emp_data AS

    TYPE emp_val_cv_type IS REF CURSOR RETURN emp%ROWTYPE;

    PROCEDURE open_emp_cv (emp_cv          IN OUT emp_val_cv_type,
                          dept_number      IN INTEGER);
    PROCEDURE fetch_emp_data (emp_cv       IN emp_val_cv_type,
                             emp_row       OUT emp%ROWTYPE);

END emp_data;

CREATE OR REPLACE PACKAGE BODY emp_data AS

    PROCEDURE open_emp_cv (emp_cv          IN OUT emp_val_cv_type,
                          dept_number      IN INTEGER) IS
    BEGIN
        OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = dept_number;
    END open_emp_cv;

    PROCEDURE fetch_emp_data (emp_cv       IN emp_val_cv_type,
                             emp_row       OUT emp%ROWTYPE) IS
    BEGIN
        FETCH emp_cv INTO emp_row;
    END fetch_emp_data;

END emp_data;
```

The following example shows how you can call the EMP_DATA package procedures from a PL/SQL block:

```
DECLARE
-- declare a cursor variable
emp_curs emp_data.emp_val_cv_type;

dept_number dept.deptno%TYPE;
emp_row emp%ROWTYPE;

BEGIN
    dept_number := 20;

-- open the cursor using a variable
emp_data.open_emp_cv(emp_curs, dept_number);

-- fetch the data and display it
LOOP
    emp_data.fetch_emp_data(emp_curs, emp_row);
    EXIT WHEN emp_curs%NOTFOUND;
    DBMS_OUTPUT.PUT(emp_row.ename || ' ');
    DBMS_OUTPUT.PUT_LINE(emp_row.sal);
END LOOP;
END;
```

Implementing Variant Records

The power of cursor variables comes from their ability to point to different cursors. In the following package example, a discriminant is used to open a cursor variable to point to one of two different cursors:

```
CREATE OR REPLACE PACKAGE emp_dept_data AS

    TYPE cv_type IS REF CURSOR;

    PROCEDURE open_cv (cv          IN OUT cv_type,
                      discrim     IN      POSITIVE);

END emp_dept_data;
/

CREATE OR REPLACE PACKAGE BODY emp_dept_data AS

    PROCEDURE open_cv (cv          IN OUT cv_type,
                      discrim IN      POSITIVE) IS

    BEGIN
        IF discrim = 1 THEN
            OPEN cv FOR SELECT * FROM emp WHERE sal > 2000;
```

```

        ELSIF discrim = 2 THEN
            OPEN cv FOR SELECT * FROM dept;
        END IF;
    END open_cv;

END emp_dept_data;

```

You can call the OPEN_CV procedure to open the cursor variable and point it to either a query on the EMP table or on the DEPT table. How would you use this? The following PL/SQL block shows that you can fetch using the cursor variable, then use the ROWTYPE_MISMATCH predefined exception to handle either fetch:

```

DECLARE
    emp_rec  emp%ROWTYPE;
    dept_rec dept%ROWTYPE;
    cv       emp_dept_data.cv_type;

BEGIN
    emp_dept_data.open_cv(cv, 1); -- open CV for EMP fetch
    FETCH cv INTO dept_rec;       -- but fetch into DEPT record
                                -- which raises ROWTYPE_MISMATCH

    DBMS_OUTPUT.PUT(dept_rec.deptno);
    DBMS_OUTPUT.PUT_LINE('  ' || dept_rec.loc);

EXCEPTION
    WHEN ROWTYPE_MISMATCH THEN
        BEGIN
            DBMS_OUTPUT.PUT_LINE
                ('Row type mismatch, fetching EMP data...');
            FETCH cv into emp_rec;
            DBMS_OUTPUT.PUT(emp_rec.deptno);
            DBMS_OUTPUT.PUT_LINE('  ' || emp_rec.ename);
        END;
END;

```

Hiding PL/SQL Code

You can deliver your stored procedures in object code format using the PL/SQL Wrapper. Wrapping your PL/SQL code hides your application internals. To run the PL/SQL Wrapper, enter the WRAP command at your system prompt using the following syntax:

```
WRAP INAME=input_file [ONAME=output_file]
```

For complete instructions on using the PL/SQL Wrapper, see the *PL/SQL User's Guide and Reference*.

Error Handling

Oracle allows user-defined errors in PL/SQL code to be handled so that user-specified error numbers and messages are returned to the client application. Once received, the client application can handle the error based on the user-specified error number and message returned by Oracle.

User-specified error messages are returned using the `RAISE_APPLICATION_ERROR` procedure:

```
RAISE_APPLICATION_ERROR(error_number, 'text', keep_error_stack)
```

This procedure terminates procedure execution, rolls back any effects of the procedure, and returns a user-specified error number and message (unless the error is trapped by an exception handler). `ERROR_NUMBER` must be in the range of -20000 to -20999. Error number -20000 should be used as a generic number for messages where it is important to relay information to the user, but having a unique error number is not required. `TEXT` must be a character expression, 2 Kbytes or less (longer messages are ignored). `KEEP_ERROR_STACK` can be `TRUE`, if you want to add the error to any already on the stack, or `FALSE`, if you want to replace the existing errors. By default, this option is `FALSE`.



Attention: Some of the Oracle-supplied packages, such as `DBMS_OUTPUT`, `DBMS_DESCRIBE`, and `DBMS_ALERT`, use application error numbers in the range -20000 to -20005. See the descriptions of these packages for more information.

The `RAISE_APPLICATION_ERROR` procedure is often used in exception handlers or in the logic of PL/SQL code. For example, the following exception handler selects the string for the associated user-defined error message and calls the `RAISE_APPLICATION_ERROR` procedure:

```
...
WHEN NO_DATA_FOUND THEN
    SELECT error_string INTO message
    FROM error_table,
    V$NLS_PARAMETERS V
    WHERE error_number = -20101 AND LANG = v.value AND
          v.name = "NLS_LANGUAGE";
    raise_application_error(-20101, message);
...
```

Several examples earlier in this chapter also demonstrate the use of the `RAISE_APPLICATION_ERROR` procedure. The next section has an example of passing a user-specified error number from a trigger to a procedure. For information on exception handling when calling remote procedures, see page 7 – 33.

Declaring Exceptions and Exception Handling Routines

User-defined exceptions are explicitly defined and signaled within the PL/SQL block to control processing of errors specific to the application. When an exception is *raised* (signaled), the normal execution of the PL/SQL block stops and a routine called an exception handler is invoked. Specific exception handlers can be written to handle any internal or user-defined exception.

Application code can check for a condition that requires special attention using an IF statement. If there is an error condition, two options are available:

- Issue a `RAISE` statement that names the appropriate exception. A `RAISE` statement stops the execution of the procedure and control passes to an exception handler (if any).
- Call the `RAISE_APPLICATION_ERROR` procedure to return a user-specified error number and message.

You can also define an exception handler to handle user-specified error messages. For example, Figure 7 – 2 illustrates

- an exception and associated exception handler in a procedure
- a conditional statement that checks for an error (such as transferring funds not available) and issues a user-specified error number and message within a trigger
- how user-specified error numbers are returned to the calling environment (in this case, a procedure) and how that application can define an exception that corresponds to the user-specified error number

Declare a user-defined exception in a procedure or package body (private exceptions) or in the specification of a package (public exceptions). *Define* an exception handler in the body of a procedure (standalone or package).

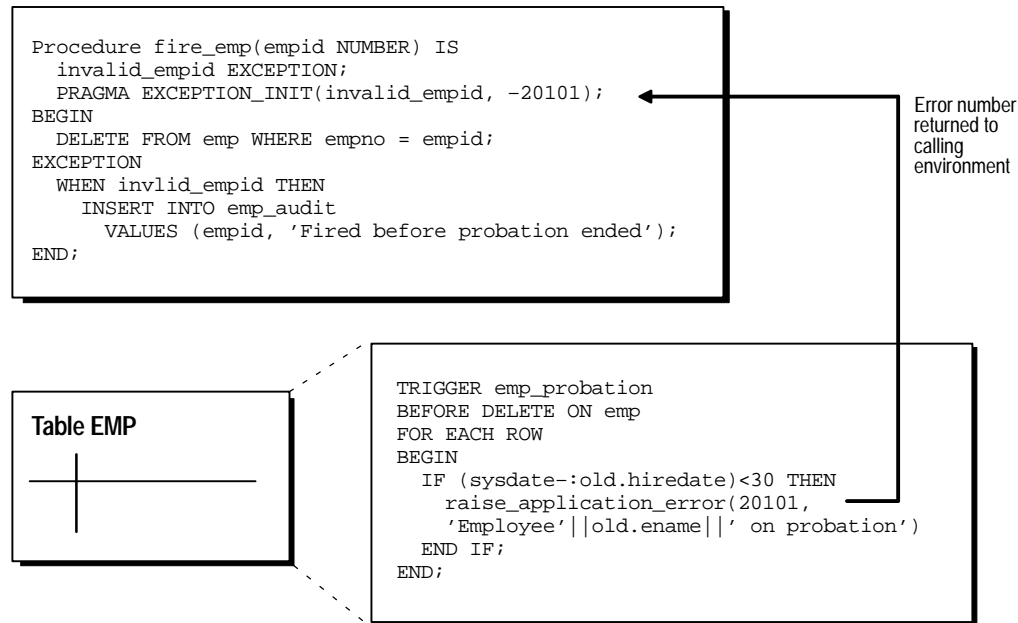


Figure 7 – 2 Exceptions and User-Defined Errors

Unhandled Exceptions

In database PL/SQL program units, an unhandled user-error condition or internal error condition that is not trapped by an appropriate exception handler causes the implicit rollback of the program unit. If the program unit includes a COMMIT statement before the point at which the unhandled exception is observed, the implicit rollback of the program unit can only be completed back to the previous commit.

Additionally, unhandled exceptions in database-stored PL/SQL program units propagate back to client-side applications that call the containing program unit. In such an application, only the application program unit call is rolled back (not the entire application program unit) because it is submitted to the database as a SQL statement.

If unhandled exceptions in database PL/SQL program units are propagated back to database applications, the database PL/SQL code should be modified to handle the exceptions. Your application can also trap for unhandled exceptions when calling database program units and handle such errors appropriately. For more information, see “Handling Errors in Remote Procedures.”

Handling Errors in Distributed Queries

You can use a trigger or stored procedure to create a distributed query. This distributed query is decomposed by the local Oracle into a corresponding number of remote queries, which are sent to the remote nodes for execution. The remote nodes execute the queries and send the results back to the local node. The local node then performs any necessary post-processing and returns the results to the user or application.

If a portion of a distributed statement fails, for example, due to an integrity constraint violation, Oracle returns error number ORA-02055. Subsequent statements or procedure calls return error number ORA-02067 until a rollback or rollback to savepoint is issued.

You should design your application to check for any returned error messages that indicate that a portion of the distributed update has failed. If you detect a failure, you should rollback the entire transaction (or rollback to a savepoint) before allowing the application to proceed.

Handling Errors in Remote Procedures

When a procedure is executed locally or at a remote location, four types of exceptions can occur:

- PL/SQL user-defined exceptions, which must be declared using the keyword `EXCEPTION`
- PL/SQL predefined exceptions, such as `NO_DATA_FOUND`
- SQL errors, such as ORA-00900 and ORA-02015
- Application exceptions, which are generated using the `RAISE_APPLICATION_ERROR()` procedure

When using local procedures, all of these messages can be trapped by writing an exception handler, such as shown in the following example:

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    /* ...handle the exception */
```

Notice that the WHEN clause requires an exception name. If the exception that is raised does not have a name, such as those generated with RAISE_APPLICATION_ERROR, one can be assigned using PRAGMA_EXCEPTION_INIT, as shown in the following example:

```
DECLARE
    ...
    null_salary EXCEPTION;
    PRAGMA EXCEPTION_INIT(null_salary, -20101);
BEGIN
    ...
    RAISE_APPLICATION_ERROR(-20101, 'salary is missing');
    ...
EXCEPTION
    WHEN null_salary THEN
        ...
```

When calling a remote procedure, exceptions are also handled by creating a local exception handler. The remote procedure must return an error number to the local, calling procedure, which then handles the exception as shown in the previous example. Because PL/SQL user-defined exceptions always return ORA-06510 to the local procedure, these exceptions cannot be handled. All other remote exceptions can be handled in the same manner as local exceptions.

Compile Time Errors

When you use SQL*Plus to submit PL/SQL code, and the code contains errors, you receive notification that compilation errors have occurred, but no immediate indication of what the errors are. For example, if you submit a standalone (or stored) procedure PROC1 in the file *proc1.sql* as follows:

```
SVRMGR> @proc1
```

and there are one or more errors in the code, you receive a notice such as

```
MGR-00072: Warning: Procedure PROC1 created with compilation
errors
```

In this case, use the SHOW ERRORS command in SQL*Plus to get a list of the errors that were found. SHOW ERRORS with no argument lists the errors from the most recent compilation. You can qualify SHOW ERRORS using the name of a procedure, function, package, or package body:

```
SQL> SHOW ERRORS PROC1
SQL> SHOW ERRORS PROCEDURE PROC1
```

See the *SQL*Plus User's Guide and Reference* for complete information about the SHOW ERRORS command.



Attention: Before issuing the SHOW ERRORS command, use the SET CHARWIDTH command to get long lines on output. For example:

```
SET CHARWIDTH 132
```

is usually a good choice.

For example, assume you want to create a simple procedure that deletes records from the employee table using SQL*Plus:

```
CREATE PROCEDURE fire_emp(emp_id NUMBER) AS
BEGIN
    DELETE FROM emp WHERE empno = emp_id;
END
/
```

Notice that the CREATE PROCEDURE statement has two errors: the DELETE statement has an error (the 'E' is absent from WHERE) and the semicolon is missing after END.

After the CREATE PROCEDURE statement is issued and an error is returned, a SHOW ERRORS statement would return the following lines:

```
SHOW ERRORS;
```

```
ERRORS FOR PROCEDURE FIRE_EMP:
LINE/COL      ERROR
-----
3/24   PL/SQL-00103: Encountered the symbol "EMPNO" wh. . .
5/0    PL/SQL-00103: Encountered the symbol "END" when . . .
2 rows selected.
```

Notice that each line and column number where errors were found is listed by the SHOW ERRORS command.

Alternatively, you can query the following data dictionary views to list errors when using any tool or application:

- USER_ERRORS
- ALL_ERRORS
- DBA_ERRORS

The error text associated with the compilation of a procedure is updated when the procedure is replaced, and deleted when the procedure is dropped.

Original source code can be retrieved from the data dictionary using the following views: ALL_SOURCE, USER_SOURCE, and DBA_SOURCE. See the *Oracle7 Server Reference* manual for more information about these data dictionary views.

Debugging

You can debug stored procedures and triggers using the DBMS_OUTPUT supplied package. You put PUT and PUT_LINE statements in your code to output the value of variables and expressions to your terminal. See the section “Output from Stored Procedures and Triggers” on page 8 – 21 for more information about the DBMS_OUTPUT package.

A more convenient way to debug, if your platform supports it, is to use the Oracle Procedure Builder, which is part of the Oracle Developer/2000 tool set. Procedure Builder lets you execute PL/SQL procedures and triggers in a controlled debugging environment, and you can set breakpoints, list the values of variables, and perform other debugging tasks. See the *Oracle Procedure Builder Developer's Guide* for more information.

Invoking Stored Procedures

Procedures can be invoked from many different environments. For example:

- A procedure can be called within the body of another procedure or a trigger.
- A procedure can be interactively called by a user using an Oracle tool (such as SQL*Plus)
- A procedure can be explicitly called within an application (such as a SQL*Forms or precompiler application).
- A stored function can be called from a SQL statement in a manner similar to calling a built-in SQL function, such as LENGTH or ROUND.

Some common examples of invoking procedures from within these environments follow. Calling stored functions from SQL is described on page 7 – 43.

A Procedure or Trigger Calling Another Procedure

A procedure or trigger can call another stored procedure. For example, included in the body of one procedure might be the line

```
. . .  
sal_raise(emp_id, 200);  
. . .
```

This line calls the SAL_RAISE procedure. EMP_ID is a variable within the context of the procedure. Note that recursive procedure calls are allowed within PL/SQL; that is, a procedure can call itself.

Interactively Invoking Procedures From Oracle Tools

A procedure can be invoked interactively from an Oracle tool such as SQL*Plus. For example, to invoke a procedure named SAL_RAISE, owned by you, you can use an anonymous PL/SQL block, as follows:

```
BEGIN  
    sal_raise(1043, 200);  
END;
```

Note: Interactive tools such as SQL*Plus require that you follow these lines with a slash (/) to execute the PL/SQL block.

An easier way to execute a block is to use the SQL*Plus command EXECUTE, which effectively wraps BEGIN and END statements around the code you enter. For example:

```
EXECUTE sal_raise(1043, 200);
```

Some interactive tools allow session variables to be created. For example, when using SQL*Plus, the following statement creates a session variable:

```
VARIABLE assigned_empno NUMBER
```

Once defined, any session variable can be used for the duration of the session. For example, you might execute a function and capture the return value using a session variable:

```
EXECUTE :assigned_empno := hire_emp('JSMITH', 'President', \  
    1032, SYSDATE, 5000, NULL, 10);  
PRINT assigned_empno;  
ASSIGNED_EMPNO  
-----  
                2893
```


See the *SQL*Plus User's Guide and Reference* for SQL*Plus information. See your tools manual for information about performing similar operations using your development tool.

Calling Procedures within 3GL Applications

A 3GL database application such as a precompiler or OCI application can include a call to a procedure within the code of the application.

To execute a procedure within a PL/SQL block in an application, simply call the procedure. The following line within a PL/SQL block calls the FIRE_EMP procedure:

```
fire_emp(:empno);
```

In this case, :EMPNO is a host (bind) variable within the context of the application.

To execute a procedure within the code of a precompiler application, you must use the EXEC call interface. For example, the following statement calls the FIRE_EMP procedure in the code of a precompiler application:

```
EXEC SQL EXECUTE
  BEGIN
    fire_emp(:empno);
  END;
END-EXEC;
```

:EMPNO is a host (bind) variable.

For more information about calling PL/SQL procedures from within 3GL applications, see the following manuals:

- *Programmer's Guide to the Oracle Call Interface*
- *Programmer's Guide to the Oracle Pro*C/C++ Precompiler*, for Pro*C/C++ release 2.2
- For other precompilers, the *Programmer's Guide to the Oracle Precompilers*, plus the relevant language supplement, such as the *Pro*COBOL Supplement to the Oracle Precompilers Guide* or the *Pro*FORTRAN Supplement to the Oracle Precompilers Guide*.
- *SQL*Module User's Guide and Reference*

Name Resolution When Invoking Procedures

References to procedures and packages are resolved according to the algorithm described in the section “Name Resolution in SQL Statements” on page 4 – 48.

Privileges Required to Execute a Procedure

If you are the owner of a standalone procedure or package, you can execute the standalone procedure or packaged procedure, or any public procedure or packaged procedure at any time, as described in the previous sections. If you want to execute a standalone or packaged procedure owned by another user, the following conditions apply:

- You must have the EXECUTE privilege for the standalone procedure or package containing the procedure, or have the EXECUTE ANY PROCEDURE system privilege. If you are executing a remote procedure, you must have been granted the EXECUTE privilege or EXECUTE ANY PROCEDURE system privilege directly, not via a role.
- You must include the owner’s name in the call, as in:

```
EXECUTE jward.fire_emp (1043);
```

```
EXECUTE jward.hire_fire.fire_emp (1043);
```



Attention: A stored subprogram or package executes in the privilege domain of the **owner** of the procedure. The owner must have been **explicitly granted** the necessary object privileges to all objects referenced within the body of the code.

Specifying Values for Procedure Arguments

When you invoke a procedure, specify a value or parameter for each of the procedure’s arguments. Identify the argument values using either of the following methods, or a combination of both:

- List the values in the order the arguments appear in the procedure declaration.
- Specify the argument names and corresponding values, in any order.

For example, these statements each call the procedure `UPDATE_SAL` to increase the salary of employee number 7369 by 500:

```
sal_raise(7369, 500);

sal_raise(sal_incr=>500, emp_id=>7369);

sal_raise(7369, sal_incr=>500);
```

The first statement identifies the argument values by listing them in the order in which they appear in the procedure specification.

The second statement identifies the argument values by name and in an order different from that of the procedure specification. If you use argument names, you can list the arguments in any order.

The third statement identifies the argument values using a combination of these methods. If you use a combination of order and argument names, values identified in order must precede values identified by name.

If you have used the `DEFAULT` option to define default values for `IN` parameters to a subprogram (see the *PL/SQL User's Guide and Reference*), you can pass different numbers of actual parameters to the subprogram, accepting or overriding the default values as you please. If an actual value is not passed, the corresponding default value is used. If you want to assign a value to an argument that occurs after an omitted argument (for which the corresponding default is used), you must explicitly designate the name of the argument, as well as its value.

Invoking Remote Procedures

Invoke remote procedures using an appropriate database link and the procedure's name. The following SQL*Plus statement executes the procedure `FIRE_EMP` located in the database pointed to by the local database link named `NY`:

```
EXECUTE fire_emp@NY(1043);
```

For information on exception handling when calling remote procedures, see page 7 – 34.

Remote Procedure Calls and Parameter Values

You must explicitly pass values to all remote procedure parameters even if there are defaults. You cannot access remote package variables and constants.

Referencing Remote Objects

Remote objects can be referenced within the body of a locally defined procedure. The following procedure deletes a row from the remote employee table:

```
CREATE PROCEDURE fire_emp(emp_id NUMBER) IS
BEGIN
    DELETE FROM emp@sales WHERE empno = emp_id;
END;
```

The list below explains how to properly call remote procedures, depending on the calling environment.

- Remote procedures (standalone and packaged) can be called from within a procedure, OCI application, or precompiler application by specifying the remote procedure name, a database link, and the arguments for the remote procedure.

```
CREATE PROCEDURE local_procedure(arg1, arg2) AS
BEGIN
    ...
    remote_procedure@dblink(arg1, arg2);
    ...
END;
```

- In the previous example, you could create a synonym for REMOTE_PROCEDURE@DBLINK. This would enable you to call the remote procedure from an Oracle tool application, such as a SQL*Forms application, as well from within a procedure, OCI application, or precompiler application.

```
CREATE PROCEDURE local_procedure(arg1, arg2) AS
BEGIN
    ...
    synonym(arg1, arg2);
    ...
END;
```

- If you did not want to use a synonym, you could write a local cover procedure to call the remote procedure.

```
BEGIN local_procedure(arg1, arg2); END;
```

Here, LOCAL_PROCEDURE is defined as in the first item of this list.

Note: Synonyms can be used to create location transparency for the associated remote procedures.



Warning: Unlike stored procedures, which use compile-time binding, when referencing remote procedures, runtime binding is used. The user account to which you connect depends on the database link.

All calls to remotely stored procedures are assumed to perform updates; therefore, this type of referencing always requires two-phase commit of that transaction (even if the remote procedure is read-only). Furthermore, if a transaction that includes a remote procedure call is rolled back, the work done by the remote procedure is also rolled back. A procedure called remotely cannot execute a COMMIT, ROLLBACK, or SAVEPOINT statement.

A *distributed update* modifies data on two or more nodes. A distributed update is possible using a procedure that includes two or more remote updates that access data on different nodes. Statements in the construct are sent to the remote nodes and the execution of the construct succeeds or fails as a unit. If part of a distributed update fails and part succeeds, a rollback (of the entire transaction or to a savepoint) is required to proceed. Consider this when creating procedures that perform distributed updates.

Pay special attention when using a local procedure that calls a remote procedure. If a timestamp mismatch is found during execution of the local procedure, the remote procedure is not executed and the local procedure is invalidated.

Synonyms for Procedures and Packages

Synonyms can be created for standalone procedures and packages to

- hide the identity of the name and owner of a procedure or package
- provide location transparency for remotely stored procedures (standalone or within a package)

When a privileged user needs to invoke a procedure, an associated synonym can be used. Because the procedures defined within a package are not individual objects (that is, the package is the object), synonyms cannot be created for individual procedures within a package.

Calling Stored Functions from SQL Expressions

You can include user-written PL/SQL functions in SQL expressions. (You must be using PL/SQL release 2.1 or greater.) By using PL/SQL functions in SQL statements, you can do the following:

- Increase user productivity by extending SQL. Expressiveness of the SQL statement increases where activities are too complex, too awkward, or unavailable with SQL.
- Increase query efficiency. Functions used in the WHERE clause of a query can filter data using criteria that would otherwise have to be evaluated by the application.
- Manipulate character strings to represent special datatypes (for example, latitude, longitude, or temperature).
- Provide parallel query execution. If the query is parallelized, SQL statements in your PL/SQL function may be executed in parallel also (using the parallel query option).

Using PL/SQL Functions

PL/SQL functions must be created as top-level functions or declared within a package specification before they can be named within a SQL statement. Stored PL/SQL functions are used in the same manner as built-in Oracle functions (such as SUBSTR or ABS). PL/SQL functions can be placed wherever an Oracle function can be placed within a SQL statement; that is, wherever expressions can occur in SQL.

For example, they can be called from the following:

- the select list of the SELECT command
- the condition of the WHERE and HAVING clause
- the CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- the VALUES clause of the INSERT command
- the SET clause of the UPDATE command

However stored PL/SQL functions cannot be called from a CHECK constraint clause of a CREATE or ALTER TABLE command or be used to specify a default value for a column. These situations require an unchanging definition.

Note: Unlike functions, which are called as part of an expression, procedures are called as statements. Therefore, PL/SQL procedures are *not* directly callable from SQL statements. However, functions called from a PL/SQL statement or referenced in a SQL expression can call a PL/SQL procedure.

Syntax

Use the following syntax to reference a PL/SQL function from SQL:

```
[ [schema.]package. ]function_name[ @dblink ] [ (param_1...param_n) ]
```

For example, to reference a function that you have created that is called MY_FUNC, in the MY_FUNCS_PKG package, in the SCOTT schema, and that takes two numeric parameters, you could call it as

```
SELECT scott.my_funcs_pkg.my_func(10,20) from dual
```

Naming Conventions

If only one of the optional schema or package names is given, the first identifier can be either a schema name or a package name. For example, to determine whether PAYROLL in the reference PAYROLL.TAX_RATE is a schema or package name, Oracle proceeds as follows:

- Oracle first checks for the PAYROLL package in the current schema.
- If a PAYROLL package is not found, Oracle looks for a schema named PAYROLL that contains a top-level TAX_RATE function. If the TAX_RATE function is not found in the PAYROLL schema, an error message is returned.
- If the PAYROLL package is found in the current schema, Oracle looks for a TAX_RATE function in the PAYROLL package. If a TAX_RATE function is not found in the PAYROLL package, an error message is returned.

You can also refer to a stored top-level function using any synonym that you have defined for it.

Name Precedence

In SQL statements, the names of database columns take precedence over the names of functions with no parameters. For example, if schema SCOTT creates the following two objects:

```
CREATE TABLE emp(new_sal NUMBER ...);
CREATE FUNCTION new_sal RETURN NUMBER IS ...;
```

Then in the following two statements, the reference to NEW_SAL refers to the column EMP.NEW_SAL:

```
SELECT new_sal FROM emp;
SELECT emp.new_sal FROM emp;
```

To access the function NEW_SAL, you would enter the following:

```
SELECT scott.new_sal FROM emp;
```

Example

For example, to call the TAX_RATE PL/SQL function from schema SCOTT, execute it against the SS_NO and SAL columns in TAX_TABLE, and place the results in the variable INCOME_TAX, specify the following:

```
SELECT scott.tax_rate (ss_no, sal)
       INTO income_tax
       FROM tax_table
       WHERE ss_no = tax_id;
```

Listed below are sample calls to PL/SQL functions that are allowed in SQL expressions.

```
circle_area(radius)
payroll.tax_rate(empno)
scott.payroll.tax_rate(dependents, empno)@ny
```

Arguments

To pass any number of arguments to a function, supply the arguments within the parentheses. You must use positional notation; named notation is not currently supported. For functions that do not accept arguments, omit the parentheses.

The argument's datatypes and the function's return type are limited to those types that are supported by SQL. For example, you cannot call a PL/SQL function that returns a PL/SQL BINARY_INTEGER from a SQL statement.

Using Default Values

The stored function *gross_pay* initializes two of its formal parameters to default values using the `DEFAULT` clause, as follows:

```
CREATE FUNCTION gross_pay
  (emp_id IN NUMBER,
   st_hrs IN NUMBER DEFAULT 40,
   ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
  ...
```

When calling *gross_pay* from a procedural statement, you can always accept the default value of *st_hrs*. That is because you can use named notation, which lets you skip parameters, as in

```
IF gross_pay(eenum,ot_hrs => otime) > pay_limit THEN ...
```

However, when calling *gross_pay* from a SQL expression, you cannot accept the default value of *st_hrs* unless you accept the default value of *ot_hrs*. That is because you cannot use named notation.

Meeting Basic Requirements

To be callable from SQL expressions, a user-defined PL/SQL function must meet the following basic requirements:

- It must be a stored function, *not* a function defined within a PL/SQL block or subprogram.
- It must be a row function, *not* a column (group) function; that is, it cannot take an entire column of data as its argument.
- All its formal parameters must be IN parameters; none can be an OUT or IN OUT parameter.
- The datatypes of its formal parameters must be Oracle Server internal types such as CHAR, DATE, or NUMBER, *not* PL/SQL types such as BOOLEAN, RECORD, or TABLE.
- Its return type (the datatype of its result value) must be an Oracle Server internal type.

For example, the following stored function meets the basic requirements:

```
CREATE FUNCTION gross_pay
  (emp_id IN NUMBER,
   st_hrs IN NUMBER DEFAULT 40,
   ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
  st_rate  NUMBER;
  ot_rate  NUMBER;
```

```

BEGIN
    SELECT srate, orate INTO st_rate, ot_rate FROM payroll
    WHERE acctno = emp_id;
    RETURN st_hrs * st_rate + ot_hrs * ot_rate;
END gross_pay;

```

Controlling Side Effects

To execute a SQL statement that calls a stored function, the Oracle Server must know the *purity level* of the function. That is, the extent to which the function is free of side effects. In this context, *side effects* are references to database tables or packaged variables.

Side effects can prevent the parallelization of a query, yield order-dependent (and therefore indeterminate) results, or require that package state be maintained across user sessions (which is not allowed). Therefore, the following rules apply to stored functions called from SQL expressions:

- The function cannot modify database tables; therefore, it cannot execute an INSERT, UPDATE, or DELETE statement.
- Functions that read or write the values of packaged variables cannot be executed remotely or in parallel.
- Only functions called from a SELECT, VALUES, or SET clause can write the values of packaged variables.
- The function cannot call another subprogram that breaks one of the foregoing rules. Also, the function cannot reference a view that breaks one of the foregoing rules. (Oracle replaces references to a view with a stored SELECT operation, which can include function calls.)

For standalone functions, Oracle can enforce these rules by checking the function body. However, the body of a packaged function is hidden; only its specification is visible. So, for packaged functions, you must use the pragma (compiler directive) `RESTRICT_REFERENCES` to enforce the rules.

The pragma tells the PL/SQL compiler to deny the packaged function read/write access to database tables, packaged variables, or both. If you try to compile a function body that violates the pragma, you get a compilation error.

Calling Packaged Functions

To call a packaged function from SQL expressions, you must assert its purity level by coding the pragma `RESTRICT_REFERENCES` in the package specification (not in the package body). The pragma must follow the function declaration but need not follow it immediately. Only one pragma can reference a given function declaration.

To code the pragma `RESTRICT_REFERENCES`, you use the syntax

```
PRAGMA RESTRICT_REFERENCES (  
    function_name, WNDS [, WNPS] [, RNDS] [, RNPS]);
```

where:

WNDS means “writes no database state” (does not modify database tables)

WNPS means “writes no package state” (does not change the values of packaged variables)

RNDS means “reads no database state” (does not query database tables)

RNPS means “reads no package state” (does not reference the values of packaged variables)

You can pass the arguments in any order, but you *must* pass the argument `WNDS`. No argument implies another. For instance, `RNPS` does not imply `WNPS`.

In the example below, the function *compound* neither reads nor writes database or package state, so you can assert the maximum purity level. Always assert the highest purity level a function allows. That way, the PL/SQL compiler will never reject the function unnecessarily.

```
CREATE PACKAGE finance AS -- package specification  
    ...  
    FUNCTION compound  
        (years IN NUMBER,  
         amount IN NUMBER,  
         rate IN NUMBER) RETURN NUMBER;  
    PRAGMA RESTRICT_REFERENCES (compound, WNDS, WNPS, RNDS, RNPS);  
END finance;  
  
CREATE PACKAGE BODY finance AS --package body  
    ...  
    FUNCTION compound  
        (years IN NUMBER,  
         amount IN NUMBER,  
         rate IN NUMBER) RETURN NUMBER IS
```

```

BEGIN
    RETURN amount * POWER((rate / 100) + 1, years);
END compound;
-- no pragma in package body
END finance;

```

Later, you might call *compound* from a PL/SQL block, as follows:

```

BEGIN
    ...
    SELECT finance.compound(yrs,amt,rte) -- function call
INTO interest          FROM accounts      WHERE acctno = acct_id;

```

Referencing Packages with an Initialization Part

Packages can have an initialization part, which is hidden in the package body. Typically, the initialization part holds statements that initialize public variables. In the following example, the SELECT statement initializes the public variable *prime_rate*:

```

CREATE PACKAGE loans AS
    prime_rate REAL; -- public packaged variable
    ...
END loans;

CREATE PACKAGE BODY loans AS
    ...
BEGIN -- initialization part
    SELECT prime INTO prime_rate FROM rates;
END loans;

```

The initialization code is run only once—the first time the package is referenced. If the code reads or writes database state or package state other than its own, it can cause side effects. Moreover, a stored function that references the package (and thereby runs the initialization code) can cause side effects indirectly. So, to call the function from SQL expressions, you must use the pragma `RESTRICT_REFERENCES` to assert or imply the purity level of the initialization code.

To assert the purity level of the initialization code, you use a variant of the pragma `RESTRICT_REFERENCES`, in which the function name is replaced by a package name. You code the pragma in the package specification, where it is visible to other users. That way, anyone referencing the package can see the restrictions and conform to them.

To code the variant pragma `RESTRICT_REFERENCES`, you use the syntax

```
PRAGMA RESTRICT_REFERENCES (  
    package_name, WNDS [, WNPS] [, RNDS] [, RNPS]);
```

where the arguments `WNDS`, `WNPS`, `RNDS`, and `RNPS` have the usual meaning.

In the example below, the initialization code reads database state and writes package state. However, you can assert `WNPS` because the code is writing the state of its own package, which is permitted. So, you assert `WNDS`, `WNPS`, `RNPS`—the highest purity level the function allows. (If the public variable *prime_rate* were in another package, you could not assert `WNPS`.)

```
CREATE PACKAGE loans AS  
    PRAGMA RESTRICT_REFERENCES (loans, WNDS, WNPS, RNPS);  
    prime_rate REAL;  
    ...  
END loans;  
  
CREATE PACKAGE BODY loans AS  
    ...  
BEGIN  
    SELECT prime INTO prime_rate FROM rates;  
END loans;
```

You can place the pragma anywhere in the package specification, but placing it at the top (where it stands out) is a good idea.

To imply the purity level of the initialization code, your package must have a `RESTRICT_REFERENCES` pragma for one of the functions it declares. From the pragma, Oracle can infer the purity level of the initialization code (because the code cannot break any rule enforced by a pragma). In the next example, the pragma for the function *discount* implies that the purity level of the initialization code is at least `WNDS`:

```
CREATE PACKAGE loans AS  
    ...  
    FUNCTION discount (...) RETURN NUMBER;  
    PRAGMA RESTRICT_REFERENCES (discount, WNDS);  
END loans;  
...
```

To draw an inference, Oracle can combine the assertions of all `RESTRICT_REFERENCES` pragmas. For example, the following pragmas (combined) imply that the purity level of the initialization code is at least `WNDS`, `RNDS`:

```
CREATE PACKAGE loans AS
...
FUNCTION discount (...) RETURN NUMBER;
FUNCTION credit_ok (...) RETURN CHAR;
PRAGMA RESTRICT_REFERENCES (discount, WNDS);
PRAGMA RESTRICT_REFERENCES (credit_ok, RNDS);
END loans;
...
```

Avoiding Problems

To call a packaged function from SQL expressions, you must assert its purity level using the pragma `RESTRICT_REFERENCES`. However, if the package has an initialization part, the PL/SQL compiler might not let you assert the highest purity level the function allows. As a result, you might be unable to call the function remotely, in parallel, or from certain SQL clauses.

This happens when a packaged function is purer than the package initialization code. Remember, the first time a package is referenced, its initialization code is run. If that reference is a function call, any additional side effects caused by the initialization code occur during the call. So, in effect, the initialization code lowers the purity level of the function.

To avoid this problem, move the package initialization code into a subprogram. That way, your application can run the code explicitly (rather than implicitly during package instantiation) without affecting your packaged functions.

A similar problem arises when a packaged function is purer than a subprogram it calls. This lowers the purity level of the function. Therefore, the `RESTRICT_REFERENCES` pragma for the function must specify the lower purity level. Otherwise, the PL/SQL compiler will reject the function. In the following example, the compiler rejects the function because its pragma asserts `RNDS` but the function calls a procedure that reads database state:

```
CREATE PACKAGE finance AS
...
FUNCTION compound (years IN NUMBER,
                  amount IN NUMBER) RETURN NUMBER;
PRAGMA RESTRICT_REFERENCES (compound, WNDS, WNPS, RNDS, RNPS);
END finance;
```

```

CREATE PACKAGE BODY finance AS
...
    FUNCTION compound (years  IN NUMBER,
                        amount IN NUMBER) RETURN NUMBER IS
        rate  NUMBER;
    PROCEDURE calc_loan_rate (loan_rate OUT NUMBER) IS
        prime_rate REAL;
    BEGIN
        SELECT p_rate INTO prime_rate FROM rates;
        ...
    END;
BEGIN
    calc_loan_rate(rate);
    RETURN amount * POWER((rate / 100) + 1, years);
END compound;
END finance;

```

Overloading

PL/SQL lets you *overload* packaged (but not standalone) functions. That is, you can use the same name for different functions if their formal parameters differ in number, order, or datatype family.

However, a `RESTRICT_REFERENCES` pragma can apply to only one function declaration. So, a pragma that references the name of overloaded functions always applies to the nearest foregoing function declaration. In the following example, the pragma applies to the second declaration of *valid*:

```

CREATE PACKAGE tests AS
    FUNCTION valid (x NUMBER) RETURN CHAR;
    FUNCTION valid (x DATE) RETURN CHAR;
    PRAGMA RESTRICT_REFERENCES (valid, WNDS);
...

```

Privileges Required

To call a PL/SQL function from SQL, you must either own or have `EXECUTE` privileges on the function. To select from a view defined with a PL/SQL function, you are required to have `SELECT` privileges on the view. No separate `EXECUTE` privileges are needed to select from the view.

Supplied Packages

Several packaged procedures are provided with Oracle to either allow PL/SQL access to some SQL features, or to extend the functionality of the database. You may need to take advantage of the functionality provided by these packages when creating your application, or you may simply want to use these packages for ideas in creating your own stored procedures. This section lists each of the supplied packages and indicates where they are described in more detail.

These packages run as the invoking user rather than the package owner. The packaged procedures are callable through public synonyms of the same name.

Support for SQL Features

Oracle is supplied with the following packaged procedures, which allow PL/SQL to access some features of SQL. The footnotes at the end of Table 7 – 3 explain any restrictions on the use of each procedure. You should consult the package specifications for the most up-to-date information on these packages.

Package	Procedure(Arguments)	SQL Command Equivalent
DBMS_SESSION	close_database_link(dblink varchar2)	ALTER SESSION CLOSE DATABASE dblink
	reset_package (see note 5)	This procedure reinitializes the state of all packages; there is no SQL equivalent
	set_label(lbl varchar2) (note 4)	ALTER SESSION SET LABEL text
	set_mls_label_format(fmt varchar2) (note 4)	ALTER SESSION SET MLS_LABEL_FORMAT = fmt
	set_nls(param varchar2, value varchar2) (notes 1,4)	ALTER SESSION SET nls_param = nls_param_values
	set_role(role_cmd varchar2) (notes 1, 6)	SET ROLE ...
	set_sql_trace(sql_trace boolean)	ALTER SESSION SET SQL_TRACE = [TRUE FALSE]
	unique_session_id return varchar2	This function returns a unique session ID; there is no SQL equivalent.
	is_role_enabled return boolean	This function is used to determine if a role is enabled; there is no SQL equivalent.
	set_close_cached_open_cursors(close_cursors boolean)	ALTER SESSION SET CLOSE_CACHED_OPEN_CURSORS
	free_unused_user_memory	This procedure lets you reclaim unused memory; there is no SQL equivalent.

Package	Procedure(Arguments)	SQL Command Equivalent
DBMS_DDL	alter_compile(type varchar2, schema varchar2, name varchar2) (notes 1, 2, 3, 4	ALTER PROCEDURE proc COMPILE
		ALTER FUNCTION func COMPILE
		ALTER PACKAGE pack COMPILE
	analyze_object(type varchar2, schema varchar2, name varchar2, method varchar2, estimate_rows number default null, estimate_percent number default null)	ANALYZE INDEX
		ANALYZE TABLE
		ANALYZE CLUSTER
DBMS_TRANSACTION	advise_commit	ALTER SESSION ADVISE COMMIT
	advise_rollback	ALTER SESSION ADVISE ROLLBACK
	advise_nothing	ALTER SESSION ADVISE NOTHING
	commit (notes 1,2,4)	COMMIT
	commit_comment(cmnt varchar2) (notes 1,2,4)	COMMIT COMMENT text
	commit_force(xid varchar2, scn varchar2 default null) (notes 1,2,3,4)	COMMIT FORCE text ...
	read_only (notes 1,3,4)	SET TRANSACTION READ ONLY
	read_write (notes 1,3,4)	SET TRANSACTION READ WRITE
	rollback (notes 1,2,4)	ROLLBACK
	rollback_force(xid varchar2) (notes 1,2,3,4)	ROLLBACK ... FORCE text ...
	rollback_savepoint(svpt varchar2) (notes 1,2,4)	ROLLBACK ... TO SAVEPOINT ...
	savepoint(savept varchar2) (notes 1,2,4)	SAVEPOINT savepoint
	use_rollback_segment(rb_name varchar2) (notes 1,2,4)	SET TRANSACTION USE ROLLBACK SEGMENT segment
	purge_mixed(xid in number)	See <i>Oracle7 Server Distributed Systems, Volume I</i> for more information
	begin_discrete_transaction (notes 1,3,4,5)	See the <i>Oracle7 Server Tuning</i> manual for more information

Table 7 – 3 (continued) Supplied Packages: SQL Features

Package	Procedure(Arguments)	SQL Command Equivalent
	local_transaction_id(create_transaction BOOLEAN default FALSE) return VARCHAR2	See <i>Oracle7 Server Distributed Systems, Volume I</i> for more information
	step_id return number	See <i>Oracle7 Server Distributed Systems, Volume I</i> for more information
DBMS_UTILITY	compile_schema(schema varchar2) (notes 1,2,3,4)	This procedure is equivalent to calling alter_compile on all procedures, functions, and packages accessible by you. Compilation is completed in dependency order.
	analyze_schema(schema varchar2, method varchar2, estimate_rows number default null, estimate_percent number default null)	This procedure is equivalent to calling analyze_object on all objects in the given schema.
	format_error_stack return varchar2	This function formats the error stack into a variable.
	format_call_stack return varchar2	This function formats the current call stack into a variable.
	is_parallel_server return boolean	This function returns TRUE when running in Parallel Server mode.
	get_time return number	This function returns the time in hundredths of a second.
	name_resolve(name in varchar2, context in number, schema out varchar2, part1 out varchar2, part2 out varchar2, dblink out varchar2, part1_type out number, object_number out number)	See <i>Oracle7 Server Distributed Systems, Volume I</i> for more information

Table 7 – 3 (continued) Supplied Packages: SQL Features

1 not allowed in triggers

*2 not allowed in procedures called from SQL*Forms*

3 not allowed in read-only transactions

4 not allowed in remote (coordinated) sessions

5 not allowed in recursive sessions

6 not allowed in stored procedures

For more details on each SQL command equivalent, refer to the *Oracle7 Server SQL Reference* manual. The COMMIT, ROLLBACK, ROLLBACK... TO SAVEPOINT, and SAVEPOINT procedures are directly supported by PL/SQL; they are included in the package for completeness.

Additional Functionality

Several packages are supplied with Oracle to extend the functionality of the database. The cross-reference column in Table 7 – 4 tells you where to look for more information on each of these packages.

Package Name	Description	Cross-reference
DBMS_ALERT	Supports asynchronous notification of database events.	Chapter 12
DBMS_DESCRIBE	Lets you describe the arguments of a stored procedure.	Page 7 – 57
DBMS_JOB	Lets you schedule administrative procedures that you want performed at periodic intervals.	<i>Oracle7 Server Administrator's Guide</i>
DBMS_LOCK	Lets you use the Oracle Lock Management services for your applications.	Page 3 – 19
DBMS_OUTPUT	Lets you output messages from triggers, procedures, and packages.	Page 8 – 21
DBMS_PIPE	Allows sessions in the same instance to communicate with each other.	Chapter 8
DBMS_SHARED_POOL	Lets you keep objects in shared memory, so that they will not be aged out with the normal LRU mechanism.	<i>Oracle7 Server Tuning</i>
DBMS_APPLICATION_INFO	Lets you register an application name with the database for auditing or performance tracking purposes.	<i>Oracle7 Server Tuning</i>
DBMS_SYSTEM	Provides system-level utilities, such as letting you enable SQL trace for a session.	<i>Oracle7 Server Tuning</i>
DBMS_SPACE	Provides segment space information not available through standard views.	<i>Oracle7 Server Administrator's Guide</i>
DBMS_SQL	Lets you write stored procedures and anonymous PL/SQL blocks using dynamic SQL; lets you parse any DML or DDL statement.	Chapter 10
DBMS_REFRESH	Lets you create groups of snapshots that can be refreshed together to a transactionally consistent point in time. Use of this feature requires the distributed option.	<i>Oracle7 Server Distributed Systems, Volume II</i>
DBMS_SNAPSHOT	Lets you refresh one or more snapshots that are not part of the same refresh group, purge snapshot log. Use of this feature requires the distributed option.	<i>Oracle7 Server Distributed Systems, Volume II</i>

Table 7 – 4 Supplied Packages: Additional Functionality

Package Name	Description	Cross-reference
DBMS_DEFER, DMBS_DEFER_SYS, DBMS_DEFER_QUERY	Lets you build and administer deferred remote procedure calls. Use of this feature requires the replication option.	<i>Oracle7 Server Distributed Systems, Volume II</i>
DBMS_REPCAT	Lets you use Oracle's symmetric replication facility. Use of this feature requires the replication option.	<i>Oracle7 Server Distributed Systems, Volume II</i>
DBMS_REPCAT_AUTH, DBMS_REPCAT_ADMIN	Lets you create users with the privileges needed by the symmetric replication facility. Use of this feature requires the replication option.	<i>Oracle7 Server Distributed Systems, Volume II</i>

Table 7 – 4 (continued) Supplied Packages: Additional Functionality

Describing Stored Procedures

You can use the DBMS_DESCRIBE package to get information about a stored procedure or function.

This package provides the same functionality as the Oracle Call Interface ODESSP call. The procedure DESCRIBE_PROCEDURE in this package accepts the name of a stored procedure, and a description of the procedure and each of its parameters. For more information on ODESSP, see the *Programmer's Guide to the Oracle Call Interface*.

DBMS_DESCRIBE Package

To create the DBMS_DESCRIBE package, submit the DBMSDESC.SQL and PRVTDESC.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script. See page 7 – 39 for information on granting the necessary privileges to users who will be executing this package.

Security

This package is available to PUBLIC and performs its own security checking based on the object being described.

Types

The DBMS_DESCRIBE package declares two PL/SQL table types, which are used to hold data returned by DESCRIBE_PROCEDURE in its OUT parameters. The types are

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
    INDEX BY BINARY_INTEGER;
```

```
TYPE NUMBER_TABLE IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
```

Errors

DBMS_DESCRIBE can raise application errors in the range -20000 to -20004. The errors are

```
-20000: ORU 10035: cannot describe a package ('X') only a
        procedure within a package
-20001: ORU-10032: procedure 'X' within package 'Y' does not
        exist
-20002: ORU-10033 object 'X' is remote, cannot describe; expanded
        name 'Y'
-20003: ORU-10036: object 'X' is invalid and cannot be described
-20004: syntax error attempting to parse 'X'
```

DESCRIBE_PROCEDURE Procedure

Syntax

The parameters for DESCRIBE_PROCEDURE are shown in Table 7 – 5.
The syntax is

```
PROCEDURE DESCRIBE_PROCEDURE(  
    object_name      IN  VARCHAR2,  
    reserved1        IN  VARCHAR2,  
    reserved2        IN  VARCHAR2,  
    overload         OUT NUMBER_TABLE,  
    position         OUT NUMBER_TABLE,  
    level            OUT NUMBER_TABLE,  
    argument_name    OUT VARCHAR2_TABLE,  
    datatype         OUT NUMBER_TABLE,  
    default_value    OUT NUMBER_TABLE,  
    in_out           OUT NUMBER_TABLE,  
    length           OUT NUMBER_TABLE,  
    precision        OUT NUMBER_TABLE,  
    scale            OUT NUMBER_TABLE,  
    radix            OUT NUMBER_TABLE,  
    spare            OUT NUMBER_TABLE);
```

Parameter	Mode	Description
object_name	IN	<p>The name of the procedure being described. The syntax for this parameter follows the rules used for identifiers in SQL. The name can be a synonym. This parameter is required and may not be null. The total length of the name cannot exceed 197 bytes. An incorrectly specified OBJECT_NAME can result in one of the following exceptions:</p> <p>ORA–20000 – A package was specified. You can only specify a stored procedure, stored function, packaged procedure, or packaged function.</p> <p>ORA–20001 – The procedure or function that you specified does not exist within the given package.</p> <p>ORA–20002 – The object that you specified is a remote object. This procedure cannot currently describe remote objects.</p> <p>ORA–20003 – The object that you specified is invalid and cannot be described.</p> <p>ORA–20004 – The object was specified with a syntax error.</p>
reserved1 reserved2	IN	Reserved for future use. Must be set to null or the empty string.
overload	OUT	A unique number assigned to the procedure's signature. If a procedure is overloaded, this field holds a different value for each version of the procedure.

Table 7 – 5 DBMS_DESCRIBE.DESCRIBE_PROCEDURE Parameters

Parameter	Mode	Description
position	OUT	Position of the argument in the parameter list. Position 0 returns the values for the return type of a function.
level	OUT	If the argument is a composite type, such as record, this parameter returns the level of the datatype. See the <i>Programmer's Guide to the Oracle Call Interface</i> write-up of the ODESSP call for an example of its use.
argument_name	OUT	The name of the argument associated with the procedure that you are describing.
datatype	OUT	The Oracle datatype of the argument being described. The datatypes and their numeric type codes are: 0 placeholder for procedures with no arguments 1 VARCHAR, VARCHAR2, STRING 2 NUMBER, INTEGER, SMALLINT, REAL, FLOAT, DECIMAL 3 BINARY_INTEGER, PLS_INTEGER, POSITIVE, NATURAL 8 LONG 11 ROWID 12 DATE 23 RAW 24 LONG RAW 96 CHAR (ANSI FIXED CHAR), CHARACTER 106 MLS LABEL 250 PL/SQL RECORD 251 PL/SQL TABLE 252 PL/SQL BOOLEAN
default_value	OUT	1 if the argument being described has a default value; otherwise, the value is 0.
in_out	OUT	Describes the mode of the parameter: 0 IN 1 OUT 2 IN OUT
length	OUT	The data length, in bytes, of the argument being described.
precision	OUT	If the argument being described is of datatype 2 (NUMBER), this parameter is the precision of that number.
scale	OUT	If the argument being described is of datatype 2 (NUMBER, etc.), this parameter is the scale of that number.
radix	OUT	If the argument being described is of datatype 2 (NUMBER, etc.), this parameter is the radix of that number.
spare	OUT	Reserved for future functionality.

Table 7 – 5 (continued) DBMS_DESCRIBE.DESCRIBE_PROCEDURE Parameters

Return Values

All values from DESCRIBE_PROCEDURE are returned in its OUT parameters. The datatypes for these are PL/SQL tables, to accommodate a variable number of parameters.

Examples

One use of the DESCRIBE_PROCEDURE procedure would be as an external service interface.

For example, consider a client that provides an OBJECT_NAME of SCOTT.ACCOUNT_UPDATE where ACCOUNT_UPDATE is an overloaded function with specification:

```
table account (account_no number, person_id number,
               balance number(7,2))
table person  (person_id number(4), person_nm varchar2(10))

function ACCOUNT_UPDATE (account_no    number,
                        person          person%rowtype,
                        amounts         dbms_describe.number_table,
                        trans_date      date)
return                accounts.balance%type;

function ACCOUNT_UPDATE (account_no    number,
                        person          person%rowtype,
                        amounts         dbms_describe.number_table,
                        trans_no        number)
return                accounts.balance%type;
```

The describe of this procedure might look similar to the output shown below.

overload	position	argument	level	datatype	length	prec	scale	rad
	1	0	0	2	22	7	2	10
	1	1	ACCOUNT	0	2	0	0	0
	1	2	PERSON	0	250	0	0	0
	1	1	PERSON_ID	1	2	22	4	0
	1	2	PERSON_NM	1	1	10	0	0
	1	3	AMOUNTS	0	251	0	0	0
	1	1		1	2	22	0	0
	1	4	TRANS_DATE	0	12	0	0	0
	2	0		0	2	22	7	2
	2	1	ACCOUNT_NO	0	2	22	0	0
	2	2	PERSON	0	2	22	4	0
	2	3	AMOUNTS	0	251	22	4	0
	2	1		1	2	0	0	0
	2	4	TRANS_NO	0	2	0	0	0

The following PL/SQL procedure has as its parameters all of the PL/SQL datatypes:

```
CREATE OR REPLACE PROCEDURE p1 (
    pvc2    IN    VARCHAR2,
    pvc     OUT   VARCHAR,
    pstr    IN OUT STRING,
    plong   IN    LONG,
    prowid  IN    ROWID,
    pchara  IN    CHARACTER,
    pchar   IN    CHAR,
    praw    IN    RAW,
    plraw   IN    LONG RAW,
    pbinint IN    BINARY_INTEGER,
    pplsint IN    PLS_INTEGER,
    pbool   IN    BOOLEAN,
    pnat    IN    NATURAL,
    ppos    IN    POSITIVE,
    pposn   IN    POSITIVEN,
    pnatn   IN    NATURALN,
    pnum    IN    NUMBER,
    pintgr  IN    INTEGER,
    pint    IN    INT,
    psmall  IN    SMALLINT,
    pdec    IN    DECIMAL,
    preal   IN    REAL,
    pfloat  IN    FLOAT,
    pnumer  IN    NUMERIC,
    pdp     IN    DOUBLE PRECISION,
    pdate   IN    DATE,
    pmls    IN    MLSLABEL) AS

BEGIN
    NULL;
END;
```

If you describe this procedure using the package below:

```
CREATE OR REPLACE PACKAGE describe_it AS

    PROCEDURE desc_proc (name VARCHAR2);

END describe_it;

CREATE OR REPLACE PACKAGE BODY describe_it AS

    PROCEDURE prt_value(val VARCHAR2, isize INTEGER) IS
        n INTEGER;
    BEGIN
        n := isize - LENGTHB(val);
```

```

IF n < 0 THEN
    n := 0;
END IF;
DBMS_OUTPUT.PUT(val);
FOR i in 1..n LOOP
    DBMS_OUTPUT.PUT(' ');
END LOOP;
END prt_value;

PROCEDURE desc_proc (name VARCHAR2) IS

    overload    DBMS_DESCRIBE.NUMBER_TABLE;
    position    DBMS_DESCRIBE.NUMBER_TABLE;
    c_level     DBMS_DESCRIBE.NUMBER_TABLE;
    arg_name    DBMS_DESCRIBE.VARCHAR2_TABLE;
    dty         DBMS_DESCRIBE.NUMBER_TABLE;
    def_val     DBMS_DESCRIBE.NUMBER_TABLE;
    p_mode      DBMS_DESCRIBE.NUMBER_TABLE;
    length      DBMS_DESCRIBE.NUMBER_TABLE;
    precision   DBMS_DESCRIBE.NUMBER_TABLE;
    scale       DBMS_DESCRIBE.NUMBER_TABLE;
    radix       DBMS_DESCRIBE.NUMBER_TABLE;
    spare       DBMS_DESCRIBE.NUMBER_TABLE;
    idx         INTEGER := 0;

BEGIN
    DBMS_DESCRIBE.DESCRIBE_PROCEDURE(
        name,
        null,
        null,
        overload,
        position,
        c_level,
        arg_name,
        dty,
        def_val,
        p_mode,
        length,
        precision,
        scale,
        radix,
        spare);

    DBMS_OUTPUT.PUT_LINE('Position      Name          DTY   Mode');
    LOOP
        idx := idx + 1;
        prt_value(TO_CHAR(position(idx)), 12);
        prt_value(arg_name(idx), 12);
        prt_value(TO_CHAR(dty(idx)), 5);
    END LOOP;

```

```

        prt_value(TO_CHAR(p_mode(idx)), 5);
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.NEW_LINE;
        DBMS_OUTPUT.NEW_LINE;

    END desc_proc;
END describe_it;
```

then the results, as shown below, list all the numeric codes for the PL/SQL datatypes:

Position	Name	Datatype_Code	Mode
1	PVC2	1	0
2	PVC	1	1
3	PSTR	1	2
4	PLONG	8	0
5	PROWID	11	0
6	PCHARA	96	0
7	PCHAR	96	0
8	PRAW	23	0
9	PLRAW	24	0
10	PBININT	3	0
11	PPLSINT	3	0
12	PBOOL	252	0
13	PNAT	3	0
14	PPOS	3	0
15	PPOSN	3	0
16	PNATN	3	0
17	PNUM	2	0
18	PINTGR	2	0
19	PINT	2	0
20	PSMALL	2	0
21	PDEC	2	0
22	PREAL	2	0
23	PFLOAT	2	0
24	PNUMER	2	0
25	PDP	2	0
26	PDATE	12	0
27	PMLS	106	0

Listing Information about Procedures and Packages

The following data dictionary views provide information about procedures and packages:

- ALL_ERRORS, USER_ERRORS, DBA_ERRORS
- ALL_SOURCE, USER_SOURCE, DBA_SOURCE
- USER_OBJECT_SIZE, DBA_OBJECT_SIZE

The OBJECT_SIZE views show the sizes of the PL/SQL objects. For a complete description of these data dictionary views, see your *Oracle7 Server Reference* manual.

The following statements are used in Examples 1 through 3:

```
CREATE PROCEDURE fire_emp(emp_id NUMBER) AS
BEGIN
    DELETE FROM em WHERE empno = emp_id;
END;
/
CREATE PROCEDURE hire_emp (name VARCHAR2, job VARCHAR2,
    mgr NUMBER, hiredate DATE, sal NUMBER,
    comm NUMBER, deptno NUMBER)
IS
BEGIN
    INSERT INTO emp VALUES (emp_sequence.NEXTVAL, name,
        job, mgr, hiredate, sal, comm, deptno);
END;
/
```

The first CREATE PROCEDURE statement has an error in the DELETE statement. (The 'p' is absent from 'emp'.)

Example 1
Listing Compilation
Errors for Objects

The following query returns all the errors for the objects in the associated schema:

```
SELECT name, type, line, position, text
FROM user_errors;
```

The following results are returned:

NAME	TYPE	LIN	POS	TEXT
-----	----	---	----	-----
FIRE_EMP	PROC	3	15	PL/SQL-00201: identifier 'EM' must be declared
FIRE_EMP	PROC	3	3	PL/SQL: SQL Statement ignored

Example 2
Listing Source Code
for a Procedure

The following query returns the source code for the HIRE_EMP procedure created in the example statement at the beginning of this section:

```
SELECT line, text FROM user_source
       WHERE name = 'HIRE_EMP';
```

The following results are returned:

```
LINE    TEXT
-----
1  PROCEDURE hire_emp (name VARCHAR2, job VARCHAR2,
2    mgr NUMBER, hiredate DATE, sal NUMBER,
3    comm NUMBER, deptno NUMBER)
4  IS
5  BEGIN
6    INSERT INTO emp VALUES (emp_seq.NEXTVAL, name,
7    job, mgr, hiredate, sal, comm, deptno);
8  END;
```

Example 3
Listing Size
Information for a
Procedure

The following query returns information about the amount of space in the SYSTEM tablespace that is required to store the HIRE_EMP procedure:

```
SELECT name, source_size + parsed_size + code_size +
       error_size "TOTAL SIZE"
FROM user_object_size
WHERE name = 'HIRE_EMP';
```

The following results are returned:

NAME	TOTAL SIZE
-----	-----
HIRE_EMP	3897

PL/SQL Input/Output

This chapter describes how to use Oracle-supplied packages that allow PL/SQL to communicate with external processes, sessions, and files.


The packages are

- DBMS_PIPE, to send and receive information between sessions, asynchronously.
- DBMS_OUTPUT, to send messages from a PL/SQL program to other PL/SQL programs in the same session, or to a display window running SQL*Plus.
- UTL_FILE, which allows a PL/SQL program to read information from a disk file, and write information to a file.

Database Pipes

The DBMS_PIPE package allows two or more sessions in the same instance to communicate. Oracle *pipes* are similar in concept to the pipes used in UNIX, but Oracle pipes are not implemented using the operating system pipe mechanisms. Information sent through Oracle pipes is buffered in the system global area (SGA). All information in pipes is lost when the instance is shut down.

Depending upon your security requirements, you may choose to use either a *public pipe* or a *private pipe*.

 **Attention:** Pipes are independent of transactions. Be careful using pipes when transaction control can be affected.

Summary

Table 8 – 1 summarizes the procedures you can call in the DBMS_PIPE package.

Function/Procedure	Description	Refer to Page
CREATE_PIPE	Explicitly create a pipe (necessary for private pipes).	8 – 4
PACK_MESSAGE	Build message in local buffer.	8 – 6
SEND_MESSAGE	Send message on named pipe. Implicitly create a public pipe if named pipe does not exist.	8 – 6
RECEIVE_MESSAGE	Copy message from named pipe into local buffer.	8 – 8
NEXT_ITEM_TYPE	Return datatype of next item in buffer.	8 – 9
UNPACK_MESSAGE	Access next item in buffer.	8 – 10
REMOVE_PIPE	Remove the named pipe.	8 – 10
PURGE	Purge contents of named pipe.	8 – 11
RESET_BUFFER	Purge contents of local buffer.	8 – 12
UNIQUE_SESSION_NAME	Return unique session name.	8 – 12

Table 8 – 1 DBMS_PIPE Package Functions and Procedures

Creating the DBMS_PIPE Package

To create the DBMS_PIPE package, submit the DBMSPIPE.SQL and PRVTPPIPE.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script. See page 7 – 39 for information on granting the necessary privileges to users who will be executing this package.

Public Pipes

You can create a public pipe either implicitly or explicitly. For *implicit* public pipes, the pipe is automatically created when referenced for the first time, and it disappears when it no longer contains data. Because the pipe descriptor is stored in the SGA, there is some space usage overhead until the empty pipe is aged out of the cache.

You can create an *explicit* public pipe by calling the CREATE_PIPE function with the PRIVATE flag set to FALSE. You must deallocate explicitly-created pipes by calling the REMOVE_PIPE function.

The domain of a public pipe is the schema in which it was created, either explicitly or implicitly.

Writing and Reading

Each public pipe works asynchronously. Any number of schema users can write to a public pipe, as long as they have EXECUTE permission on the DBMS_PIPE package, and know the name of the public pipe.

Any schema user with the appropriate privileges and knowledge can read information from a public pipe. However, once buffered information is read by one user, it is emptied from the buffer, and is not available for other readers of the same pipe.

The sending session builds a message using one or more calls to the PACK_MESSAGE procedure. This procedure adds the message to the session's local message buffer. The information in this buffer is sent by calling the SEND_MESSAGE procedure, designating the pipe name to be used to send the message. When SEND_MESSAGE is called, all messages that have been stacked in the local buffer are sent.

A process that wants to receive a message calls the RECEIVE_MESSAGE procedure, designating the pipe name from which to receive the message. The process then calls the UNPACK_MESSAGE procedure to access each of the items in the message.

Private Pipes

You must explicitly create a private pipe by calling the `CREATE_PIPE` function. Once created, the private pipe persists in shared memory until you explicitly deallocate it by calling the `REMOVE_PIPE` function. A private pipe is also deallocated when the database instance is shut down.

You cannot create a private pipe if an implicit pipe exists in memory and has the same name as the private pipe you are trying to create. In this case `CREATE_PIPE` returns an error.

Access to a private pipe is restricted to the following:

- sessions running under the same userid as the creator of the pipe
- stored subprograms executing in the same userid privilege domain as the pipe creator
- users connected as `SYSDBA` or `INTERNAL`

An attempt by any other user to send or receive messages on the pipe, or to remove the pipe, results in an immediate error. Any attempt by another user to create a pipe with the same name also causes an error.

As with public pipes, you must first build your message using calls to `PACK_MESSAGE` before calling `SEND_MESSAGE`. Similarly you must call `RECEIVE_MESSAGE` to retrieve the message before accessing the items in the message by calling `UNPACK_MESSAGE`.

Errors

`DBMS_PIPE` package routines can return the following errors:

ORA-23321: Pipename may not be null

ORA-23322: Insufficient privilege to access pipe

ORA-23321 can be returned by `CREATE_PIPE`, or any subprogram that takes a pipe name as a parameter. ORA-23322 can be returned by any subprogram that references a private pipe in its parameter list.

CREATE_PIPE

Call `CREATE_PIPE` to explicitly create a public or private pipe. If the `PRIVATE` flag is `TRUE`, the pipe creator is assigned as the owner of the private pipe. Explicitly created pipes can only be removed by calling `REMOVE_PIPE`, or by shutting down the instance.



Warning: Do not use a pipe name beginning with `ORA$`; these names are reserved for use by Oracle Corporation.

The parameters for the CREATE_PIPE function are shown in Table 8 – 2 and the possible return values and their meanings are described in Table 8 – 3. The syntax for this function is

```
DBMS_PIPE.CREATE_PIPE(pipeName      IN VARCHAR2,
                        maxPipeSize  IN INTEGER DEFAULT 8192,
                        private      IN BOOLEAN DEFAULT TRUE)

RETURN INTEGER;
```

Parameter	Description
pipeName	Specify a name for the pipe that you are creating. You will need to use this name when you call SEND_MESSAGE and RECEIVE_MESSAGE. This name must be unique across the instance.
maxPipeSize	Specify the maximum size allowed for the pipe, in bytes. The total size of all of the messages on the pipe cannot exceed this amount. The message is blocked if it exceeds this maximum. The default MAXPIPIESIZE is 8192 bytes. The MAXPIPIESIZE for a pipe becomes a part of the characteristics of the pipe and persists for the life of the pipe. Callers of SEND_MESSAGE with larger values cause the MAXPIPIESIZE to be increased. Callers with a smaller value simply use the existing, larger value.
private	Use the default, TRUE, to create a private pipe. Public pipes can be implicitly created when you call SEND_MES-SAGE.

Table 8 – 2 DBMS_PIPE.CREATE_PIPE Function Parameters

Return Value or Error	Description
0	Indicates the pipe was successfully created. If the pipe already exists and the user attempting to create it is authorized to use it, Oracle returns 0, indicating success, and any data already in the pipe remains. If a user connected as SYSDBA/SYSOPER re-creates a pipe, Oracle returns status 0, but the ownership of the pipe remains unchanged.
ORA-23322	Indicates a failure due to naming conflict. If a pipe with the same name exists and was created by a different user, Oracle signals error ORA-23322, indicating the naming conflict.

Table 8 – 3 DBMS_PIPE.CREATE_PIPE Function Return Values

PACK_MESSAGE Procedures

To send a message, first make one or more calls to `PACK_MESSAGE` to build your message in the local message buffer. Then call `SEND_MESSAGE` to send the message in the local buffer on the named pipe.

The `PACK_MESSAGE` procedure is overloaded to accept items of type `VARCHAR2`, `NUMBER`, or `DATE`. In addition to the data bytes, each item in the buffer requires one byte to indicate its type, and two bytes to store its length. One additional byte is needed to terminate the message. If the message buffer exceeds 4096 bytes, Oracle raises exception `ORA-6558`.

When you call `SEND_MESSAGE` to send this message, you must indicate the name of the pipe on which you want to send the message. If this pipe already exists, you must have sufficient privileges to access this pipe. If the pipe does not already exist, it is created automatically.



Warning: Do not use a pipe name beginning with `ORA$`; these names are reserved for use by Oracle Corporation.

Syntax

The syntax for the `PACK_MESSAGE` procedures is shown below. Note that the `UNPACK_MESSAGE` procedure itself is overloaded to accept items of type `VARCHAR2`, `NUMBER`, or `DATE`. There are two additional procedures to pack `RAW` and `ROWID` items.

```
DBMS_PIPE.PACK_MESSAGE      (item IN VARCHAR2);
DBMS_PIPE.PACK_MESSAGE      (item IN NUMBER);
DBMS_PIPE.PACK_MESSAGE      (item IN DATE);
DBMS_PIPE.PACK_MESSAGE_RAW  (item IN RAW);
DBMS_PIPE.PACK_MESSAGE_ROWID (item IN ROWID);
```

SEND_MESSAGE

The parameters for the `SEND_MESSAGE` function are shown in Table 8 – 4 and the possible return values and their meanings are described in Table 8 – 5. The syntax for this function is shown below.

```
DBMS_PIPE.SEND_MESSAGE(pipeName IN VARCHAR2,
                        timeout    IN INTEGER DEFAULT MAXWAIT,
                        maxPipesize IN INTEGER DEFAULT 8192)
RETURN INTEGER;
```

Parameter	Description
pipename	Specify the name of the pipe on which you want to place the message. If you are using an explicit pipe, this is the name that you specified when you called CREATE_PIPE.
timeout	Specify the timeout period in seconds. This is the time to wait while attempting to place a message on the pipe; the return values are explained below. The default value is the constant MAXWAIT, which is defined as 86400000 (1000 days).
maxpipesize	<p>Specify the maximum size allowed for the pipe, in bytes. The total size of all of the messages on the pipe cannot exceed this amount. The message is blocked if it exceeds this maximum. The default MAXPIPE_SIZE is 8192 bytes.</p> <p>The MAXPIPE_SIZE for a pipe becomes a part of the characteristics of the pipe and persists for the life of the pipe. Callers of SEND_MESSAGE with larger values cause the MAXPIPE_SIZE to be increased. Callers with a smaller value simply use the existing, larger value. Specifying MAXPIPE_SIZE as part of the SEND_MESSAGE procedure eliminates the need for a separate call to open the pipe. If you created the pipe explicitly, you can use the optional MAXPIPE_SIZE parameter to override the creation pipe size specification.</p>

Table 8 – 4 DBMS_PIPE.SEND_MESSAGE Function Parameters

Return Value or Error	Description
0	<p>Indicates the pipe was successfully created.</p> <p>If the pipe already exists and the user attempting to create it is authorized to use it, Oracle returns 0, indicating success, and any data already in the pipe remains.</p> <p>If a user connected as SYSDBA/SYSOPER re-creates a pipe, Oracle returns status 0, but the ownership of the pipe remains unchanged.</p>
1	Indicates the pipe has timed out. This procedure can time-out either because it cannot get a lock on the pipe, or because the pipe remains too full to be used. If the pipe was implicitly created and is empty, it is removed.

Table 8 – 5 DBMS_PIPE.SEND_MESSAGE Function Return Values

Return Value or Error	Description
3	Indicates an interrupt has occurred. If the pipe was implicitly created and is empty, it is removed.
ORA-23322	Indicates insufficient privileges to write to the pipe. If a pipe with the same name exists and was created by a different user, Oracle signals error ORA-23322, indicating the naming conflict.

Table 8 – 5 DBMS_PIPE.SEND_MESSAGE Function Return Values

RECEIVE_MESSAGE

To receive a message from a pipe, first call `RECEIVE_MESSAGE` to copy the message into the local message buffer. When you receive a message, it is removed from the pipe; that is, a message can only be received once. For implicitly created pipes, the pipe is removed after the last record is removed from the pipe.

If the pipe that you specify when you call `RECEIVE_MESSAGE` does not already exist, Oracle implicitly creates the pipe and then waits to receive the message. If the message does not arrive within a designated timeout interval, the call returns and the pipe is removed.

After receiving the message, you must make one or more calls to `UNPACK_MESSAGE` to access the individual items in the message. The `UNPACK_MESSAGE` procedure is overloaded to unpack items of type `DATE`, `NUMBER`, `VARCHAR2`, and there are two additional procedures to unpack `RAW` and `ROWID` items. If you do not know the type of data that you are attempting to unpack, you can call `NEXT_ITEM_TYPE` to determine the type of the next item in the buffer.

Syntax

The parameters for the `RECEIVE_MESSAGE` function are shown in Table 8 – 6 and the possible return values and their meanings are described in Table 8 – 7. The syntax for this function is shown below.

```
DBMS_PIPE.RECEIVE_MESSAGE (pipename      IN VARCHAR2,
                             timeout      IN INTEGER
                             DEFAULT maxwait)
RETURN INTEGER;
```

Parameter	Description
pipename	Specify the name of the pipe on which you want to receive a message. Names beginning with ORA\$ are reserved for use by Oracle.
timeout	Specify the timeout period in seconds. This is the time to wait to receive a message on the pipe. The default value is the constant MAXWAIT, which is defined as 86400000 (1000 days). A timeout of 0 allows you to read without blocking.

Table 8 – 6 DBMS_PIPE.RECEIVE_MESSAGE Function Parameters

Return Value or Error	Description
0	Indicates the message was received successfully.
1	Indicates the pipe has timed out. If the pipe was implicitly created and is empty, it is removed.
2	Indicates the record in the pipe is too large for the buffer. (This should not happen.)
3	Indicates an interrupt has occurred.
ORA-23322	Indicates the user has insufficient privileges to read from the pipe.

Table 8 – 7 DBMS_PIPE.RECEIVE_MESSAGE Function Return Values

NEXT_ITEM_TYPE

After you have called RECEIVE_MESSAGE to place pipe information in a local buffer, you can call NEXT_ITEM_TYPE to determine the datatype of the next item in the local message buffer. When NEXT_ITEM_TYPE returns 0, the local buffer is empty.

Syntax

The possible return values and their meanings for the NEXT_ITEM_TYPE function are described in Table 8 – 8. The syntax for this function is shown below.

```
DBMS_PIPE.NEXT_ITEM_TYPE RETURN INTEGER;
```

Return Value	Description
0	no more items
6	NUMBER
9	VARCHAR2
12	DATE

Table 8 – 8 DBMS_PIPE.NEXT_ITEM_TYPE Function Return Values

UNPACK_MESSAGE Procedures

After you have called `RECEIVE_MESSAGE` to place pipe information in a local buffer, you call `UNPACK_MESSAGE` to retrieve items from the buffer.

Syntax

The syntax for the `UNPACK_MESSAGE` procedures is shown below. Note that the `UNPACK_MESSAGE` procedure is overloaded to return items of type `VARCHAR2`, `NUMBER`, or `DATE`. There are two additional procedures to unpack `RAW` and `ROWID` items.

```
DBMS_PIPE.UNPACK_MESSAGE      (item OUT VARCHAR2);
DBMS_PIPE.UNPACK_MESSAGE      (item OUT NUMBER);
DBMS_PIPE.UNPACK_MESSAGE      (item OUT DATE);
DBMS_PIPE.UNPACK_MESSAGE_RAW  (item IN DATE);
DBMS_PIPE.UNPACK_MESSAGE_ROWID (item IN DATE);
```

If the message buffer contains no more items, or if the item received is not of the same type as that requested, the `ORA-2000` exception is raised.

REMOVE_PIPE

Pipes created implicitly by `SEND_MESSAGE` are automatically removed when empty.

Pipes created explicitly by `CREATE_PIPE` are removed only by calling `REMOVE_PIPE` or when the instance is shut down. All unconsumed records in the pipe are removed before the pipe is deleted. This is similar to calling `PURGE` on an implicitly created pipe.

Syntax

The REMOVE_PIPE function accepts only one parameter—the name of the pipe that you want to remove. The possible return values and their meanings are described in Table 8 – 9. The syntax for this function is

```
DBMS_PIPE.REMOVE_PIPE(pipeName IN VARCHAR2)
RETURN INTEGER;
```

Return Value or Error	Description
0	Indicates the pipe was successfully removed. If the pipe does not exist, or if the pipe already exists and the user attempting to remove it is authorized to do so, Oracle returns 0, indicating success, and any data remaining in the pipe is removed.
ORA-23322	Indicates a failure due to insufficient privileges. If the pipe exists, but the user is not authorized to access the pipe, Oracle signals error ORA-23322, indicating insufficient privileges.

Table 8 – 9 DBMS_PIPE.REMOVE_PIPE Function Return Values

Managing Pipes

The DBMS_PIPE package contains additional procedures and functions that you might find useful.

Purging the Contents of a Pipe

Call PURGE to empty the contents of a pipe. An empty implicitly created pipe is aged out of the shared global area according to the least-recently-used algorithm. Thus, calling PURGE lets you free the memory associated with an implicitly created pipe.

Because PURGE calls RECEIVE_MESSAGE, the local buffer might be overwritten with messages as they are purged from the pipe. Also, you can receive an ORA-23322, insufficient privileges, error if you attempt to purge a pipe to which you have insufficient access rights.

```
DBMS_PIPE.PURGE(pipeName IN VARCHAR2);
```


Resetting the Message Buffer

Call `RESET_BUFFER` to reset the `PACK_MESSAGE` and `UNPACK_MESSAGE` positioning indicators to 0. Because all pipes share a single buffer, you may find it useful to reset the buffer before using a new pipe. This ensures that the first time you attempt to send a message to your pipe, you do not inadvertently send an expired message remaining in the buffer.

Syntax

The syntax for the `RESET_BUFFER` procedure is shown below.

```
DBMS_PIPE.RESET_BUFFER;
```

Getting a Unique Session Name

Call `UNIQUE_SESSION_NAME` to receive a name that is unique among all of the sessions that are currently connected to a database. Multiple calls to this function from the same session always return the same value. The return value can be up to 30 bytes. You might find it useful to use this function to supply the `PIPENAME` parameter for your `SEND_MESSAGE` and `RECEIVE_MESSAGE` calls.

```
DBMS_PIPE.UNIQUE_SESSION_NAME RETURN VARCHAR2;
```

Example 1: Debugging

The following example shows a procedure a PL/SQL program can call to place debugging information in a pipe:

```
CREATE OR REPLACE PROCEDURE debug (msg VARCHAR2) AS
    status NUMBER;
BEGIN
    dbms_pipe.pack_message(LENGTH(msg));
    dbms_pipe.pack_message(msg);
    status := dbms_pipe.send_message('plsql_debug');
    IF status != 0 THEN
        raise_application_error(-20099, 'Debug error');
    END IF;
END debug;
```

This example shows the Pro*C code that receives messages from the `PLSQL_DEBUG` pipe in the PL/SQL example above, and displays the messages. If the Pro*C session is run in a separate window, it can be used to display any messages that are sent to the debug procedure from a PL/SQL program executing in a separate session.

```

#include <stdio.h>
#include <string.h>

EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR username[20];
    int      status;
    int      msg_length;
    char     retval[2000];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

void sql_error();

main()
{
    /* prepare username */
    strcpy(username.arr, "SCOTT/TIGER");
    username.len = strlen(username.arr);

    EXEC SQL WHENEVER SQLERROR DO sql_error();
    EXEC SQL CONNECT :username;

    printf("connected\n");

    /* start an endless loop to look for and print
       messages on the pipe */
    for (;;)
    {
        EXEC SQL EXECUTE
            DECLARE
                len INTEGER;
                typ INTEGER;
                sta INTEGER;
                chr VARCHAR2(2000);
            BEGIN
                chr := '';
                sta := dbms_pipe.receive_message('plsql_debug');
                IF sta = 0 THEN
                    dbms_pipe.unpack_message(len);
                    dbms_pipe.unpack_message(chr);
                END IF;
                :status := sta;
                :retval := chr;
                IF len IS NOT NULL THEN
                    :msg_length := len;
                ELSE
                    :msg_length := 2000;
                END IF;
            
```

```

        END;
    END-EXEC;
    if (status == 0)
        printf("\n%.*s\n", msg_length, retval);
    else
        printf("abnormal status, value is %d\n", status);
    }
}

void sql_error()
{
    char msg[1024];
    int rlen, len;
    len = sizeof(msg);
    sqlglm(msg, &len, &rlen);
    printf("ORACLE ERROR\n");
    printf("%.*s\n", rlen, msg);
    exit(1);
}

```

Example 2: Execute System Commands

The following example shows PL/SQL and Pro*C code that can let a PL/SQL stored procedure (or anonymous block) call PL/SQL procedures to send commands over a pipe to a Pro*C program that is listening for them.

The Pro*C program just sleeps, waiting for a message to arrive on the named pipe. When a message arrives, the C program processes it, carrying out the required action, such as executing a UNIX command through the *system()* call, or executing a SQL command using embedded SQL.

DAEMON.SQL is the source code for the PL/SQL package. This package contains procedures that use the DBMS_PIPE package to send and receive message to and from the Pro*C daemon. Note that full handshaking is used. The daemon will always send a message back to the package (except in the case of the 'STOP' command). This is valuable, since it allows the PL/SQL procedures to be sure that the Pro*C daemon is running.

You can call the DAEMON packaged procedures from an anonymous PL/SQL block using SQL*Plus or Server Manager. For example:

```
SVRMGR> variable rv number
SVRMGR> execute :rv := DAEMON.EXECUTE_SYSTEM('ls -la');
```

would, on a UNIX system, cause the Pro*C daemon to execute the command *system("ls -la")*.

Remember that the daemon needs to be running first. So you might want to run it in the background, or in another window beside the SQL*Plus or Server Manager session from which you call it.

The DAEMON.SQL also uses the DBMS_OUTPUT package (see page 8 – 21) to display the results. For this example to work, you must have execute privileges on this package.

DAEMON.SQL

This is the code for the PL/SQL DAEMON package:

```
CREATE OR REPLACE PACKAGE daemon AS
    FUNCTION execute_sql(command VARCHAR2,
                        timeout NUMBER DEFAULT 10)
        RETURN NUMBER;

    FUNCTION execute_system(command VARCHAR2,
                        timeout NUMBER DEFAULT 10)
        RETURN NUMBER;

    PROCEDURE stop(timeout NUMBER DEFAULT 10);
END daemon;
/
CREATE OR REPLACE PACKAGE BODY daemon AS

    FUNCTION execute_system(command VARCHAR2,
                        timeout NUMBER DEFAULT 10)
        RETURN NUMBER IS

        status          NUMBER;
        result           VARCHAR2(20);
        command_code    NUMBER;
        pipe_name       VARCHAR2(30);
    BEGIN
        pipe_name := DBMS_PIPE.UNIQUE_SESSION_NAME;

        DBMS_PIPE.PACK_MESSAGE('SYSTEM');
        DBMS_PIPE.PACK_MESSAGE(pipe_name);
        DBMS_PIPE.PACK_MESSAGE(command);
        status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
```

```

IF status <> 0 THEN
    RAISE_APPLICATION_ERROR(-20010,
        'Execute_system: Error while sending. Status = ' ||
        status);
END IF;

status := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name, timeout);
IF status <> 0 THEN
    RAISE_APPLICATION_ERROR(-20011,
        'Execute_system: Error while receiving.
        Status = ' || status);
END IF;

DBMS_PIPE.UNPACK_MESSAGE(result);
IF result <> 'done' THEN
    RAISE_APPLICATION_ERROR(-20012,
        'Execute_system: Done not received.');
```

```

END IF;

DBMS_PIPE.UNPACK_MESSAGE(command_code);
DBMS_OUTPUT.PUT_LINE('System command executed. result = ' ||
    command_code);

RETURN command_code;
END execute_system;

FUNCTION execute_sql(command VARCHAR2,
    timeout NUMBER DEFAULT 10)
RETURN NUMBER IS

    status      NUMBER;
    result      VARCHAR2(20);
    command_code NUMBER;
    pipe_name   VARCHAR2(30);

BEGIN
    pipe_name := DBMS_PIPE.UNIQUE_SESSION_NAME;

    DBMS_PIPE.PACK_MESSAGE('SQL');
    DBMS_PIPE.PACK_MESSAGE(pipe_name);
    DBMS_PIPE.PACK_MESSAGE(command);
    status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
    IF status <> 0 THEN
        RAISE_APPLICATION_ERROR(-20020,
            'Execute_sql: Error while sending. Status = ' || status);
    END IF;

    status := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name, timeout);

```

```

IF status <> 0 THEN
    RAISE_APPLICATION_ERROR(-20021,
        'execute_sql: Error while receiving.
        Status = ' || status);
END IF;

DBMS_PIPE.UNPACK_MESSAGE(result);
IF result <> 'done' THEN
    RAISE_APPLICATION_ERROR(-20022,
        'execute_sql: done not received. ');
END IF;

DBMS_PIPE.UNPACK_MESSAGE(command_code);
DBMS_OUTPUT.PUT_LINE
    ('SQL command executed.  sqlcode = ' || command_code);
RETURN command_code;
END execute_sql;

PROCEDURE stop(timeout NUMBER DEFAULT 10) IS
    status NUMBER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('STOP');
    status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
    IF status <> 0 THEN
        RAISE_APPLICATION_ERROR(-20030,
            'stop: error while sending.  status = ' || status);
    END IF;
END stop;
END daemon;

```

daemon.pc

This is the code for the Pro*C daemon. You must precompile this using the Pro*C Precompiler, Version 1.5.x or later. You must also specify the USERID and SQLCHECK options, as the example contains embedded PL/SQL code. For example:

```
proc iname=daemon userid=scott/tiger sqlcheck=semantics
```

Then C-compile and link in the normal way.

```

#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;

```

```

EXEC SQL BEGIN DECLARE SECTION;
    char *uid = "scott/tiger";
    int status;
    VARCHAR command[20];
    VARCHAR value[2000];
    VARCHAR return_name[30];
EXEC SQL END DECLARE SECTION;

void
connect_error()
{
    char msg_buffer[512];
    int msg_length;
    int buffer_size = 512;

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    sqlglm(msg_buffer, &buffer_size, &msg_length);
    printf("Daemon error while connecting:\n");
    printf("%. *s\n", msg_length, msg_buffer);
    printf("Daemon quitting.\n");
    exit(1);
}

void
sql_error()
{
    char msg_buffer[512];
    int msg_length;
    int buffer_size = 512;

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    sqlglm(msg_buffer, &buffer_size, &msg_length);
    printf("Daemon error while executing:\n");
    printf("%. *s\n", msg_length, msg_buffer);
    printf("Daemon continuing.\n");
}

main()
{
    EXEC SQL WHENEVER SQLERROR DO connect_error();
    EXEC SQL CONNECT :uid;
    printf("Daemon connected.\n");

    EXEC SQL WHENEVER SQLERROR DO sql_error();
    printf("Daemon waiting...\n");
    while (1) {
        EXEC SQL EXECUTE
            BEGIN
                :status := DBMS_PIPE.RECEIVE_MESSAGE('daemon');
                IF :status = 0 THEN

```

```

        DBMS_PIPE.UNPACK_MESSAGE(:command);
    END IF;
END;
END-EXEC;
if (status == 0)
{
    command.arr[command.len] = '\0';
    if (!strcmp((char *) command.arr, "STOP"))
    {
        printf("Daemon exiting.\n");
        break;
    }

    else if (!strcmp((char *) command.arr, "SYSTEM"))
    {
        EXEC SQL EXECUTE
        BEGIN
            DBMS_PIPE.UNPACK_MESSAGE(:return_name);
            DBMS_PIPE.UNPACK_MESSAGE(:value);
        END;
        END-EXEC;
        value.arr[value.len] = '\0';
        printf("Will execute system command '%s'\n", value.arr);

        status = system(value.arr);
        EXEC SQL EXECUTE
        BEGIN
            DBMS_PIPE.PACK_MESSAGE('done');
            DBMS_PIPE.PACK_MESSAGE(:status);
            :status := DBMS_PIPE.SEND_MESSAGE(:return_name);
        END;
        END-EXEC;

        if (status)
        {
            printf
            ("Daemon error while responding to system command.");
            printf("    status: %d\n", status);
        }
    }
    else if (!strcmp((char *) command.arr, "SQL")) {
        EXEC SQL EXECUTE
        BEGIN
            DBMS_PIPE.UNPACK_MESSAGE(:return_name);
            DBMS_PIPE.UNPACK_MESSAGE(:value);
        END;
        END-EXEC;
        value.arr[value.len] = '\0';
        printf("Will execute sql command '%s'\n", value.arr);
    }
}

```



```

EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL EXECUTE IMMEDIATE :value;
status = sqlca.sqlcode;

EXEC SQL WHENEVER SQLERROR DO sql_error();
EXEC SQL EXECUTE
BEGIN
    DBMS_PIPE.PACK_MESSAGE('done');
    DBMS_PIPE.PACK_MESSAGE(:status);
    :status := DBMS_PIPE.SEND_MESSAGE(:return_name);
END;
END-EXEC;

if (status)
{
    printf("Daemon error while responding to sql command.");
    printf("  status: %d\n", status);
}
}
else
{
    printf
        ("Daemon error: invalid command '%s' received.\n",
         command.arr);
}
}
else
{
    printf("Daemon error while waiting for signal.");
    printf("  status = %d\n", status);
}
}
EXEC SQL COMMIT WORK RELEASE;
exit(0);
}

```

Output from Stored Procedures and Triggers

Oracle provides a public package, DBMS_OUTPUT, which you can use to send messages from stored procedures, packages, and triggers. The PUT and PUT_LINE procedures in this package allow you to place information in a buffer that can be read by another trigger, procedure, or package.

Server Manager or SQL*Plus can also display messages buffered by the DBMS_OUTPUT procedures. To do this, you must issue the command SET SERVEROUTPUT ON in Server Manager or SQL*Plus.

In a separate PL/SQL procedure or anonymous block, you can display the buffered information by calling the GET_LINE procedure. If you do not call GET_LINE, or do not display the messages on your screen in SQL*Plus or Server Manager, the buffered messages are ignored. The DBMS_OUTPUT package is especially useful for displaying PL/SQL debugging information.

Note: Messages sent using the DBMS_OUTPUT are not actually sent until the sending subprogram or trigger completes. There is no mechanism to flush output during the execution of a procedure.

Summary

Table 8 – 10 shows the procedures that are callable from the DBMS_OUTPUT package:

Function/Procedure	Description	Refer to Page
ENABLE	enable message output	8 – 22
DISABLE	disable message output	8 – 23
PUT_LINE	place a line in the buffer	8 – 23
PUT	place partial line in buffer	8 – 23
NEW_LINE	terminate a line created with PUT	8 – 23
GET_LINE	retrieve one line of information from buffer	8 – 24
GET_LINES	retrieve array of lines from buffer	8 – 24

Table 8 – 10 DBMS_OUTPUT Package Functions and Procedures

Creating the DBMS_OUTPUT Package

To create the DBMS_OUTPUT package, submit the DBMSOTPT.SQL and PRVTOTPT.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script. See page 7 – 39 for information on granting the necessary privileges to users who will be executing this package.

Errors

The DBMS_OUTPUT package routines raise the application error –20000, and the output procedures can return the following errors:

```
ORU-10027: buffer overflow
ORU-10028: line length overflow
```

ENABLE Procedure

This procedure enables calls to PUT, PUT_LINE, NEW_LINE, GET_LINE, and GET_LINES. Calls to these procedures are ignored if the DBMS_OUTPUT package is not enabled. It is not necessary to call this procedure when you use the SERVEROUTPUT option of Server Manager or SQL*Plus.

You must specify the amount of information, in bytes, to buffer. Items are stored in the DBMS_OUTPUT package. If the buffer size is exceeded, you receive the following error message:

```
ORA-20000, ORU-10027: buffer overflow, limit of <buffer_limit>
bytes.
```

Multiple calls to ENABLE are allowed. If there are multiple calls to ENABLE, BUFFER_SIZE is the largest of the values specified. The maximum size is 1000000 and the minimum is 2000.

Syntax

The syntax for the ENABLE procedure is

```
DBMS_OUTPUT.ENABLE(buffer_size IN INTEGER DEFAULT 2000);
```

DISABLE Procedure

The DISABLE procedure disables calls to PUT, PUT_LINE, NEW_LINE, GET_LINE, and GET_LINES, and purges the buffer of any remaining information. As with ENABLE, you do not need to call this procedure if you are using the SERVEROUTPUT option of Server Manager or SQL*Plus.

Syntax

The syntax for the DISABLE procedure is shown below.

```
DBMS_OUTPUT.DISABLE;
```

PUT and PUT_LINE Procedures

You can either place an entire line of information into the buffer by calling PUT_LINE, or you can build a line of information piece by piece by making multiple calls to PUT. Both of these procedures are overloaded to accept items of type VARCHAR2, NUMBER, or DATE to place in the buffer.

All items are converted to VARCHAR2 as they are retrieved. If you pass an item of type NUMBER or DATE, when that item is retrieved, it is formatted with TO_CHAR using the default format. If you want to use a different format, you should pass in the item as VARCHAR2 and format it explicitly.

When you call PUT_LINE, the item that you specify is automatically followed by an end-of-line marker. If you make calls to PUT to build a line, you must add your own end-of-line marker by calling NEW_LINE. GET_LINE and GET_LINES do not return lines that have not been terminated with a newline character.

If your line exceeds the buffer limit, you receive an error message.



Attention: Output that you create using PUT or PUT_LINE is buffered in the SGA. The output cannot be retrieved until the PL/SQL program unit from which it was buffered returns to its caller. So, for example, Server Manager or SQL*Plus do not display DBMS_OUTPUT messages until the PL/SQL program completes. In this release, there is no mechanism for flushing the DBMS_OUTPUT buffers within the PL/SQL program.

Syntax

The PUT and PUT_LINE procedure are overloaded; they can take an IN parameter of either NUMBER, VARCHAR2, or DATE. The syntax for the PUT and PUT_LINE, and the NEW_LINE procedures is

```
DBMS_OUTPUT.PUT      (item IN NUMBER);
DBMS_OUTPUT.PUT      (item IN VARCHAR2);
DBMS_OUTPUT.PUT      (item IN DATE);
DBMS_OUTPUT.PUT_LINE(item IN NUMBER);
DBMS_OUTPUT.PUT_LINE(item IN VARCHAR2);
DBMS_OUTPUT.PUT_LINE(item IN DATE);
DBMS_OUTPUT.NEW_LINE;
```

GET_LINE and GET_LINES Procedures

You can choose to retrieve a single line from the buffer, or an array of lines. Call the GET_LINE procedure to retrieve a single line of buffered information. To reduce the number of calls to the server, call the GET_LINES procedure to retrieve an array of lines from the buffer. You can choose to automatically display this information if you are using Server Manager or SQL*Plus by using the special SET SERVEROUTPUT ON command.

After calling GET_LINE or GET_LINES, any lines not retrieved before the next call to PUT, PUT_LINE, or NEW_LINE are discarded to avoid confusing them with the next message.

Syntax

The parameters for the GET_LINE procedure are described in Table 8 – 11. The syntax for this procedure is shown below.

```
DBMS_OUTPUT.GET_LINE(line  OUT VARCHAR2,
                      status OUT INTEGER);
```

Parameter	Description
line	Returns a single line of buffered information, excluding a final newline character. The maximum length of this parameter is 255 bytes.
status	If the call completes successfully, the status returns as 0. If there are no more lines in the buffer, the status is 1.

Table 8 – 11 DBMS_OUTPUT.GET_LINE Procedure Parameters

The parameters for the GET_LINES procedure are described in Table 8 – 12. The syntax for this procedure is

```
DBMS_OUTPUT.GET_LINES(lines      OUT  CHARARR,
                      numlines IN OUT INTEGER);
```

where CHARARR is a table of VARCHAR2(255), defined as a type in the DBMS_OUTPUT package specification.

Parameter	Description
lines	Returns an array of lines of buffered information. The maximum length of each line in the array is 255 bytes.
numlines	Specify the number of lines you want to retrieve from the buffer. After retrieving the specified number of lines, the procedure returns the number of lines actually retrieved. If this number is less than the number of lines requested, there are no more lines in the buffer.

Table 8 – 12 DBMS_OUTPUT.GET_LINE Procedure Parameters

Examples Using the DBMS_OUTPUT Package

The DBMS_OUTPUT package is commonly used to debug stored procedures and triggers, as shown in example 1. This package can also be used to allow a user to retrieve information about an object and format this output, as shown in example 2.

Example 1 An example of a function that queries the employee table and returns the total salary for a specified department follows. The function includes several calls to the PUT_LINE procedure:

```
CREATE FUNCTION dept_salary (dnum NUMBER) RETURN NUMBER IS
  CURSOR emp_cursor IS
    SELECT sal, comm FROM emp WHERE deptno = dnum;
  total_wages  NUMBER(11, 2) := 0;
  counter      NUMBER(10) := 1;
BEGIN
  FOR emp_record IN emp_cursor LOOP
    emp_record.comm := NVL(emp_record.comm, 0);
    total_wages := total_wages + emp_record.sal
      + emp_record.comm;
    DBMS_OUTPUT.PUT_LINE('Loop number = ' || counter ||
      ' ; Wages = ' || TO_CHAR(total_wages)); /* Debug line */
    counter := counter + 1; /* Increment debug counter */
  END LOOP;
  /* Debug line */
  DBMS_OUTPUT.PUT_LINE('Total wages = ' ||
    TO_CHAR(total_wages));
  RETURN total_wages;
END dept_salary;
```

Assume the EMP table contains the following rows:

EMPNO	SAL	COMM	DEPT
-----	-----	-----	-----
1002	1500	500	20
1203	1000		30
1289	1000		10
1347	1000	250	20

Assume you execute the following statements in the Server Manager SQL Worksheet input pane:

```
SET SERVEROUTPUT ON
VARIABLE salary NUMBER;
EXECUTE :salary := dept_salary(20);
```

You would then see the following information displayed in the output pane:

```
Loop number = 1; Wages = 2000
Loop number = 2; Wages = 3250
Total wages = 3250
```

```
PL/SQL procedure successfully executed.
```

Example 2 This example assumes that the user has used the EXPLAIN PLAN command to retrieve information about the execution plan for a statement and store it in PLAN_TABLE, and that the user has assigned a statement ID to this statement. The example EXPLAIN_OUT procedure retrieves the information from this table and formats the output in a nested manner that more closely depicts the order of steps undergone in processing the SQL statement.

```

/*****
/* Create EXPLAIN_OUT procedure. User must pass STATEMENT_ID to */
/* to procedure, to uniquely identify statement.                */
*****/
CREATE OR REPLACE PROCEDURE explain_out
(statement_id IN VARCHAR2) AS

-- Retrieve information from PLAN_TABLE into cursor
-- EXPLAIN_ROWS.
CURSOR explain_rows IS
    SELECT level, id, position, operation, options,
           object_name
    FROM plan_table
    WHERE statement_id = explain_out.statement_id
    CONNECT BY PRIOR id = parent_id
           AND statement_id = explain_out.statement_id
    START WITH id = 0
    ORDER BY id;

BEGIN
-- Loop through information retrieved from PLAN_TABLE
FOR line IN explain_rows LOOP

-- At start of output, include heading with estimated cost.
IF line.id = 0 THEN
    DBMS_OUTPUT.PUT_LINE ('Plan for statement '
        || statement_id
        || ', estimated cost = ' || line.position);
END IF;

-- Output formatted information. LEVEL is used to
-- determine indentation level.
DBMS_OUTPUT.PUT_LINE (lpad(' ', 2*(line.level-1)) ||
    line.operation || ' ' || line.options || ' ' ||
    line.object_name);
END LOOP;
END;

```


PL/SQL File I/O

The release 7.3 Oracle Server adds file input/output capabilities to PL/SQL. This is done through the supplied package UTL_FILE.

The file I/O capabilities are similar to those of the standard operating system stream file I/O (OPEN, GET, PUT, CLOSE), with some limitations. For example, you call the FOPEN function to return a *file handle*, which you then use in subsequent calls to GET_LINE or PUT to perform stream I/O to a file. When you are done performing I/O on the file, you call FCLOSE to complete any output and to free any resources associated with the file.

Summary

Table 8 – 13 summarizes the procedures you can call in the UTL_FILE package.

Function/Procedure	Description	Refer to Page
FOPEN	Open a file for input or ouput. Create an output file if it does not exist.	8 – 32
IS_OPEN	Determine if a file handle refers to an open file.	8 – 33
FCLOSE	Close a file.	8 – 34
FCLOSE_ALL	Close all open file handles.	8 – 34
GET_LINE	Read a line of text from an open file.	8 – 35
PUT	Write a line to a file. Do not append a line terminator.	8 – 36
PUT_LINE	Write a line to a file. Append an OS–specific line terminator.	8 – 38
PUTF	A PUT procedure with formatting.	8 – 38
NEW_LINE	Write one or more OS–specific line terminators to a file.	8 – 37
FFLUSH	Physically write all pending output to a file.	8 – 40

Table 8 – 13 UTL_FILE Procedures

Security

The PL/SQL file I/O feature is available for both client side and server side PL/SQL. The client implementation is subject to normal operating system file permission checking, and so does not need any additional security constraints. But the server implementation might be running in a privileged mode, and so will need additional security restrictions that limit the power of this feature.

Note: The UTL_FILE package is similar to the client-side TEXT_IO package currently provided by Oracle Procedure Builder. Restrictions for a server implementation require some API differences between UTL_FILE and TEXT_IO. In PL/SQL file I/O, errors are returned to the caller using PL/SQL exceptions.

Server Security

Server security for PL/SQL file I/O consists of a restriction on the directories that can be accessed. Accessible directories must be specified in the instance parameter initialization file (INIT.ORA).

You specify the accessible directories for the UTL_FILE functions in the initialization file using the UTL_FILE_DIR parameter, as follows:

```
UTL_FILE_DIR = <directory name>
```

For example, if the initialization file for the instance contains the line

```
UTL_FILE_DIR = /usr/jsmith/my_app
```

then the directory `/usr/jsmith/my_app` is accessible to the FOPEN function. Note that a directory named `/usr/jsmith/My_App` would *not* be accessible on case-sensitive operating systems.

The parameter specification

```
UTL_FILE_DIR = *
```

has a special meaning. This entry in effect turns off directory access checking, and makes any directory accessible to the UTL_FILE functions.



Warning: The '*' option should be used with great caution. For obvious security reasons, Oracle does not recommend that you use this option in production systems. Also, do not include '.' (the current directory for UNIX) in the accessible directories list.



Warning: To ensure security on file systems that allow symbolic links, users must not be allowed WRITE permission to directories accessible by PL/SQL file I/O functions. The symbolic links and PL/SQL file I/O could be used to circumvent normal operating system permission checking, and allow users read/write access to directories to which they would not otherwise have access.

File Ownership and Protections

On UNIX systems, a file created by the FOPEN function has as its owner the owner of the shadow process running the instance. In the normal case, this owner is *oracle*. Files created using FOPEN are always writeable and readable using the UTL_FILE routines, but non-privileged users who need to read these files outside of PL/SQL might have to get their system administrator to give them access.

Examples (UNIX-Specific)

If the parameter initialization file contains only

```
UTL_FILE_DIR=/appl/gl/log
UTL_FILE_DIR=/appl/gl/out
```

then the following file locations and filenames are valid:

FILE LOCATION	FILENAME
/appl/gl/log	L10324.log
/appl/gl/out	O10324.out

but the following file locations and filename are *invalid*:

FILE LOCATION	FILENAME	
/appl/gl/log/backup	L10324.log	# subdirectory
/APPL/gl/log	L10324.log	# uppercase
/appl/gl/log	backup/L10324.log	#dir in name
/usr/tmp	T10324.tmp	# not in INIT.ORA

There are no user-level file permissions. All file locations specified by the UTL_FILE_DIR parameters are valid, for both reading and writing, for all users of the file I/O procedures. ***This can override operating system file permissions.***

Declared Types

The specification for the UTL_FILE package declares one PL/SQL type: FILE_TYPE. The declaration is

```
TYPE file_type IS RECORD (id BINARY_INTEGER);
```

The contents of FILE_TYPE are private to the UTL_FILE package. Users of the package should not reference or change components of this record.

Exceptions

The specification for the UTL_FILE package declares seven exceptions. These exceptions are raised to indicate error conditions. The exceptions are shown in Table 8 – 14.

Exception Name	Description
INVALID_PATH	File location or filename was invalid.
INVALID_MODE	The <code>open_mode</code> parameter in FOPEN was invalid.
INVALID_FILEHANDLE	The file handle was invalid.
INVALID_OPERATION	The file could not be opened or operated on as requested.
READ_ERROR	An operating system error occurred during the read operation.
WRITE_ERROR	An operating system error occurred during the write operation.
INTERNAL_ERROR	An unspecified error in PL/SQL.

Table 8 – 14 UTL_FILE Package Exceptions

In addition to these package exceptions, procedures in the UTL_FILE package can also raise predefined PL/SQL exceptions such as NO_DATA_FOUND or VALUE_ERROR.

Functions and Procedures

The remainder of this section describes the individual functions and procedures that make up the UTL_FILE package.

FOPEN

FOPEN opens a file for input or output. The file *location* must be an accessible directory, as defined in the instance's initialization parameter UTL_FILE_DIR. The complete directory path must already exist; it is not created by FOPEN. FOPEN returns a file handle, which must be used in all subsequent I/O operations on the file.

The parameters for this procedure are described in Table 8 – 15, and the syntax is shown below.

Syntax

```

FUNCTION FOPEN(location IN VARCHAR2,
               filename IN VARCHAR2,
               open_mode IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;

```

Parameters	Description
<code>location</code>	The operating system–specific string that specifies the directory or area in which to open the file.
<code>filename</code>	The name of the file, including extension (file type), without any directory path information. (Under the UNIX operating system, the filename cannot be terminated with a '/'.)
<code>open_mode</code>	A string that specifies how the file is to be opened (either upper- or lowercase letters can be used). The supported values, and the UTL_FILE package procedures that can be used with them are: <div style="margin-left: 20px;"> '<code>r</code>' read text <code>(GET_LINE)</code> '<code>w</code>' write text <code>(PUT, PUT_LINE, NEW_LINE, PUTF,</code> <code>FFLUSH)</code> '<code>a</code>' append text <code>(PUT, PUT_LINE, NEW_LINE, PUTF,</code> <code>FFLUSH)</code> </div>

Table 8 – 15 FOPEN Function Parameters


Note: If you open a file that does not exist using the 'a' value for OPEN_MODE, the file is created in write ('w') mode.

Return Value

FOPEN returns a file handle, which must be passed to all subsequent procedures that operate on that file. The specific contents of the file handle are private to the UTL_FILE package, and individual components should not be referenced or changed by the UTL_FILE user.

Notes

The *file location* and *file name* parameters are supplied to the FOPEN function as separate strings, so that the file location can be checked against the list of accessible directories as specified in the initialization file. Together, the file location and name must represent a legal filename on the system, and the directory must be accessible. A subdirectory of an accessible directory is not necessarily also accessible; it too must be specified using a complete path name in the initialization file.

 **Attention:** Operating system-specific parameters, such as C-shell environment variables under UNIX, cannot be used in the file location or file name parameters.

Exceptions

FOPEN can raise any of the following exceptions:

- UTL_FILE.INVALID_PATH
- UTL_FILE.INVALID_MODE
- UTL_FILE.INVALID_OPERATION

IS_OPEN

IS_OPEN tests a file handle to see if it identifies an open file. IS_OPEN reports only whether a file handle represents a file that has been opened, but not yet closed. It does not guarantee that there will be no operating system errors when you attempt to use the file handle.

The parameter for this function is described in Table 8 – 16, and the syntax is shown below.

Syntax

```
FUNCTION IS_OPEN(file_handle IN FILE_TYPE)
RETURN BOOLEAN;
```

Parameter	Description
file_handle	An active file handle returned by an FOPEN call.

Table 8 – 16 IS_OPEN Function Parameters

Return Value

TRUE or FALSE.

Exceptions

IS_OPEN does not raise any exceptions.

FCLOSE

FCLOSE closes an open file identified by a file handle. You could receive a WRITE_ERROR exception when closing a file, as there might be buffered data yet to be written when FCLOSE executes.

The parameters for this procedure are described in Table 8 – 17, and the syntax is shown below.

Syntax

```
PROCEDURE FCLOSE (file_handle IN OUT FILE_TYPE);
```

Parameter	Description
file_handle	An active file handle returned by an FOPEN call.

Table 8 – 17 FCLOSE Procedure Parameters


Exceptions

FCLOSE can raise the following exceptions:

- UTL_FILE.WRITE_ERROR
- UTL_FILE.INVALID_FILEHANDLE

FCLOSE_ALL

FCLOSE_ALL closes all open file handles for the session. This can be used as an emergency cleanup procedure, for example when a PL/SQL program exits on an exception.

 **Attention:** FCLOSE_ALL does not alter the state of the open file handles held by the user. This means that an IS_OPEN test on a file handle after an FCLOSE_ALL call still returns TRUE, even though the file has been closed. No further read or write operations can be performed on a file that was open before an FCLOSE_ALL.

Syntax

```
PROCEDURE FCLOSE_ALL;
```

Exception

FCLOSE_ALL can raise the exception:

- UTL_FILE.WRITE_ERROR

GET_LINE

GET_LINE reads a line of text from the open file identified by the file handle, and places the text in the output buffer parameter. Text is read up to but not including the line terminator, or up to the end of the file.

If the line does not fit in the buffer, a VALUE_ERROR exception is raised. If no text was read due to "end of file," the NO_DATA_FOUND exception is raised.

Because the line terminator character is not read into the buffer, reading blank lines returns empty strings.

The maximum size of an input record is 1022 bytes.

The parameters for this procedure are described in Table 8 – 18, and the syntax is shown below.

Syntax

```
PROCEDURE GET_LINE(file_handle      IN  FILE_TYPE,
                   buffer            OUT VARCHAR2);
```

Parameters	Description
file_handle	An active file handle returned by an FOPEN call. The file must be open for reading (mode 'r'), otherwise an INVALID_OPERATION exception is raised.
buffer	The data buffer to receive the line read from the file.

Table 8 – 18 GET_LINE Procedure Parameters

Exceptions

GET_LINE can raise any of the following exceptions:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.READ_ERROR
- NO_DATA_FOUND
- VALUE_ERROR

PUT

PUT writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. No line terminator is appended by PUT; use NEW_LINE to terminate the line or use PUT_LINE to write a complete line with a line terminator.

The parameters for this procedure are described in Table 8 – 19, and the syntax is shown below.

Syntax

```
PROCEDURE PUT(file_handle IN FILE_TYPE,
              buffer       IN VARCHAR2);
```

Parameters	Description
file_handle	An active file handle returned by an FOPEN call.
buffer	The buffer that contains the text to be written to the file. You must have opened the file using mode 'w' or mode 'a', otherwise an INVALID_OPERATION exception is raised.

Table 8 – 19 PUT Procedure Parameters

Exceptions

PUT can raise any of the following exceptions:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

NEW_LINE

NEW_LINE writes one or more line terminators to the file identified by the input file handle. This procedure is separate from PUT because the line terminator is a platform-specific character or sequence of characters.

The parameters for this procedure are described in Table 8 – 20, and the syntax is shown below.

Syntax

```
PROCEDURE NEW_LINE (file_handle IN FILE_TYPE,  
                    lines        IN NATURAL := 1);
```

Parameters	Description
file_handle	An active file handle returned by an FOPEN call.
lines	The number of line terminators to be written to the file.

Table 8 – 20 NEW_LINE Procedure Parameters

Exceptions

NEW_LINE can raise any of the following exceptions:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

PUT_LINE

PUT_LINE writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. PUT_LINE terminates the line with the platform-specific line terminator character or characters.

The maximum size for an output record is 1023 bytes.

The parameters for this procedure are described in Table 8 – 21, and the syntax is shown below.

Syntax

```
PROCEDURE PUT_LINE(file_handle  IN FILE_TYPE,
                   buffer        IN VARCHAR2);
```

Parameters	Description
file_handle	An active file handle returned by an FOPEN call.
buffer	The text buffer that contains the lines to be written to the file.

Table 8 – 21 PUT_LINE Procedure Parameters

Exceptions

PUT_LINE can raise any of the following exceptions:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

PUTF

PUTF is a formatted PUT procedure. It works like a limited *printf()*. The format string can contain any text, but the character sequences '%s' and '\n' have special meaning:

- %s Substitute this sequence with the string value of the next argument in the argument list (see the “Syntax” section below).
- \n Substitute with the appropriate platform-specific line terminator.

The parameters for this procedure are described in Table 8 – 22, and the syntax is shown below.

Syntax

```
PROCEDURE PUTF(file_handle IN FILE_TYPE,
               format      IN VARCHAR2,
               [arg1       IN VARCHAR2,
               ...arg5     IN VARCHAR2]);
```

Parameters	Description
file_handle	An active file handle returned by an FOPEN call.
format	The format string that can contain text as well as the format- ting characters '\n' and '%s'.
arg1..arg5	From one to five optional argument strings. Argument strings are substituted, in order, for the '%s' formatters in the format string. If there are more formatters in the format parameter string than there are arguments, an empty string is substituted for each '%s' for which there is no argument.

Table 8 – 22 PUTF Procedure Parameters

Example

The following example writes the lines

```
Hello, world!
I come from Zork with greetings for all earthlings.

my_world varchar2(4) := 'Zork';
...
PUTF(my_handle, 'Hello, world!\nI come from %s with %s.\n',
      my_world,
      'greetings for all earthlings');
```

If there are more %s formatters in the format parameter than there are arguments, an empty string is substituted for each %s for which there is no matching argument.

Exceptions

PUTF can raise any of the following exceptions:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

FFLUSH

FFLUSH physically writes all pending data to the file identified by the file handle. Normally, data being written to a file is buffered. The FFLUSH procedure forces any buffered data to be written to the file.

Flushing is useful when the file must be read while still open. For example, debugging messages can be flushed to the file so that they can be read immediately.

The parameter for this procedure is described in Table 8 – 23, and the syntax is shown below.

Syntax

```
PROCEDURE FFLUSH (file_handle IN FILE_TYPE);
```

Parameters	Description
file_handle	An active file handle returned by an FOPEN call.

Table 8 – 23 FFLUSH Procedure Parameters

Exceptions

FFLUSH can raise any of the following exceptions:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

Using Database Triggers

This chapter discusses database triggers—procedures that are stored in the database and implicitly executed (“fired”) when a table is modified. Topics include

- creating triggers
- altering triggers
- debugging triggers
- enabling and disabling triggers
- sample trigger applications

If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for more information about defining and using database triggers.

Designing Triggers

Use the following guidelines when designing triggers:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Use database triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
- Do not define triggers that duplicate the functionality already built into Oracle. For example, do not define triggers to enforce data integrity rules that can be easily enforced using declarative integrity constraints.
- Limit the size of triggers (60 lines or fewer is a good guideline). If the logic for your trigger requires much more than 60 lines of PL/SQL code, it is better to include most of the code in a stored procedure, and call the procedure from the trigger.
- **Be careful not to create recursive triggers.** For example, creating an AFTER UPDATE statement trigger on the EMP table that itself issues an UPDATE statement on EMP causes the trigger to fire recursively until it has run out of memory.

Creating Triggers

Triggers are created using the CREATE TRIGGER command. This command can be used with any interactive tool, such as SQL*Plus or Server Manager. When using an interactive tool, a solitary slash (/) on the last line is used to activate the CREATE TRIGGER statement. The following statement creates a trigger for the EMP table:

```
CREATE TRIGGER print_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
WHEN (new.empno > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff := new.sal - old.sal;
    dbms_output.put('Old salary: ' || :old.sal);
    dbms_output.put(' New salary: ' || :new.sal);
    dbms_output.put_line(' Difference ' || sal_diff);
END;
/
```

If you enter a SQL statement such as

```
UPDATE emp SET sal = sal + 500.00 WHERE deptno = 10
```

the trigger will fire once for each row that is updated, and it prints the new and old salaries, and the difference.

The CREATE (or CREATE OR REPLACE) statement fails if any errors exist in the PL/SQL block.

The following sections use this example to illustrate the way that parts of a trigger are specified. For more realistic examples of CREATE TRIGGER statements, see “Examples of Trigger Applications” on page 9 – 21.

Prerequisites

Before creating any triggers, while connected as SYS, submit the CATPROC.SQL script. This script automatically runs all of the scripts required for, or used within, the procedural extensions to the Oracle Server.



OSDoc

Additional Information: The location of this file is operating system dependent; see your platform-specific Oracle documentation.

Naming Triggers

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not have to be unique with respect to other schema objects such as tables, views, and procedures. For example, a table and a trigger can have the same name (although, to avoid confusion, this is not recommended).

The BEFORE/AFTER Options

Either the BEFORE or AFTER option must be used in the CREATE TRIGGER statement to specify exactly when the trigger body is fired in relation to the triggering statement being executed. In a CREATE TRIGGER statement, the BEFORE or AFTER option is specified just before the triggering statement. For example, the PRINT_SALARY_CHANGES trigger in the previous example is a BEFORE trigger.

Note: AFTER row triggers are slightly more efficient than BEFORE row triggers. With BEFORE row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement. Alternatively, with AFTER row triggers, the data blocks need only be read once for both the triggering statement and the trigger.

Triggering Statement

The triggering statement specifies

- the type of SQL statement that fires the trigger body. The possible options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
- the table associated with the trigger. Note that exactly one table (but not a view) can be specified in the triggering statement.

For example, the PRINT_SALARY_CHANGES trigger on page 9 – 3 fires after any DELETE, INSERT, or UPDATE on the EMP table. Any of the following statements would trigger the PRINT_SALARY_CHANGES trigger given in the previous example:

```
DELETE FROM emp;  
INSERT INTO emp VALUES ( . . . );  
INSERT INTO emp SELECT . . . FROM . . . ;  
UPDATE emp SET . . . ;
```

Column List for UPDATE

If a triggering statement specifies UPDATE, an optional list of columns can be included in the triggering statement. If you include a column list, the trigger is fired on an UPDATE statement only when one of the specified columns is updated. If you omit a column list, the trigger is fired when any column of the associated table is updated. A column list cannot be specified for INSERT or DELETE triggering statements.

The previous example of the PRINT_SALARY_CHANGES trigger might have included a column list in the triggering statement, as in

```
. . . BEFORE DELETE OR INSERT OR UPDATE OF ename ON emp . . .
```

FOR EACH ROW Option

The FOR EACH ROW option determines whether the trigger is a *row trigger* or a *statement trigger*. If you specify FOR EACH ROW, the trigger fires once for each row of the table that is affected by the triggering statement. The absence of the FOR EACH ROW option means that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

For example, you define the following trigger:

```
CREATE TRIGGER log_salary_increase
AFTER UPDATE ON emp
FOR EACH ROW
WHEN (new.sal > 1000)
BEGIN
    INSERT INTO emp_log (emp_id, log_date, new_salary, action)
        VALUES (:new.empno, SYSDATE, :new.sal, 'NEW SAL');
END;
```

and then issue the SQL statement

```
UPDATE emp SET sal = sal + 1000.0
WHERE deptno = 20;
```

If there are five employees in department 20, the trigger will fire five times when this statement is issued, since five rows are affected.

The following trigger fires only once for each UPDATE of the EMP table:

```
CREATE TRIGGER log_emp_update
AFTER UPDATE ON emp
BEGIN
    INSERT INTO emp_log (log_date, action)
        VALUES (SYSDATE, 'EMP COMMISSIONS CHANGED');
END;
```

For the order of trigger firing, see the *Oracle7 Server Concepts* manual.

The WHEN Clause

Optionally, a trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause (a WHEN clause cannot be included in the definition of a statement trigger). If included, the expression in the WHEN clause is evaluated for each row that the trigger affects. If the expression evaluates to TRUE for a row, the trigger body is fired on behalf of that row. However, if the expression evaluates to FALSE or NOT TRUE (that is, unknown, as with nulls) for a row, the trigger body is not fired for that row. The evaluation of the WHEN clause does not have an effect on the execution of the triggering SQL statement (that is, the triggering statement is **not** rolled back if the expression in a WHEN clause evaluates to FALSE).

For example, in the PRINT_SALARY_CHANGES trigger, the trigger body would not be executed if the new value of EMPNO is zero, NULL, or negative. In more realistic examples, you might test if one column value is less than another.

The expression in a WHEN clause of a row trigger can include correlation names, which are explained below. The expression in a WHEN clause must be a SQL expression and cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in the WHEN clause.

The Trigger Body

The trigger body is a PL/SQL block that can include SQL and PL/SQL statements. These statements are executed if the triggering statement is issued and the trigger restriction (if included) evaluates to TRUE. The trigger body for row triggers has some special constructs that can be included in the code of the PL/SQL block: correlation names and the REFERENCING option, and the conditional predicates INSERTING, DELETING, and UPDATING.

Accessing Column Values in Row Triggers

Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement. Two *correlation names* exist for every column of the table being modified: one for the old column value and one for the new column value. Depending on the type of triggering statement, certain correlation names might not have any meaning.

- A trigger fired by an INSERT statement has meaningful access to new column values only. Because the row is being created by the INSERT, the old values are null.
- A trigger fired by an UPDATE statement has access to both old and new column values for both BEFORE and AFTER row triggers.
- A trigger fired by a DELETE statement has meaningful access to old column values only. Because the row will no longer exist after the row is deleted, the new values are null.

The new column values are referenced using the NEW qualifier before the column name, while the old column values are referenced using the OLD qualifier before the column name. For example, if the triggering statement is associated with the EMP table (with the columns SAL, COMM, etc.), you can include statements in the trigger body similar to

```
IF :new.sal > 10000 . . .  
IF :new.sal < :old.sal . . .
```

Old and new values are available in both BEFORE and AFTER row triggers. A NEW column value can be assigned in a BEFORE row trigger, but not in an AFTER row trigger (because the triggering statement takes effect before an AFTER row trigger is fired). If a BEFORE row trigger changes the value of NEW.COLUMN, an AFTER row trigger fired by the same statement sees the change assigned by the BEFORE row trigger.

Correlation names can also be used in the Boolean expression of a WHEN clause. A colon must precede the OLD and NEW qualifiers when they are used in a trigger's body, but a colon is not allowed when using the qualifiers in the WHEN clause or the REFERENCING option.

The REFERENCING Option

The REFERENCING option can be specified in a trigger body of a row trigger to avoid name conflicts among the correlation names and tables that might be named "OLD" or "NEW". Since this is rare, this option is infrequently used.

For example, assume you have a table named NEW with columns FIELD1 (number) and FIELD2 (character). The following CREATE TRIGGER example shows a trigger associated with the NEW table that can use correlation names and avoid naming conflicts between the correlation names and the table name:

```
CREATE TRIGGER PRINT_SALARY_CHANGES
BEFORE UPDATE ON new
REFERENCING new AS newest
FOR EACH ROW
BEGIN
    :newest.field2 := TO_CHAR (:newest.field1);
END;
```

Notice that the NEW qualifier is renamed to NEWEST using the REFERENCING option, and is then used in the trigger body.

Conditional Predicates

If more than one type of DML operation can fire a trigger (for example, "ON INSERT OR DELETE OR UPDATE OF emp"), the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to execute specific blocks of code, depending on the type of statement that fires the trigger. Assume this is the triggering statement:

```
INSERT OR UPDATE ON emp
```

Within the code of the trigger body, you can include the following conditions:

```
IF INSERTING THEN . . . END IF;
IF UPDATING THEN . . . END IF;
```

The first condition evaluates to TRUE only if the statement that fired the trigger is an INSERT statement; the second condition evaluates to TRUE only if the statement that fired the trigger is an UPDATE statement.

In an UPDATE trigger, a column name can be specified with an UPDATING conditional predicate to determine if the named column is being updated. For example, assume a trigger is defined as

```
CREATE TRIGGER . . .
. . . UPDATE OF sal, comm ON emp . . .
BEGIN
. . . IF UPDATING ('SAL') THEN . . . END IF;
END;
```

The code in the THEN clause executes only if the triggering UPDATE statement updates the SAL column. The following statement would fire the above trigger and cause the UPDATING (sal) conditional predicate to evaluate to TRUE:

```
UPDATE emp SET sal = sal + 100;
```

Error Conditions and Exceptions in the Trigger Body

If a predefined or user-defined error condition or exception is raised during the execution of a trigger body, all effects of the trigger body, as well as the triggering statement, are rolled back (unless the error is trapped by an exception handler). Therefore, a trigger body can prevent the execution of the triggering statement by raising an exception. User-defined exceptions are commonly used in triggers that enforce complex security authorizations or integrity constraints.

For more information about error processing in PL/SQL program units, see “Handling Errors” and “Declaring Exceptions and Exception Handling Routines” on page 7 – 31.

Triggers and Handling Remote Exceptions

A trigger that accesses a remote site cannot do remote exception handling if the network link is unavailable. For example:

```
CREATE TRIGGER example
AFTER INSERT ON emp
FOR EACH ROW
BEGIN
    INSERT INTO emp@remote          -- <- compilation fails here
    VALUES ('x');                 -- when dblink is inaccessible
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO emp_log
        VALUES ('x');
END;
```

A trigger is compiled when it is created. Thus, if a remote site is unavailable when the trigger must compile, Oracle cannot validate the statement accessing the remote database, and the compilation fails. The previous example exception statement cannot execute because the trigger does not complete compilation.

Because stored procedures are stored in a compiled form, the work-around for the above example is as follows:

```
CREATE TRIGGER example
AFTER INSERT ON emp
FOR EACH ROW
BEGIN
    insert_row_proc;
END;

CREATE PROCEDURE insert_row_proc
BEGIN
    INSERT INTO emp@remote
    VALUES ('x');
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO emp_log
        VALUES ('x');
END;
```

The trigger in this example compiles successfully and calls the stored procedure, which already has a validated statement for accessing the remote database; thus, when the remote INSERT statement fails because the link is down, the exception is caught.

Restrictions on Creating Triggers

Coding a trigger requires some restrictions that are not required for standard PL/SQL blocks. The following sections discuss these restrictions.

Valid SQL Statements in Trigger Bodies

The body of a trigger can contain DML SQL statements. It can also contain SELECT statements, but they must be SELECT ... INTO ... statements or the SELECT statement in the definition of a cursor).

DDL statements are not allowed in the body of a trigger. Also, no transaction control statements are allowed in a trigger. The commands ROLLBACK, COMMIT, and SAVEPOINT cannot be used.

Note: A procedure called by a trigger cannot execute the above transaction control statements because the procedure executes within the context of the trigger body.

Statements inside a trigger can reference remote objects. However, pay special attention when calling remote procedures from within a local trigger; since if a timestamp or signature mismatch is found during execution of the trigger, the remote procedure is not executed and the trigger is invalidated.

LONG and LONG RAW Datatypes

LONG and LONG RAW datatypes in triggers are subject to the following restrictions:

- A SQL statement within a trigger can insert data into a column of LONG or LONG RAW datatype.
- If data from a LONG or LONG RAW column can be converted to a constrained datatype (such as CHAR and VARCHAR2), a LONG or LONG RAW column can be referenced in a SQL statement within a trigger. Note that the maximum length for these datatypes is 32000 bytes.
- Variables cannot be declared using the LONG or LONG RAW datatypes.
- :NEW and :OLD cannot be used with LONG or LONG RAW columns.

References to Package Variables

If an UPDATE or DELETE statement detects a conflict with a concurrent UPDATE, Oracle performs a transparent rollback to savepoint and restarts the update. This can occur many times before the statement completes successfully. Each time the statement is restarted, the BEFORE STATEMENT trigger is fired again. The rollback to savepoint does not undo changes to any package variables referenced in the trigger. The package should include a counter variable to detect this situation.

Row Evaluation Order

A relational database does not guarantee the order of rows processed by a SQL statement. Therefore, do not create triggers that depend on the order in which rows will be processed. For example, do not assign a value to a global package variable in a row trigger if the current value of the global variable is dependent on the row being processed by the row trigger. Also, if global package variables are updated within a

trigger, it is best to initialize those variables in a BEFORE statement trigger.

When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading*. Oracle allows up to 32 triggers to cascade at any one time. However, you can effectively limit the number of trigger cascades using the initialization parameter OPEN_CURSORS, because a cursor must be opened for every execution of a trigger.

Trigger Evaluation Order

Although any trigger can execute a sequence of operations either in-line or by calling procedures, using multiple triggers of the same type enhances database administration by permitting the modular installation of applications that have triggers on the same tables.

Oracle executes all triggers of the same type before executing triggers of a different type. If you have multiple triggers of the same type on a single table, Oracle chooses an arbitrary order to execute these triggers. See the *Oracle7 Server Concepts* manual for more information on the firing order of triggers.

Each subsequent trigger sees the changes made by the previously fired triggers. Each trigger can see the old and new values. The old values are the original values and the new values are the current values as set by the most recently fired UPDATE or INSERT trigger.

To ensure that multiple triggered actions occur in a specific order, you must consolidate these actions into a single trigger (for example, by having the trigger call a series of procedures).

You cannot open a database that contains multiple triggers of the same type if you are using any version of Oracle before release 7.1, nor can you open such a database if your COMPATIBLE initialization parameter is set to a version earlier than 7.1.0.

Mutating and Constraining Tables

A *mutating table* is a table that is currently being modified by an UPDATE, DELETE, or INSERT statement, or a table that might need to be updated by the effects of a declarative DELETE CASCADE referential integrity constraint. A *constraining table* is a table that a triggering statement might need to read either directly, for a SQL statement, or indirectly, for a declarative referential integrity constraint. A table is mutating or constraining only to the session that issued the statement in progress.

Tables are never considered mutating or constraining *for statement triggers* unless the trigger is fired as the result of a DELETE CASCADE.

For all row triggers, or for statement triggers that were fired as the result of a DELETE CASCADE, there are two important restrictions regarding mutating and constraining tables. These restrictions prevent a trigger from seeing an inconsistent set of data.

- The SQL statements of a trigger cannot read from (query) or modify a mutating table of the triggering statement.
- The statements of a trigger cannot change the PRIMARY, FOREIGN, or UNIQUE KEY columns of a constraining table of the triggering statement.

There is an exception to this restriction; BEFORE ROW and AFTER ROW triggers fired by a single row INSERT to a table do not treat that table as mutating or constraining. Note that INSERT statements that may involve more than one row, such as INSERT INTO emp SELECT . . . , are not considered single row inserts, even if they only result in one row being inserted.

Figure 9 – 1 illustrates the restriction placed on mutating tables.

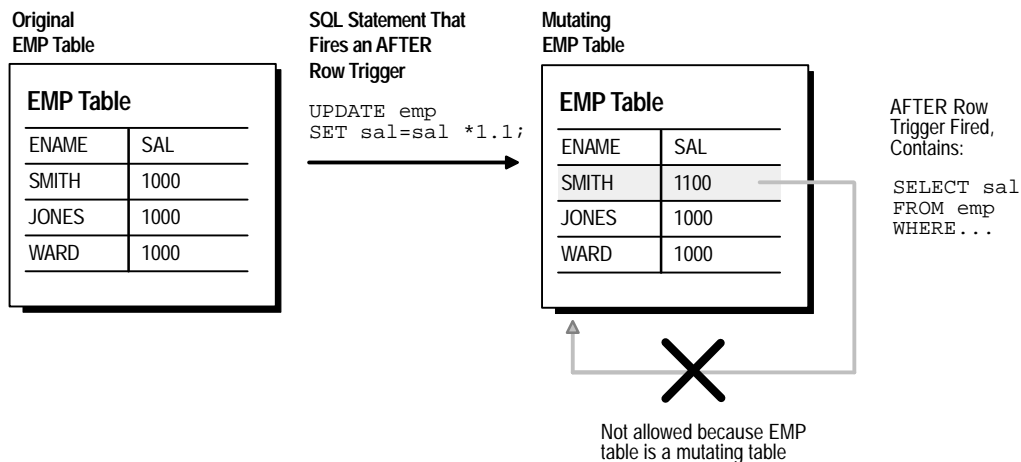


Figure 9 – 1 Mutating Tables

Notice that the SQL statement is executed for the first row of the table and then an AFTER ROW trigger is fired. In turn, a statement in the AFTER ROW trigger body attempts to query the original table. However, because the EMP table is mutating, this query is not allowed by Oracle. If attempted, a runtime error occurs, the effects of the trigger body and triggering statement are rolled back, and control is returned to the user or application.

Consider the following trigger:

```
CREATE OR REPLACE TRIGGER emp_count
AFTER DELETE ON EMP
FOR EACH ROW
DECLARE
    n INTEGER;
BEGIN
    SELECT COUNT(*) INTO n FROM emp;
    DBMS_OUTPUT.PUT_LINE(' There are now ' || n ||
        ' employees.');
```

END;

If the SQL statement

```
DELETE FROM emp WHERE empno = 7499;
```

is issued, the following error is returned:

```
ORA-04091: table SCOTT.EMP is mutating, trigger/function may not
see it
```

Oracle returns this error when the trigger fires since the table is mutating when the first row is deleted. (Only one row is deleted by the statement, since EMPNO is a primary key, but Oracle has no way of knowing that.)

If you delete the line “FOR EACH ROW” from the trigger above, the trigger becomes a statement trigger, the table is not mutating when the trigger fires, and the trigger does output the correct data.

If you need to update a mutating or constraining table, you could use a temporary table, a PL/SQL table, or a package variable to bypass these restrictions. For example, in place of a single AFTER row trigger that updates the original table, resulting in a mutating table error, you may be able to use two triggers—an AFTER row trigger that updates a temporary table, and an AFTER statement trigger that updates the original table with the values from the temporary table.

Declarative integrity constraints are checked at various times with respect to row triggers; see the *Oracle7 Server Concepts* manual for information about the interaction of triggers and integrity constraints.

Because declarative referential integrity constraints are currently not supported between tables on different nodes of a distributed database, the constraining table restrictions do not apply to triggers that access remote nodes. These restrictions are also not enforced among tables in the same database that are connected by loop-back database links. A loop-back database link makes a local table appear remote by defining a SQL*Net path back to the database that contains the link.

You should not use loop-back database links to circumvent the trigger restrictions. Such applications might behave unpredictably. Trigger restrictions, which were implemented to prevent row triggers from seeing an inconsistent set of data, might be enforced on loop-back database links in future releases of Oracle.

Who is the Trigger User?

If you issue the statement

```
SELECT username FROM USER_USERS
```

in a trigger, the name of the owner of the trigger is returned, not the name of user who is updating the table.

Privileges Required to Create Triggers

To create a trigger in your schema, you must have the CREATE TRIGGER system privilege, and either

- own the table specified in the triggering statement, or
- have the ALTER privilege for the table in the triggering statement, or
- have the ALTER ANY TABLE system privilege

To create a trigger in another user's schema, you must have the CREATE ANY TRIGGER system privilege. With this privilege, the trigger can be created in any schema and can be associated with any user's table.

Privileges for Referenced Schema Objects

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger's owner explicitly (not via a role). The statements in the trigger body operate under the privilege domain of the trigger's owner, not the privilege domain of the user issuing the triggering statement. This is similar to stored procedures; see page 7 – 39 for details.

When Triggers Are Compiled

In previous releases of the Oracle7 Server, triggers were basically like PL/SQL anonymous blocks, with the addition of the :NEW and :OLD capabilities. A PL/SQL anonymous block is compiled each time it is loaded into memory. Compilation involves three stages:

1. syntax checking: PL/SQL syntax is checked and a parse tree is generated
2. semantic checking: type checking and further processing on the parse tree
3. code generation: the pcode is generated

The CREATE TRIGGER statement creates a trigger, and stores its source in the data dictionary. In releases prior to 7.3, steps 1 and 2 are performed at CREATE TRIGGER time, so that syntactic and semantic errors can be reported back to the user who issued the CREATE TRIGGER command. However, the compiler output is discarded. At execution time, the source for the trigger is parsed and executed just like an anonymous PL/SQL block.

In Oracle7 release 7.3, triggers are fully compiled when the CREATE TRIGGER command is issued, and the pcode is stored in the data dictionary. Hence, firing the trigger no longer requires the opening of a shared cursor to run the trigger action. Instead, the trigger is executed directly.

If errors occur during the compilation of a trigger, the trigger is still created. If a DML statement fires this trigger, the DML statement will fail. (In both release 7.3 and releases 7.2 and earlier, runtime trigger errors always cause the DML statement to fail.) You can use the SHOW ERRORS command in SQL*Plus or Server Manager to see any compilation errors when you create a trigger, or you can SELECT the errors from the USER_ERRORS view.

Dependencies

Compiled triggers have dependencies. They become invalid if a depended-on object, such as a stored procedure or a function called from the trigger body, is modified. Triggers that are invalidated for dependency reasons are recompiled when next invoked.

You can examine the ALL_DEPENDENCIES view to see the dependencies for a trigger. For example, the statement

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
FROM ALL_DEPENDENCIES
WHERE OWNER = 'SCOTT' and TYPE = 'TRIGGER';
```

shows the dependencies for the triggers in the SCOTT schema.

Recompiling a Trigger

Use the ALTER TRIGGER command to recompile a trigger manually. For example, the command

```
ALTER TRIGGER print_salary_changes COMPILE;
```

recompiles the PRINT_SALARY_CHANGES trigger

Migration Issues

Non-compiled triggers cannot be fired under compiled trigger releases (such as 7.3). If upgrading from a non-compiled trigger release to a compiled trigger release, all existing triggers must be compiled. The upgrade script *cat73xx.sql* invalidates all triggers so that they are automatically recompiled when first executed under the Oracle release 7.3 server. (The *xx* stands for a variable minor release number.)

Downgrading from 7.3 or later to a release prior to 7.3 requires that you execute the *cat73xxd.sql* downgrade script. This handles portability issues between stored and non-stored trigger releases.

Debugging a Trigger

You can debug a trigger using the same facilities available for stored procedures. See the section “Debugging” on page 7 – 36.

Modifying a Trigger

Like a stored procedure, a trigger cannot be explicitly altered; it must be replaced with a new definition. (The ALTER TRIGGER command is used only to recompile, enable or disable a trigger.).

When replacing a trigger, you must include the OR REPLACE option in the CREATE TRIGGER statement. The OR REPLACE option is provided to allow a new version of an existing trigger to replace the older version without affecting any grants made for the original version of the trigger.

Alternatively, the trigger can be dropped using the DROP TRIGGER command, and you can rerun the CREATE TRIGGER command.

To drop a trigger, the trigger must be in your schema or you must have the DROP ANY TRIGGER system privilege.

Enabling and Disabling Triggers

A trigger can be in one of two distinct modes:

enabled	An enabled trigger executes its trigger body if a triggering statement is issued and the trigger restriction (if any) evaluates to TRUE.
disabled	A disabled trigger does not execute its trigger body, even if a triggering statement is issued and the trigger restriction (if any) evaluates to TRUE.

Disabling Triggers

You might temporarily disable a trigger if

- an object it references is not available
- you have to perform a large data load and want it to proceed quickly without firing triggers
- you are reloading data

By default, triggers are enabled when first created. Disable a trigger using the ALTER TRIGGER command with the DISABLE option. For example, to disable the trigger named REORDER of the INVENTORY table, enter the following statement:

```
ALTER TRIGGER reorder DISABLE;
```

All triggers associated with a table can be disabled with one statement using the ALTER TABLE command with the DISABLE clause and the ALL TRIGGERS option. For example, to disable all triggers defined for the INVENTORY table, enter the following statement:

```
ALTER TABLE inventory
    DISABLE ALL TRIGGERS;
```

Enabling Triggers

By default, a trigger is automatically enabled when it is created; however, it can later be disabled. Once you have completed the task that required the trigger to be disabled, re-enable the trigger so it fires when appropriate.

Enable a disabled trigger using the ALTER TRIGGER command with the ENABLE option. To enable the disabled trigger named REORDER of the INVENTORY table, enter the following statement:

```
ALTER TRIGGER reorder ENABLE;
```

All triggers defined for a specific table can be enabled with one statement using the ALTER TABLE command with the ENABLE clause with the ALL TRIGGERS option. For example, to enable all triggers defined for the INVENTORY table, enter the following statement:

```
ALTER TABLE inventory
    ENABLE ALL TRIGGERS;
```

Privileges Required to Enable and Disable Triggers

To enable or disable triggers using the ALTER TABLE command, you must either own the table, have the ALTER object privilege for the table, or have the ALTER ANY TABLE system privilege. To enable or disable triggers using the ALTER TRIGGER command, you must own the trigger or have the ALTER ANY TRIGGER system privilege.

Listing Information About Triggers

The following data dictionary views reveal information about triggers:

- USER_TRIGGERS
- ALL_TRIGGERS
- DBA_TRIGGERS

The *Oracle7 Server Reference* manual gives a complete description of these data dictionary views. For example, assume the following statement was used to create the REORDER trigger:

```
CREATE TRIGGER reorder
AFTER UPDATE OF parts_on_hand ON inventory
FOR EACH ROW
WHEN(new.parts_on_hand < new.reorder_point)
DECLARE
    x NUMBER;
BEGIN
    SELECT COUNT(*) INTO x
    FROM pending_orders
    WHERE part_no = :new.part_no;
    IF x = 0 THEN
        INSERT INTO pending_orders
        VALUES (:new.part_no, :new.reorder_quantity,
            sysdate);
    END IF;
END;
```

The following two queries return information about the REORDER trigger:

```
SELECT trigger_type, triggering_event, table_name
FROM user_triggers
WHERE name = 'REORDER';
```

TYPE	TRIGGERING_STATEMENT	TABLE_NAME
-----	-----	-----
AFTER EACH ROW	UPDATE	INVENTORY

```
SELECT trigger_body
FROM user_triggers
WHERE name = 'REORDER';
```

```
TRIGGER_BODY
-----
DECLARE
    x NUMBER;
BEGIN
    SELECT COUNT(*) INTO x
```

```

        FROM pending_orders
        WHERE part_no = :new.part_no;
    IF x = 0
        THEN INSERT INTO pending_orders
            VALUES (:new.part_no, :new.reorder_quantity,
                sysdate);
    END IF;
END;
```

Examples of Trigger Applications

You can use triggers in a number of ways to customize information management in an Oracle database. For example, triggers are commonly used to

- provide sophisticated auditing
- prevent invalid transactions
- enforce referential integrity (either those actions not supported by declarative integrity constraints or across nodes in a distributed database)
- enforce complex business rules
- enforce complex security authorizations
- provide transparent event logging
- automatically generate derived column values

This section provides an example of each of the above trigger applications. These examples are not meant to be used as is, but are provided to assist you in designing your own triggers.

Auditing with Triggers

Triggers are commonly used to supplement the built-in auditing features of Oracle. Although triggers can be written to record information similar to that recorded by the AUDIT command, triggers should be used only when more detailed audit information is required. For example, use triggers to provide value-based auditing on a per-row basis for tables.

Sometimes, the Oracle AUDIT command is considered a *security* audit facility, while triggers can provide *financial* audit facility.

When deciding whether to create a trigger to audit database activity, consider what Oracle's auditing features provide, compared to auditing defined by triggers.

DML as well as DDL auditing	Standard auditing options permit auditing of DML and DDL statements regarding all types of schema objects and structures. Comparatively, triggers only permit auditing of DML statements issued against tables.
Centralized audit trail	All database audit information is recorded centrally and automatically using the auditing features of Oracle.
Declarative method	Auditing features enabled using the standard Oracle features are easier to declare and maintain, and less prone to errors when compared to auditing functions defined by triggers.
Auditing options can be audited	Any changes to existing auditing options can also be audited to guard against malicious database activity.
Session and execution time auditing	Using the database auditing features, records can be generated once every time an audited statement is issued (BY ACCESS) or once for every session that issues an audited statement (BY SESSION). Triggers cannot audit by session; an audit record is generated each time a trigger-audited table is referenced.
Auditing of unsuccessful data access	Database auditing can be set to audit when unsuccessful data access occurs. However, any audit information generated by a trigger is rolled back if the triggering statement is rolled back.
Sessions can be audited	Connections and disconnections, as well as session activity (physical I/Os, logical I/Os, deadlocks, etc.), can be recorded using standard database auditing.

When using triggers to provide sophisticated auditing, AFTER triggers are normally used. By using AFTER triggers, auditing information is recorded after the triggering statement is subjected to any applicable integrity constraints, preventing cases where the audit processing is carried out unnecessarily for statements that generate exceptions to integrity constraints.

When to use AFTER row vs. AFTER statement triggers depends on the information being audited. For example, row triggers provide value-based auditing on a per-row basis for tables. Triggers can also require the user to supply a “reason code” for issuing the audited SQL statement, which can be useful in both row and statement-level auditing situations.

The following example demonstrates a trigger that audits modifications to the EMP table on a per-row basis. It requires that a “reason code” be stored in a global package variable before the update.

Example This trigger demonstrates

- how triggers can be used to provide value-based auditing
- how to use public package variables

Comments within the code explain the functionality of the trigger.

```
CREATE TRIGGER audit_employee
AFTER INSERT OR DELETE OR UPDATE ON emp
FOR EACH ROW
BEGIN
/* AUDITPACKAGE is a package with a public package
   variable REASON. REASON could be set by the
   application by a command such as EXECUTE
   AUDITPACKAGE.SET_REASON(reason_string). Note that a
   package variable has state for the duration of a
   session and that each session has a separate copy of
   all package variables. */

IF auditpackage.reason IS NULL THEN
    raise_application_error(-20201, 'Must specify reason'
        || ' with AUDITPACKAGE.SET_REASON(reason_string)');
END IF;

/* If the above conditional evaluates to TRUE, the
   user-specified error number and message is raised,
   the trigger stops execution, and the effects of the
   triggering statement are rolled back. Otherwise, a
   new row is inserted into the predefined auditing
   table named AUDIT_EMPLOYEE containing the existing
   and new values of the EMP table and the reason code
   defined by the REASON variable of AUDITPACKAGE. Note
   that the "old" values are NULL if triggering
   statement is an INSERT and the "new" values are NULL
   if the triggering statement is a DELETE. */
```

```

INSERT INTO audit_employee VALUES
    (:old.ssn, :old.name, :old.job_classification, :old.sal,
     :new.ssn, :new.name, :new.job_classification, :new.sal,
     auditpackage.reason, user, sysdate );
END;

```

Optionally, you can also set the reason code back to NULL if you wanted to force the reason code to be set for every update. The following simple AFTER statement trigger sets the reason code back to NULL after the triggering statement is executed:

```

CREATE TRIGGER audit_employee_reset
AFTER INSERT OR DELETE OR UPDATE ON emp
BEGIN
    auditpackage.set_reason(NULL);
END;

```

Notice that the previous two triggers are both fired by the same type of SQL statement. However, the AFTER row trigger is fired once for each row of the table affected by the triggering statement, while the AFTER statement trigger is fired only once after the triggering statement execution is completed.

Another example of using triggers to do auditing is shown below. This trigger tracks changes being made to the EMP table, and stores this information in AUDIT_TABLE and AUDIT_TABLE_VALUES.

```

CREATE OR REPLACE TRIGGER audit_emp
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW
DECLARE
    time_now DATE;
    terminal CHAR(10);
BEGIN

    -- get current time, and the terminal of the user
    time_now := SYSDATE;
    terminal := USERENV('TERMINAL');

    -- record new employee primary key
    IF INSERTING THEN
        INSERT INTO audit_table
            VALUES (audit_seq.NEXTVAL, user, time_now,
                    terminal, 'EMP', 'INSERT', :new.empno);

    -- record primary key of the deleted row
    ELSIF DELETING THEN
        INSERT INTO audit_table
            VALUES (audit_seq.NEXTVAL, user, time_now,
                    terminal, 'EMP', 'DELETE', :old.empno);

```

```

-- for updates, record the primary key
-- of the row being updated
ELSE
    INSERT INTO audit_table
        VALUES (audit_seq.NEXTVAL, user, time_now,
            terminal, 'EMP', 'UPDATE', :old.empno);

-- and for SAL and DEPTNO, record old and new values
IF UPDATING ('SAL') THEN
    INSERT INTO audit_table_values
        VALUES (audit_seq.CURRVAL, 'SAL',
            :old.sal, :new.sal);

ELSIF UPDATING ('DEPTNO') THEN
    INSERT INTO audit_table_values
        VALUES (audit_seq.CURRVAL, 'DEPTNO',
            :old.deptno, :new.deptno);
END IF;
END IF;
END;
/

```

Integrity Constraints and Triggers

Triggers and declarative integrity constraints can both be used to constrain data input. However, triggers and integrity constraints have significant differences.

Declarative integrity constraints are statements about the database that are always true. A constraint applies to existing data in the table and any statement that manipulates the table; for more information, see Chapter 6.

Triggers constrain what a transaction can do. A trigger does not apply to data loaded before the definition of the trigger; therefore, it is not known if all data in a table conforms to the rules established by an associated trigger.

Although triggers can be written to enforce many of the same rules supported by Oracle's declarative integrity constraint features, triggers should only be used to enforce complex business rules that cannot be defined using standard integrity constraints. The declarative integrity constraint features provided with Oracle offer the following advantages when compared to constraints defined by triggers:

Centralized integrity checks	All points of data access must adhere to the global set of rules defined by the integrity constraints corresponding to each schema object.
Declarative method	Constraints defined using the standard integrity constraint features are much easier to write and are less prone to errors when compared with comparable constraints defined by triggers.

While most aspects of data integrity can be defined and enforced using declarative integrity constraints, triggers can be used to enforce complex business constraints not definable using declarative integrity constraints. For example, triggers can be used to enforce

- UPDATE and DELETE SET NULL, and UPDATE and DELETE SET DEFAULT referential actions
- referential integrity when the parent and child tables are on different nodes of a distributed database
- complex check constraints not definable using the expressions allowed in a CHECK constraint

Enforcing Referential Integrity Using Triggers

Many cases of referential integrity can be enforced using triggers. However, only use triggers when you want to enforce the UPDATE and DELETE SET NULL (when referenced data is updated or deleted, all associated dependent data is set to NULL), and UPDATE and DELETE SET DEFAULT (when referenced data is updated or deleted, all associated dependent data is set to a default value) referential actions, or when you want to enforce referential integrity between parent and child tables on different nodes of a distributed database.

When using triggers to maintain referential integrity, declare the PRIMARY (or UNIQUE) KEY constraint in the parent table. If referential integrity is being maintained between a parent and child table in the same database, you can also declare the foreign key in the child table, but disable it; this prevents the corresponding PRIMARY KEY constraint from being dropped (unless the PRIMARY KEY constraint is explicitly dropped with the CASCADE option).

To maintain referential integrity using triggers:

- A trigger must be defined for the child table that guarantees values inserted or updated in the foreign key correspond to values in the parent key.
- One or more triggers must be defined for the parent table. These triggers guarantee the desired referential action (RESTRICT, CASCADE, or SET NULL) for values in the foreign key when values are updated or deleted in the parent key. No action is required for inserts into the parent table (no dependent foreign keys exist).

The following sections provide examples of the triggers necessary to enforce referential integrity. The EMP and DEPT table relationship is used in these examples.

Several of the triggers include statements that lock rows (SELECT ... FOR UPDATE). This operation is necessary to maintain concurrency as the rows are being processed.

Foreign Key Trigger for Child Table The following trigger guarantees that before an INSERT or UPDATE statement affects a foreign key value, the corresponding value exists in the parent key. The mutating table exception included in the example below allows this trigger to be used with the UPDATE_SET_DEFAULT and UPDATE_CASCADE triggers. This exception can be removed if this trigger is used alone.

```
CREATE TRIGGER emp_dept_check
BEFORE INSERT OR UPDATE OF deptno ON emp
FOR EACH ROW WHEN (new.deptno IS NOT NULL)

-- Before a row is inserted, or DEPTNO is updated in the EMP
-- table, fire this trigger to verify that the new foreign
-- key value (DEPTNO) is present in the DEPT table.
DECLARE
    dummy INTEGER; -- used for cursor fetch below
    invalid_department EXCEPTION;
    valid_department EXCEPTION;
    mutating_table EXCEPTION;
    PRAGMA EXCEPTION_INIT (mutating_table, -4091);
-- Cursor used to verify parent key value exists. If
-- present, lock parent key's row so it can't be
-- deleted by another transaction until this
-- transaction is committed or rolled back.
CURSOR PRINT_SALARY_CHANGES_cursor (dn NUMBER) IS
    SELECT deptno
    FROM dept
    WHERE deptno = dn
    FOR UPDATE OF deptno;
```



```

BEGIN
    OPEN dummy_cursor (:new.deptno);
    FETCH dummy_cursor INTO dummy;

    -- Verify parent key.  If not found, raise user-specified
    -- error number and message.  If found, close cursor
    -- before allowing triggering statement to complete.
    IF dummy_cursor%NOTFOUND THEN
        RAISE invalid_department;

    ELSE
        RAISE valid_department;
    END IF;
    CLOSE dummy_cursor;
EXCEPTION
    WHEN invalid_department THEN
        CLOSE dummy_cursor;
        raise_application_error(-20000, 'Invalid Department'
            || ' Number' || TO_CHAR(:new.deptno));
    WHEN valid_department THEN
        CLOSE dummy_cursor;
    WHEN mutating_table THEN
        NULL;
END;

```

UPDATE and DELETE RESTRICT Trigger for the Parent Table The following trigger is defined on the DEPT table to enforce the UPDATE and DELETE RESTRICT referential action on the primary key of the DEPT table:

```

CREATE TRIGGER dept_restrict
BEFORE DELETE OR UPDATE OF deptno ON dept
FOR EACH ROW

-- Before a row is deleted from DEPT or the primary key
-- (DEPTNO) of DEPT is updated, check for dependent
-- foreign key values in EMP; rollback if any are found.
DECLARE
    dummy INTEGER;          -- used for cursor fetch below
    employees_present EXCEPTION;
    employees_not_present EXCEPTION;

-- Cursor used to check for dependent foreign key values.
CURSOR dummy_cursor (dn NUMBER) IS
    SELECT deptno FROM emp WHERE deptno = dn;

```

```

BEGIN
    OPEN dummy_cursor (:old.deptno);
    FETCH dummy_cursor INTO dummy;

    -- If dependent foreign key is found, raise user-specified
    -- error number and message.  If not found, close cursor
    -- before allowing triggering statement to complete.
    IF dummy_cursor%FOUND THEN
        RAISE employees_present; /* dependent rows exist */
    ELSE
        RAISE employees_not_present; /* no dependent rows */
    END IF;
    CLOSE dummy_cursor;

EXCEPTION
    WHEN employees_present THEN
        CLOSE dummy_cursor;
        raise_application_error(-20001, 'Employees Present in'
            || ' Department ' || TO_CHAR(:old.deptno));
    WHEN employees_not_present THEN
        CLOSE dummy_cursor;
END;

```

Note: This trigger will not work with self-referential tables (that is, tables with both the primary/unique key and the foreign key). Also, this trigger does not allow triggers to cycle (such as, A fires B fires A).

UPDATE and DELETE SET NULL Triggers for Parent Table The following trigger is defined on the DEPT table to enforce the UPDATE and DELETE SET NULL referential action on the primary key of the DEPT table:

```

CREATE TRIGGER dept_set_null
AFTER DELETE OR UPDATE OF deptno ON dept
FOR EACH ROW

-- Before a row is deleted from DEPT or the primary key
-- (DEPTNO) of DEPT is updated, set all corresponding
-- dependent foreign key values in EMP to NULL.
BEGIN
    IF UPDATING AND :OLD.deptno != :NEW.deptno OR DELETING THEN
        UPDATE emp SET emp.deptno = NULL
            WHERE emp.deptno = :old.deptno;
    END IF;
END;

```

DELETE Cascade Trigger for Parent Table The following trigger on the DEPT table enforces the DELETE CASCADE referential action on the primary key of the DEPT table:

```

CREATE TRIGGER dept_del_cascade
AFTER DELETE ON dept
FOR EACH ROW

-- Before a row is deleted from DEPT, delete all
-- rows from the EMP table whose DEPTNO is the same as
-- the DEPTNO being deleted from the DEPT table.
BEGIN
    DELETE FROM emp
        WHERE emp.deptno = :old.deptno;
END;

```

Note: Typically, the code for DELETE cascade is combined with the code for UPDATE SET NULL or UPDATE SET DEFAULT to account for both updates and deletes.

UPDATE Cascade Trigger for Parent Table The following trigger ensures that if a department number is updated in the DEPT table, this change is propagated to dependent foreign keys in the EMP table:

```

-- Generate a sequence number to be used as a flag for
-- determining if an update has occurred on a column.
CREATE SEQUENCE update_sequence
    INCREMENT BY 1 MAXVALUE 5000
    CYCLE;

CREATE PACKAGE integritypackage AS
    updateseq NUMBER;
END integritypackage;

CREATE or replace PACKAGE BODY integritypackage AS
END integritypackage;
ALTER TABLE emp ADD update_id NUMBER;    -- create flag col.

CREATE TRIGGER dept_cascadel BEFORE UPDATE OF deptno ON dept
DECLARE
    dummy NUMBER;

-- Before updating the DEPT table (this is a statement
-- trigger), generate a new sequence number and assign
-- it to the public variable UPDATESEQ of a user-defined
-- package named INTEGRITYPACKAGE.
BEGIN
    SELECT update_sequence.NEXTVAL
        INTO dummy
        FROM dual;
    integritypackage.updateseq := dummy;
END;

```

```

CREATE TRIGGER dept_cascade2 AFTER DELETE OR UPDATE
  OF deptno ON dept FOR EACH ROW

-- For each department number in DEPT that is updated,
-- cascade the update to dependent foreign keys in the
-- EMP table. Only cascade the update if the child row
-- has not already been updated by this trigger.
BEGIN
  IF UPDATING THEN
    UPDATE emp
      SET deptno = :new.deptno,
          update_id = integritypackage.updateseq /*from 1st*/
      WHERE emp.deptno = :old.deptno
      AND update_id IS NULL;
    /* only NULL if not updated by the 3rd trigger
       fired by this same triggering statement */
  END IF;
  IF DELETING THEN

    -- Before a row is deleted from DEPT, delete all
    -- rows from the EMP table whose DEPTNO is the same as
    -- the DEPTNO being deleted from the DEPT table.
    DELETE FROM emp
      WHERE emp.deptno = :old.deptno;
  END IF;
END;

CREATE TRIGGER dept_cascade3 AFTER UPDATE OF deptno ON dept
BEGIN  UPDATE emp
  SET update_id = NULL
  WHERE update_id = integritypackage.updateseq;
END;

```

Note: Because this trigger updates the EMP table, the EMP_DEPT_CHECK trigger, if enabled, is also fired. The resulting mutating table error is trapped by the EMP_DEPT_CHECK trigger. You should carefully test any triggers that require error trapping to succeed to ensure that they will always work properly in your environment.

Enforcing Complex Check Constraints

Triggers can enforce integrity rules other than referential integrity. For example, this trigger performs a complex check before allowing the triggering statement to execute. Comments within the code explain the functionality of the trigger.

```
CREATE TRIGGER salary_check
BEFORE INSERT OR UPDATE OF sal, job ON emp
FOR EACH ROW
DECLARE
    minsal          NUMBER;
    maxsal          NUMBER;
    salary_out_of_range  EXCEPTION;
BEGIN

    /* Retrieve the minimum and maximum salary for the
    employee's new job classification from the SALGRADE
    table into MINSAL and MAXSAL. */

    SELECT minsal, maxsal INTO minsal, maxsal FROM salgrade
        WHERE job_classification = :new.job;

    /* If the employee's new salary is less than or greater
    than the job classification's limits, the exception is
    raised. The exception message is returned and the
    pending INSERT or UPDATE statement that fired the
    trigger is rolled back. */

    IF (:new.sal < minsal OR :new.sal > maxsal) THEN
        RAISE salary_out_of_range;
    END IF;
EXCEPTION
    WHEN salary_out_of_range THEN
        raise_application_error (-20300,
            'Salary ' || TO_CHAR(:new.sal) || ' out of range for '
            || 'job classification ' || :new.job
            || ' for employee ' || :new.name);
    WHEN NO_DATA_FOUND THEN
        raise_application_error (-20322,
            'Invalid Job Classification '
            || :new.job_classification);
END;
```

Complex Security Authorizations and Triggers

Triggers are commonly used to enforce complex security authorizations for table data. Only use triggers to enforce complex security authorizations that cannot be defined using the database security features provided with Oracle. For example, a trigger can prohibit updates to salary data of the EMP table during weekends, holidays, and non-working hours.

When using a trigger to enforce a complex security authorization, it is best to use a BEFORE statement trigger. Using a BEFORE statement trigger has these benefits:

- The security check is done before the triggering statement is allowed to execute so that no wasted work is done by an unauthorized statement.
- The security check is performed only once for the triggering statement, not for each row affected by the triggering statement.

Example This example shows a trigger used to enforce security. The comments within the code explain the functionality of the trigger.

```
CREATE TRIGGER emp_permit_changes
BEFORE INSERT OR DELETE OR UPDATE ON emp
DECLARE
    dummy INTEGER;
    not_on_weekends EXCEPTION;
    not_on_holidays EXCEPTION;
    non_working_hours EXCEPTION;
BEGIN
    /* check for weekends */
    IF (TO_CHAR(sysdate, 'DY') = 'SAT' OR
        TO_CHAR(sysdate, 'DY') = 'SUN') THEN
        RAISE not_on_weekends;
    END IF;
    /* check for company holidays */
    SELECT COUNT(*) INTO dummy FROM company_holidays
        WHERE TRUNC(day) = TRUNC(sysdate);
    /* TRUNC gets rid of time parts of dates */
    IF dummy > 0 THEN
        RAISE not_on_holidays;
    END IF;
    /* Check for work hours (8am to 6pm) */
    IF (TO_CHAR(sysdate, 'HH24') < 8 OR
        TO_CHAR(sysdate, 'HH24') > 18) THEN
        RAISE non_working_hours;
    END IF;
EXCEPTION
```

```

WHEN not_on_weekends THEN
    raise_application_error(-20324,'May not change '
        ||'employee table during the weekend');
WHEN not_on_holidays THEN
    raise_application_error(-20325,'May not change '
        ||'employee table during a holiday');
WHEN non_working_hours THEN
    raise_application_error(-20326,'May not change '
        ||'emp table during non-working hours');
END;

```

Transparent Event Logging and Triggers

Triggers are very useful when you want to transparently perform a related change in the database following certain events.

Example The REORDER trigger example on page 9 – 20 shows a trigger that reorders parts as necessary when certain conditions are met (that is, a triggering statement is issued and the PARTS_ON_HAND value is less than the REORDER_POINT value).

Derived Column Values and Triggers

Triggers can derive column values automatically based upon a value provided by an INSERT or UPDATE statement. This type of trigger is useful to force values in specific columns that depend on the values of other columns in the same row. BEFORE row triggers are necessary to complete this type of operation because

- the dependant values must be derived before the insert or update occurs so that the triggering statement can use the derived values.
- the trigger must fire for each row affected by the triggering INSERT or UPDATE statement.

Example The following example illustrates how a trigger can be used to derive new column values for a table whenever a row is inserted or updated. Comments within the code explain its functionality.

```
BEFORE INSERT OR UPDATE OF ename ON emp

/* Before updating the ENAME field, derive the values for
   the UPPERNAME and SOUNDEXNAME fields. Users should be
   restricted from updating these fields directly. */
FOR EACH ROW

BEGIN
    :new.uppername := UPPER(:new.ename);
    :new.soundexname := SOUNDEX(:new.ename);
END;
```


Using Dynamic SQL

This chapter describes the dynamic SQL package, `DBMS_SQL`. The following topics are described in this chapter:

- the differences between the `DBMS_SQL` package and the Oracle Call Interfaces
- using the `DBMS_SQL` package to execute DDL
- procedures and functions provided in the `DBMS_SQL` package

Overview

You can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are not embedded in your source program; rather, they are stored in character strings that are input to, or built by, the program at runtime.

This permits you to create procedures that are more general purpose. For example, using dynamic SQL allows you to create a procedure that operates on a table whose name is not known until runtime.

Additionally, you can parse any data manipulation language (DML) or data definition language (DDL) statement using the DBMS_SQL package. This helps solve the problem of not being able to parse data definition language statements directly using PL/SQL. For example, you might now choose to issue a DROP TABLE statement from within a stored procedure by using the PARSE procedure supplied with the DBMS_SQL package.

Creating the DBMS_SQL Package

To create the DBMS_SQL package, submit the DBMSSQL.SQL and PRVTSQL.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script. See page 7 – 39 for information on granting the necessary privileges to users who will be executing this package.

Using DBMS_SQL

The ability to use dynamic SQL from within stored procedures generally follows the model of the Oracle Call Interface (OCI). You should refer to the *Programmer's Guide to the Oracle Call Interface* for additional information on the concepts presented in this chapter.

PL/SQL differs somewhat from other common programming languages, such as C. For example, addresses (also called pointers) are not user visible in PL/SQL. As a result, there are some differences between the Oracle Call Interface and the DBMS_SQL package. These differences include the following:

- The OCI uses bind by address, while the DBMS_SQL package uses bind by value.
- With DBMS_SQL you must call VARIABLE_VALUE to retrieve the value of an OUT parameter for an anonymous block, and you must call COLUMN_VALUE after fetching rows to actually retrieve the values of the columns in the rows into your program.
- The current release of the DBMS_SQL package does not provide DESCRIBE or CANCEL cursor procedures, nor support for the array interface.
- Indicator variables are not required because nulls are fully supported as values of a PL/SQL variable.

A sample usage of the DBMS_SQL package is shown below. For users of the Oracle Call Interfaces, this code should seem fairly straightforward. Each of the functions and procedures used in this example is described later in this chapter. A more detailed example, which shows how you can use the DBMS_SQL package to build a query statement dynamically, begins on page 10 – 22. This example does not actually require the use of dynamic SQL, because the text of the statement is known at compile time. However, it illustrates the concepts of this package.

```

/* The DEMO procedure deletes all of the employees from the EMP
 * table whose salaries are greater than the salary that you
 * specify when you run DEMO. */

CREATE OR REPLACE PROCEDURE demo(salary IN NUMBER) AS
    cursor_name INTEGER;
    rows_processed INTEGER;
BEGIN
    cursor_name := dbms_sql.open_cursor;
    dbms_sql.parse(cursor_name, 'DELETE FROM emp WHERE sal > :x',
                    dbms_sql.v7);
    dbms_sql.bind_variable(cursor_name, ':x', salary);
    rows_processed := dbms_sql.execute(cursor_name);
    dbms_sql.close_cursor(cursor_name);
EXCEPTION
WHEN OTHERS THEN
    dbms_sql.close_cursor(cursor_name);
END;
```

Execution Flow

The typical flow of procedure calls is shown in Figure 11-1. A general explanation of these procedures follows. Each of these procedures is described in greater detail starting on page 10 – 7.

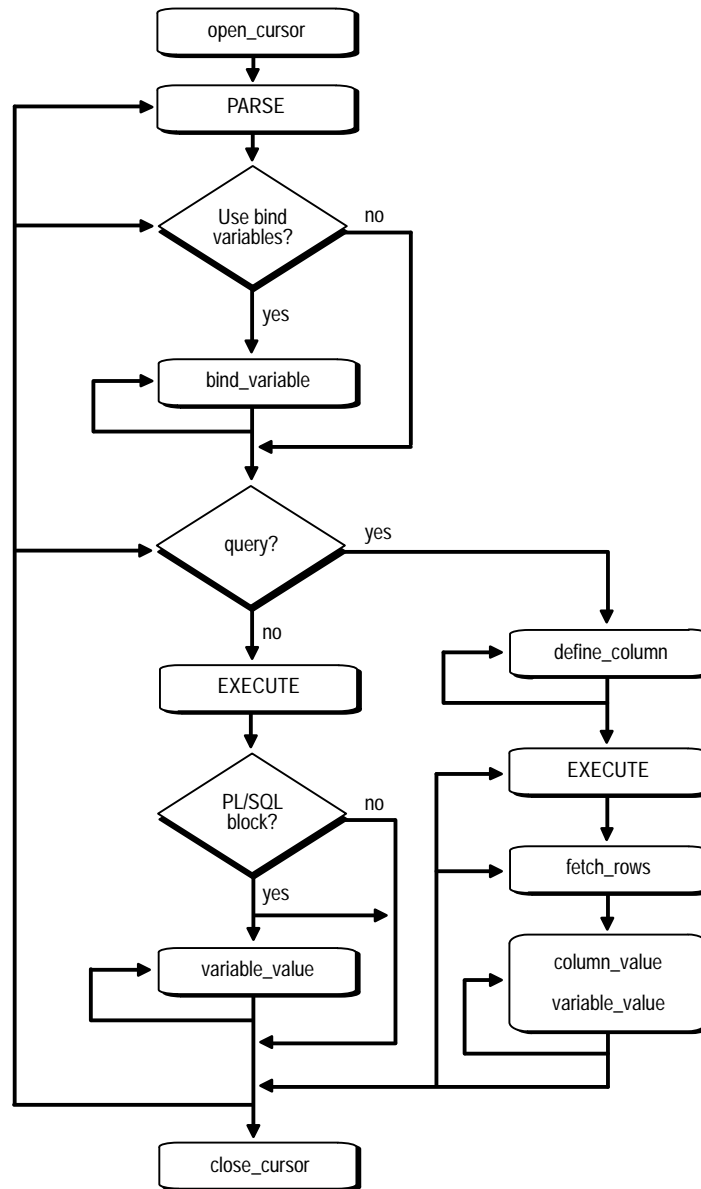


Figure 10 – 1 DBMS_SQL Execution Flow

OPEN_CURSOR

To process a SQL statement, you must have an open cursor. When you call the OPEN_CURSOR function, you receive a cursor ID number for the data structure representing a valid cursor maintained by Oracle. These cursors are distinct from cursors defined at the precompiler, OCI, or PL/SQL level, and are used only by the DBMS_SQL package.

PARSE

Every SQL statement must be parsed by calling the PARSE procedure. Parsing the statement checks the statement's syntax and associates it with the cursor in your program. A complete explanation of how SQL statements are parsed is included in the *Oracle7 Server Tuning* manual.

You can parse any data manipulation language or data definition language statements. Data definition language statements are executed on the parse, which performs the implied commit.



Attention: When parsing a data definition language statement to drop a package or a procedure, a deadlock can occur if a procedure in the package is still in use by you. After a call to a procedure, that procedure is considered to be in use until execution has returned to the user side. Any such deadlock will timeout after five minutes.

BIND_VARIABLE

Many data manipulation language statements require that data in your program be input to Oracle. When you define a SQL statement that contains input data to be supplied at runtime, you must use placeholders in the SQL statement to mark where data must be supplied.

For each placeholder in the SQL statement, you must call the BIND_VARIABLE procedure to supply the value of a variable in your program to the placeholder. When the SQL statement is subsequently executed, Oracle uses the data that your program has placed in the output and input, or bind, variables.

DEFINE_COLUMN

The columns of the row being selected in a SELECT statement are identified by their relative positions as they appear in the select list, from left to right. For a query, you must call DEFINE_COLUMN to specify the variables that are to receive the SELECT values, much the way an INTO clause does for a static query.

DEFINE_COLUMN_LONG

You use the DEFINE_COLUMN_LONG procedure to define LONG columns, in the same way that DEFINE_COLUMN is used to define non-LONG columns. You must call DEFINE_COLUMN_LONG before using the COLUMN_VALUE_LONG to fetch from the LONG column.

EXECUTE

Call the EXECUTE function to execute your SQL statement.

FETCH_ROWS

Call FETCH_ROWS to retrieve the rows that satisfy the query. Each successive fetch retrieves another row, until the fetch is unable to retrieve anymore rows. Instead of calling EXECUTE and then FETCH_ROWS, you may find it more efficient to call EXECUTE_AND_FETCH if you are calling EXECUTE for a single iteration.

VARIABLE_VALUE

For queries, call COLUMN_VALUE to determine the value of a column retrieved by the FETCH_ROWS call. For anonymous blocks containing calls to PL/SQL procedures, call VARIABLE_VALUE to retrieve the values assigned to the output variables of the PL/SQL procedures when they were executed.

COLUMN_VALUE_LONG

To fetch just part of a LONG database column (which can be up to two gigabytes in size), you use the COLUMN_VALUE_LONG procedure. You can specify the offset (in bytes) into the column value, and the number of bytes to fetch.

CLOSE_CURSOR

When you no longer need a cursor for a session, close the cursor by calling CLOSE_CURSOR. If you are using an Oracle Open Gateway, you may need to close cursors at other times as well. Consult your Oracle Open Gateway documentation for additional information. If you neglect to close a cursor, the memory used by that cursor remains allocated even though it is no longer needed.

Security

This section describes the security domain for DBMS_SQL procedures when you are using the Oracle Server or Trusted Oracle Server.

For Oracle Server Users

Any DBMS_SQL procedures called from an anonymous PL/SQL block are executed using the privileges of the current user. Any DBMS_SQL procedures called from a stored procedure are executed using the privileges of the owner of the stored procedure.

For Trusted Oracle Server Users

Any DBMS_SQL procedures called from an anonymous PL/SQL block are executed using the privileges of the current user. Any DBMS_SQL procedures called from a stored procedure are executed using the discretionary access control (DAC) and system privileges of the owner of the stored procedure and the mandatory access control (MAC) privileges of the current user.

Procedures and Functions

Table 10 – 1 provides a brief description of each of the procedures and functions associated with the DBMS_SQL package, which are described in detail later in this chapter. An example of how these functions can be used begins on page 10 – 22.

Function/Procedure	Description	Refer to Page
OPEN_CURSOR	Return cursor ID number of new cursor.	10 – 9
PARSE	Parse given statement.	10 – 9
BIND_VARIABLE	Bind a given value to a given variable.	10 – 10
DEFINE_COLUMN	Define a column to be selected from the given cursor, used only with SELECT statements.	10 – 12

Table 10 – 1 DBMS_SQL Package Functions and Procedures

Function/Procedure	Description	Refer to Page
DEFINE_COLUMN_LONG	Define a LONG column to be selected from the given cursor, used only with SELECT statements.	10 – 14
EXECUTE	Execute a given cursor.	10 – 14
EXECUTE_AND_FETCH	Execute a given cursor and fetch rows.	10 – 15
FETCH_ROWS	Fetch a row from a given cursor.	10 – 15
COLUMN_VALUE	Returns value of the cursor element for a given position in a cursor.	10 – 16
COLUMN_VALUE_LONG	Returns a selected part of a LONG column, that has been defined using DEFINE_COLUMN_LONG.	10 – 17
VARIABLE_VALUE	Returns value of named variable for given cursor.	10 – 18
IS_OPEN	Returns TRUE if given cursor is open.	10 – 20
CLOSE_CURSOR	Closes given cursor and frees memory.	10 – 20
LAST_ERROR_POSITION	Returns byte offset in the SQL statement text where the error occurred.	10 – 21
LAST_ROW_COUNT	Returns cumulative count of the number of rows fetched.	10 – 21
LAST_ROW_ID	Returns ROWID of last row processed.	10 – 21
LAST_SQL_FUNCTION_CODE	Returns SQL function code for statement.	10 – 21

Table 10 – 1 DBMS_SQL Package Functions and Procedures

OPEN_CURSOR Function

Call OPEN_CURSOR to open a new cursor. When you no longer need this cursor, you must close it explicitly by calling CLOSE_CURSOR.

You can use cursors to execute the same SQL statement repeatedly or to execute a new SQL statement. When a cursor is reused, the contents of the corresponding cursor data area are reset when the new SQL statement is parsed. It is never necessary to close and reopen a cursor before reusing it.

Syntax

The OPEN_CURSOR function returns the cursor ID number of the new cursor. The syntax for this function is

```
DBMS_SQL.OPEN_CURSOR RETURN INTEGER;
```

PARSE Procedure

Call PARSE to parse the given statement in the given cursor. Currently, unlike the OCI OPARSE call, which supports deferred parsing, all statements are parsed immediately. This may change in future versions; you should not rely on this behavior.

Syntax

The parameters for the PARSE procedure are described in Table 10 – 2. The syntax for this procedure is

```
DBMS_SQL.PARSE(  
    c                IN INTEGER,  
    statement        IN VARCHAR2,  
    language_flag    IN INTEGER);
```

Parameter	Description
c	Specify the ID number of the cursor in which to parse the statement.
statement	Provide the SQL statement to be parsed. Your SQL statement should not include a final semicolon.
language_flag	This parameter determines how Oracle handles the SQL statement. The following options are recognized for this parameter: V6 – specified Version 6 behavior V7 – specifies Oracle7 behavior NATIVE – specifies normal behavior for the database to which the program is connected.

Table 10 – 2 DBMS_SQL.PARSE Procedure Parameters

BIND_VARIABLE Procedures

Call BIND_VARIABLE to bind a given value to a given variable in a cursor, based on the name of the variable in the statement. If the variable is an IN or IN/OUT variable, the given bind value must be valid for the variable type. Bind values for OUT variables are ignored.

The bind variables of a SQL statement are identified by their names. When binding a value to a bind variable, the string identifying the bind variable in the statement must contain a leading colon, as shown in the following example:

```
SELECT emp_name FROM emp WHERE SAL > :X;
```

For this example, the corresponding bind call would look similar to

```
BIND_VARIABLE(cursor_name, ':X', 3500);
```

Syntax

The parameters for the BIND_VARIABLE procedures are described in Table 10 – 3. The syntax for these procedures is shown below. Notice that the BIND_VARIABLE procedure is overloaded to accept different datatypes.

```
DBMS_SQL.BIND_VARIABLE(c IN INTEGER,  
name IN VARCHAR2,  
value IN <datatype>);
```

where <datatype> can be any one of the following types:

```
NUMBER  
DATE
```

MLSLABEL
VARCHAR2

The following syntax is also supported for the BIND_VARIABLE procedure. The square brackets [] indicate optional parameters.

```
DBMS_SQL.BIND_VARIABLE(  
    c                IN INTEGER,  
    name             IN VARCHAR2,  
    value            IN VARCHAR2  
    [,out_value_size IN INTEGER]);
```

```
DBMS_SQL.BIND_VARIABLE_CHAR(  
    c                IN INTEGER,  
    name             IN VARCHAR2,  
    value            IN CHAR  
    [,out_value_size IN INTEGER]);
```

```
DBMS_SQL.BIND_VARIABLE_RAW(  
    c                IN INTEGER,  
    name             IN VARCHAR2,  
    value            IN RAW  
    [,out_value_size IN INTEGER]);
```

```
DBMS_SQL.BIND_VARIABLE_ROWID(  
    c                IN INTEGER,  
    name             IN VARCHAR2,  
    value            IN ROWID);
```

Parameter	Description
c	Specify the ID number of the cursor to which you want to bind a value.
name	Provide the name of the variable in the statement.
value	Provide the value that you want to bind to the variable in the cursor. For IN and IN/OUT variables, the value has the same type as the type of the value being passed in for this parameter.
out_value_size	The maximum expected OUT value size, in bytes, for the VARCHAR2, RAW, CHAR OUT or IN/OUT variable. If no size is given, the length of the current value is used.

Table 10 – 3 DBMS_SQL.BIND_VARIABLE Procedure Parameters

Processing Queries

If you are using dynamic SQL to process a query, you must perform the following steps:

1. Specify the variables that are to receive the values returned by the SELECT statement by calling DEFINE_COLUMN.
2. Execute your SELECT statement by calling EXECUTE.
3. Call FETCH_ROWS (or EXECUTE_AND_FETCH) to retrieve the rows that satisfied your query.
4. Call COLUMN_VALUE to determine the value of a column retrieved by the FETCH_ROWS call for your query. If you used anonymous blocks containing calls to PL/SQL procedures, you must call VARIABLE_VALUE to retrieve the values assigned to the output variables of these procedures.

DEFINE_COLUMN Procedure

This procedure is only used with SELECT cursors. Call DEFINE_COLUMN to define a column to be selected from the given cursor. The column being defined is identified by its relative position in the SELECT list of the statement in the given cursor. The type of the COLUMN value determines the type of the column being defined.

Syntax

The parameters for the DEFINE_COLUMN procedure are described in Table 10 – 4. The syntax for this procedure is shown below. Notice that this procedure is overloaded to accept different datatypes.

```
DBMS_SQL.DEFINE_COLUMN(  
    c                IN INTEGER,  
    position         IN INTEGER  
    column           IN <datatype>);
```

where <datatype> can be any one of the following types:

```
NUMBER  
DATE  
MLSLABEL
```

The following syntax is also supported for the DEFINE_COLUMN procedure:

```
DBMS_SQL.DEFINE_COLUMN(  
    c                IN INTEGER,  
    position         IN INTEGER,  
    column           IN VARCHAR2,  
    column_size      IN INTEGER);
```

```
DBMS_SQL.DEFINE_COLUMN_CHAR(  
    c                IN INTEGER,  
    position         IN INTEGER,  
    column           IN CHAR,  
    column_size      IN INTEGER);
```

```
DBMS_SQL.DEFINE_COLUMN_RAW(  
    c                IN INTEGER,  
    position         IN INTEGER,  
    column           IN RAW,  
    column_size      IN INTEGER);
```

```
DBMS_SQL.DEFINE_COLUMN_ROWID(  
    c                IN INTEGER,  
    position         IN INTEGER,  
    column           IN ROWID);
```

Parameter	Description
c	The ID number of the cursor for the row being defined to be selected.
position	The relative position of the column in the row being defined. The first column in a statement has position 1.
column	The value of the column being defined. The type of this value determines the type for the column being defined.
column_size	The maximum expected size of the column value, in bytes, for columns of type VARCHAR2, CHAR, and RAW.

Table 10 – 4 DBMS_SQL.DEFINE_COLUMN Procedure Parameters

DEFINE_COLUMN_
LONG Procedure

Call this procedure to define a LONG column for a SELECT cursor. The column being defined is identified by its relative position in the SELECT list of the statement for the given cursor. The type of the COLUMN value determines the type of the column being defined.

Syntax

The parameters of DEFINE_COLUMN_LONG are described in Table 10 – 5. The syntax is

```
DBMS_SQL.DEFINE_COLUMN_LONG(  
    c                IN INTEGER,  
    position         IN INTEGER);
```

Parameter	Description
c	The ID number of the cursor for the row being defined to be selected.
position	The relative position of the column in the row being defined. The first column in a statement has position 1.

Table 10 – 5 DBMS_SQL.DEFINE_COLUMN_LONG Procedure Parameters

EXECUTE Function

Call EXECUTE to execute a given cursor. This function accepts the ID number of the cursor and returns the number of rows processed. The return value is only valid for INSERT, UPDATE, and DELETE statements; for other types of statements, including DDL, the return value is undefined and should be ignored.

Syntax

The syntax for the EXECUTE function is

```
DBMS_SQL.EXECUTE (  
    c                IN INTEGER)  
RETURN INTEGER;
```

EXECUTE_AND_FETCH Function

Call EXECUTE_AND_FETCH to execute the given cursor and fetch rows. This function provides the same functionality as calling EXECUTE and then calling FETCH_ROWS. Calling EXECUTE_AND_FETCH instead, however, may cut down on the number of network round-trips when used against a remote database.

Syntax

The EXECUTE_AND_FETCH function returns the number of rows actually fetched. The parameters for this procedure are described in Table 10 – 6, and the syntax is shown below.

```
DBMS_SQL.EXECUTE_AND_FETCH(  
    c                IN INTEGER,  
    exact            IN BOOLEAN DEFAULT FALSE)  
RETURN INTEGER;
```

Parameter	Description
c	Specify the ID number of the cursor to execute and fetch.
exact	Set to TRUE to raise an exception if the number of rows actually matching the query differs from one. Even if an exception is raised, the rows are still fetched and available.

Table 10 – 6 DBMS_SQL.EXECUTE_AND_FETCH Function Parameters

FETCH_ROWS Function

Call FETCH_ROWS to fetch a row from a given cursor. You can call FETCH_ROWS repeatedly as long as there are rows remaining to be fetched. These rows are retrieved into a buffer, and must be read by calling COLUMN_VALUE, for each column, after each call to FETCH_ROWS.

Syntax

The FETCH_ROWS function accepts the ID number of the cursor to fetch, and returns the number of rows actually fetched. The syntax for this function is shown below.

```
DBMS_SQL.FETCH_ROWS(  
    c                IN INTEGER)  
RETURN INTEGER;
```


COLUMN_VALUE Procedure

This procedure returns the value of the cursor element for a given position in a given cursor. This procedure is used to access the data fetched by calling `FETCH_ROWS`.

Syntax

The parameters for the `COLUMN_VALUE` procedure are described in Table 10 – 7. The syntax for this procedure is shown below. The square brackets `[]` indicate optional parameters.

```
DBMS_SQL.COLUMN_VALUE(  
    c                IN INTEGER,  
    position         IN INTEGER,  
    value            OUT <datatype>,  
    [,column_error   OUT NUMBER]  
    [,actual_length  OUT INTEGER]);
```

where <datatype> can be any one of the following types:

```
NUMBER  
DATE  
MLSLABEL  
VARCHAR2
```

The following syntax is also supported for the `COLUMN_VALUE` procedure:

```
DBMS_SQL.COLUMN_VALUE_CHAR(  
    c                IN INTEGER,  
    position         IN INTEGER,  
    value            OUT CHAR  
    [,column_error   OUT NUMBER]  
    [,actual_length  OUT INTEGER]);
```

```
DBMS_SQL.COLUMN_VALUE_RAW(  
    c                IN INTEGER,  
    position         IN INTEGER,  
    value            OUT RAW  
    [,column_error   OUT NUMBER]  
    [,actual_length  OUT INTEGER]);
```

```

dbms_sql.column_value_rowid(
    c                IN INTEGER,
    position         IN INTEGER,
    value            OUT ROWID
    [,column_error   OUT NUMBER]
    [,actual_length  OUT INTEGER]);

```

Parameter	Mode	Description
c	IN	Specify the ID number of the cursor from which you are fetching the values.
position	IN	Specify the relative position of the column in the cursor. The first column in a statement has position 1.
value	OUT	Returns the value at the specified column and row. If the row number specified is greater than the total number of rows fetched, you receive an error message. Oracle raises exception ORA-06562, inconsistent_type, if the type of this output parameter differs from the actual type of the value, as defined by the call to DEFINE_COLUMN.
column_error	OUT	Returns any error code for the specified column value.
actual_length	OUT	Returns the actual length, before any truncation, of the value in the specified column.

Table 10 – 7 DBMS_SQL.COLUMN_VALUE Procedure Parameters

COLUMN_VALUE_LONG Procedure

This procedure returns the value of the cursor element for a given position, offset, and size in a given cursor. This procedure is used to access the data fetched by calling FETCH_ROWS.

Syntax

The parameters of the COLUMN_VALUE_LONG procedure are described in Table 10 – 8. The syntax of the procedure is

```

DBMS_SQL.COLUMN_VALUE_LONG(
    c                IN INTEGER,
    position         IN INTEGER,
    length           IN INTEGER,
    offset           IN INTEGER,
    value            OUT VARCHAR2,
    value_length     OUT INTEGER);

```

Parameter	Description
c	The ID number of the cursor for the row being defined to be selected.
position	The relative position of the column in the row being defined. The first column in a statement has position 1.
length	The length in bytes of the segment of the column value that is to be selected.
offset	The byte position in the LONG column at which the SELECT is to start.
value	The value of the column segment to be SELECTed.
value_length	The (returned) length of the value that was SELECTed.

Table 10 – 8 DBMS_SQL.COLUMN_VALUE_LONG Procedure Parameters

VARIABLE_VALUE Procedure

This procedure returns the value of the named variable for a given cursor. It is also used to return the values of bind variables inside PL/SQL blocks.

Syntax

The parameters for the VARIABLE_VALUE procedure are described in Table 10 – 9. The syntax for this procedure is shown below.

```
DBMS_SQL.VARIABLE_VALUE (
    c                IN INTEGER,
    name             IN VARCHAR2,
    value            OUT <datatype>);
```

where <datatype> can be any one of the following types:

- NUMBER
- DATE
- MLSLABEL
- VARCHAR2

The following syntax is also supported for the VARIABLE_VALUE procedure:

```
DBMS_SQL.VARIABLE_VALUE_CHAR (
    c                IN INTEGER,
    name             IN VARCHAR2,
    value            OUT CHAR);
```

```

DBMS_SQL.VARIABLE_VALUE_RAW(
    c            IN INTEGER,
    name         IN VARCHAR2,
    value        OUT RAW);

```

```

DBMS_SQL.VARIABLE_VALUE_ROWID(
    c            IN INTEGER,
    name         IN VARCHAR2,
    value        OUT ROWID);

```

Parameter	Mode	Description
c	IN	Specify the ID number of the cursor from which to get the values.
name	IN	Specify the name of the variable for which you are retrieving the value.
value	OUT	Returns the value of the variable for the specified position. Oracle raises exception ORA-06562, inconsistent_type, if the type of this output parameter differs from the actual type of the value, as defined by the call to BIND_VARIABLE.

Table 10 – 9 DBMS_SQL.VARIABLE_VALUE Procedure Parameters

Processing Updates, Inserts and Deletes

If you are using dynamic SQL to process an INSERT, UPDATE, or DELETE, you must perform the following steps:

1. You must first execute your INSERT, UPDATE, or DELETE statement by calling EXECUTE. The EXECUTE procedure is described on page 10 – 14.
2. If you used anonymous blocks containing calls to PL/SQL procedures, you must call VARIABLE_VALUE to retrieve the values assigned to the output variables of these procedures. The VARIABLE_VALUE procedure is described on page 10 – 18.

IS_OPEN Function

The IS_OPEN function returns TRUE if the given cursor is currently open.

Syntax

The IS_OPEN function accepts the ID number of a cursor, and returns TRUE if the cursor is currently open, or FALSE if it is not. The syntax for this function is

```
DBMS_SQL.IS_OPEN(  
    c                               IN INTEGER)  
RETURN BOOLEAN;
```

CLOSE_CURSOR Procedure

Call CLOSE_CURSOR to close a given cursor.

Syntax

The parameter for the CLOSE_CURSOR procedure is described in Table 10 – 10. The syntax for this procedure is

```
DBMS_SQL.CLOSE_CURSOR(  
    c                               IN OUT INTEGER);
```

Parameter	Mode	Description
c	IN	Specify the ID number of the cursor that you want to close.
	OUT	The cursor is set to null. After you call CLOSE_CURSOR, the memory allocated to the cursor is released and you can no longer fetch from that cursor.

Table 10 – 10 DBMS_SQL.CLOSE_CURSOR Procedure Parameters

Locating Errors

There are additional functions in the DBMS_SQL package for obtaining information about the last referenced cursor in the session. The values returned by these functions are only meaningful immediately after a SQL statement is executed. In addition, some error-locating functions are only meaningful after certain DBMS_SQL calls. For example, you call LAST_ERROR_POSITION immediately after a PARSE.

LAST_ERROR_POSITION Function

Returns the byte offset in the SQL statement text where the error occurred. The first character in the SQL statement is at position 0.

```
DBMS_SQL.LAST_ERROR_POSITION RETURN INTEGER;
```

Call this function after a PARSE call, before any other DBMS_SQL procedures or functions are called.

LAST_ROW_COUNT Function

Returns the cumulative count of the number of rows fetched.

```
DBMS_SQL.LAST_ROW_COUNT RETURN INTEGER;
```

Call this function after a FETCH_ROWS or an EXECUTE_AND_FETCH call. If called after an EXECUTE call, the value returned will be zero.

LAST_ROW_ID Function

Returns the ROWID of the last row processed.

```
DBMS_SQL.LAST_ROW_ID RETURN ROWID;
```

Call this function after a FETCH_ROWS or an EXECUTE_AND_FETCH call.

LAST_SQL_FUNCTION_CODE Function

Returns the SQL function code for the statement. These codes are listed in the *Programmer's Guide to the Oracle Call Interface*.

```
DBMS_SQL.LAST_SQL_FUNCTION_CODE RETURN INTEGER;
```

You should call this function immediately after the SQL statement is executed; otherwise, the return value is undefined.

Examples

This section provides example procedures that make use of the DBMS_SQL package.

Example 1 The following sample procedure is passed a SQL statement, which it then parses and executes:

```
CREATE OR REPLACE PROCEDURE exec(string IN varchar2) AS
    cursor_name INTEGER;
    ret INTEGER;
BEGIN
    cursor_name := DBMS_SQL.OPEN_CURSOR;

    --DDL statements are executed by the parse call, which
    --performs the implied commit
    DBMS_SQL.PARSE(cursor_name, string, DBMS_SQL.V7);
    ret := DBMS_SQL.EXECUTE(cursor_name);
    DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
```

Creating such a procedure allows you to perform the following operations:

- The SQL statement can be dynamically generated at runtime by the calling program.
- The SQL statement can be a DDL statement.

For example, after creating this procedure, you could make the following call:

```
exec('create table acct(c1 integer)');
```

You could even call this procedure remotely, as shown in the following example. This allows you to perform remote DDL.

```
exec@hq.com('CREATE TABLE acct(c1 INTEGER)');
```

Example 2 The following sample procedure is passed the names of a source and a destination table, and copies the rows from the source table to the destination table. This sample procedure assumes that both the source and destination tables have the following columns:

```
ID of type NUMBER
NAME of type VARCHAR2(30)
BIRTHDATE of type DATE
```

This procedure does not specifically require the use of dynamic SQL; however, it illustrates the concepts of this package.

```

CREATE OR REPLACE PROCEDURE copy(source      IN VARCHAR2,
                                destination IN VARCHAR2) is

-- This procedure copies rows from a given source table to a
-- given destination table assuming that both source and
-- destination tables have the following columns:
--   - ID of type NUMBER,
--   - NAME of type VARCHAR2(30),
--   - BIRTHDATE of type DATE.
    id          NUMBER;
    name        VARCHAR2(30);
    birthdate   DATE;
    source_cursor    INTEGER;
    destination_cursor INTEGER;
    ignore        INTEGER;
BEGIN

    -- prepare a cursor to select from the source table
    source_cursor := dbms_sql.open_cursor;
    DBMS_SQL.PARSE(source_cursor,
        'SELECT id, name, birthdate FROM ' || source,
        DBMS_SQL.V7);
    DBMS_SQL.DEFINE_COLUMN(source_cursor, 1, id);
    DBMS_SQL.DEFINE_COLUMN(source_cursor, 2, name, 30);
    DBMS_SQL.DEFINE_COLUMN(source_cursor, 3, birthdate);
    ignore := DBMS_SQL.EXECUTE(source_cursor);

    -- prepare a cursor to insert into the destination table
    destination_cursor := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(destination_cursor,
        'INSERT INTO ' || destination ||
        ' VALUES (:id, :name, :birthdate)',
        DBMS_SQL.V7);

    -- fetch a row from the source table and
    -- insert it into the destination table
    LOOP
        IF DBMS_SQL.FETCH_ROWS(source_cursor)>0 THEN
            -- get column values of the row
            DBMS_SQL.COLUMN_VALUE(source_cursor, 1, id);
            DBMS_SQL.COLUMN_VALUE(source_cursor, 2, name);
            DBMS_SQL.COLUMN_VALUE(source_cursor, 3, birthdate);

            -- bind the row into the cursor that inserts into the
            -- destination table
            -- You could alter this example to require the use of
            -- dynamic SQL by inserting an if condition before the
            -- bind.
            DBMS_SQL.BIND_VARIABLE(destination_cursor, 'id', id);

```



```

        DBMS_SQL.BIND_VARIABLE(destination_cursor, 'name', name);
        DBMS_SQL.BIND_VARIABLE(destination_cursor, 'birhtdate',
                                birthdate);
        ignore := DBMS_SQL.EXECUTE(destination_cursor);
    ELSE

        -- no more row to copy
        EXIT;
    END IF;
END LOOP;

-- commit and close all cursors
COMMIT;
DBMS_SQL.CLOSE_CURSOR(source_cursor);
DBMS_SQL.CLOSE_CURSOR(destination_cursor);

EXCEPTION
    WHEN OTHERS THEN
        IF DBMS_SQL.IS_OPEN(source_cursor) THEN
            DBMS_SQL.CLOSE_CURSOR(source_cursor);
        END IF;
        IF DBMS_SQL.IS_OPEN(destination_cursor) THEN
            DBMS_SQL.CLOSE_CURSOR(destination_cursor);
        END IF;
        RAISE;
END;
```

Managing Dependencies Among Schema Objects

The definitions of certain objects, such as views and procedures, reference other objects, such as tables. Therefore, some objects are dependent upon the objects referenced in their definitions. This chapter discusses how to manage the dependencies among objects. Topics include the following:

- dependency issues
- forcing the compilation of invalid views, procedures, packages, and triggers
- listing dependency management information

If you are using Trusted Oracle, also see the *Trusted Oracle7 Server Administrator's Guide* for information about handling dependencies in Trusted Oracle.

Dependency Issues

When you create a stored procedure or function, Oracle verifies that the operations it performs are possible based on the schema objects accessed. For example, if a stored procedure contains a `SELECT` statement that selects columns from a table, Oracle verifies that the table exists and contains the specified columns. If the table is subsequently redefined so that one of its columns does not exist, the stored procedure may not work properly. For this reason, the stored procedure is said to *depend* on the table.

In cases such as this, Oracle automatically manages dependencies among schema objects. After a schema object is redefined, Oracle automatically recompiles all stored procedures and functions in your database that depend on the redefined object the next time they are called. This recompilation allows Oracle to verify that the procedures and functions can still execute properly based on the newly defined object.

Avoiding Runtime Recompilation

Runtime recompilation reduces runtime performance and the possible resulting runtime compilation errors can halt your applications. Follow these measures to avoid runtime recompilation:

- Do not redefine schema objects (such as tables, views, and stored procedures and functions) while your production applications are running. Redefining objects causes Oracle to recompile stored procedures and functions that depend on them.
- After redefining a schema object, manually recompile dependent procedures, functions, and packages. This measure not only eliminates the performance impact of runtime recompilation, but it also notifies you immediately of compilation errors, allowing you to fix them before production use.

You can manually recompile a procedure, stored function, or package with the `COMPILE` option of the `ALTER PROCEDURE`, `ALTER FUNCTION`, or `ALTER PACKAGE` command. For more information on these commands, see the *Oracle 7 Server SQL Reference* manual.

You can determine the dependencies among the schema objects in your database by running the SQL script UTLDTREE.SQL.



OSDoc

Additional Information: The exact name and location of the UTLDTREE.SQL script may vary depending on your operating system. See this script for more information on how to use it.

- Store procedures and functions in packages whenever possible. If a procedure or function is stored in a package, you can modify its definition without causing Oracle to recompile other procedures and functions that call it.

There are several dependency issues to consider before dropping a procedure or package. Additional information about dependency issues is included in the *Oracle7 Server Concepts* manual. Some guidelines for managing dependencies follow.

Use Packages Whenever Possible Packages are the most effective way of preventing unnecessary dependency checks from being performed. The following example illustrates this benefit.

Assume this situation:

- The standalone procedure PROC depends on a packaged procedure PACK_PROC.
- The PACK_PROC procedure's definition is altered by recompilation of the package body.
- The PACK_PROC procedure's specification is not altered in the package specification.

Even though the package's body is recompiled, the standalone procedure PROC that depends on the packaged procedure PACK_PROC is not invalidated, because the package's specification is not altered.

This technique is especially useful in distributed environments. If procedures are always part of a package, remote procedures that depend on packaged procedures are never invalidated unless a package specification is replaced or invalidated.

Whenever you recompile a procedure, you should consult with any other database administrators and application developers to identify any remote, dependent procedures and ensure that they are also recompiled. This eliminates recompilations at runtime and allows you to detect any compile-time errors that would otherwise be seen by the application user. See "Manually Recompiling Invalid Views, Procedures, Packages, and Triggers" on page 11 – 4 for more information.

The %TYPE and %ROWTYPE Attributes The %TYPE attribute provides the datatype of a variable, constant, or column. This attribute is particularly useful when declaring a variable or procedure argument that refers to a column in a database table. The %ROWTYPE attribute is useful if you want to declare a variable to be a record that has the same structure as a row in a table or view, or a row that is returned by a fetch from a cursor.

When you declare a construct using %TYPE and %ROWTYPE, you do not need to know the datatype of a column or structure of a table. For example, the argument list of a procedure that inserts a row into the EMP table could be declared as

```
CREATE PROCEDURE hire_fire(emp_record emp%ROWTYPE) AS ... END;
```

If you change the column or table structure, the constructs defined on their datatypes or structure automatically change accordingly.

However, while one type of dependency is eliminated using %TYPE or %ROWTYPE, another is created. If you define a construct using *object*%TYPE or *object*%ROWTYPE, the construct depends on *object*. If *object* is altered, the constructs that depend on *object* are invalidated.

Manually Recompiling

Oracle dynamically recompiles an invalid view or PL/SQL program unit the next time it is used. Alternatively, you can force the compilation of an invalid view or program unit using the appropriate SQL command with the COMPILE parameter.

Forced compilations are most often used to test for errors when it is known that a dependent view or program unit is invalid, but is not currently being used; therefore, automatic recompilation would not otherwise occur until the view or program unit is executed.

Invalid dependent objects can be identified by querying the data dictionary views USER_OBJECTS, ALL_OBJECTS, and DBA_OBJECTS; see "Listing Dependency Management Information" on page 11 – 6 for examples.

Manually Recompiling Views

To recompile a view, use the ALTER VIEW command with the COMPILE parameter. The following statement recompiles the view EMP_DEPT contained in your schema:

```
ALTER VIEW emp_dept COMPILE;
```

Privileges Required to Recompile a View

To manually recompile a view, the view must be contained in your schema or you must have the ALTER ANY TABLE system privilege.

Manually Recompiling Procedures and Functions

To recompile a procedure or function (standalone), use the ALTER PROCEDURE or ALTER FUNCTION command with the COMPILE clause. For example, the following statement recompiles the stored procedure UPDATE_SALARY contained in your schema:

```
ALTER PROCEDURE update_salary COMPILE;
```

Manually Recompiling Packages

To recompile either a package body or both a package specification and body, use the ALTER PACKAGE command with the COMPILE parameter. For example, the following SQL*Plus statements recompile just the body and the body and specification of the package ACCT_MGMT_PACKAGE, respectively:

```
SQLPLUS> ALTER PACKAGE acct_mgmt_package COMPILE BODY;
```

```
SQLPLUS> ALTER PACKAGE acct_mgmt_package COMPILE PACKAGE;
```

All packages, procedures, and functions can be recompiled using the following syntax. The objects are compiled in dependency order, enabling each to be compiled only once.

```
SQLPLUS> EXECUTE DBMS_UTILITY.COMPILE_ALL;
```

Privileges Required to Recompile a Procedure or Package

You can manually recompile a procedure or package only if it is contained in your schema *and* you have the ALTER ANY PROCEDURE system privilege.

Manually Recompiling Triggers

An existing trigger, enabled or disabled, can be manually recompiled using the ALTER TRIGGER command. For example, to force the compilation of the trigger named REORDER, enter the following statement:

```
ALTER TRIGGER reorder COMPILE;
```

Privileges Required to Recompile a Trigger

To recompile a trigger, you must own the trigger or have the ALTER ANY TRIGGER system privilege.

Listing Dependency Management Information

The following data dictionary views list information about direct dependencies and dependency management:

- USER_DEPENDENCIES, ALL_DEPENDENCIES, DBA_DEPENDENCIES
- USER_OBJECTS, ALL_OBJECTS, DBA_OBJECTS

Note: For a complete description of these data dictionary views, see the *Oracle7 Server Reference* manual.

Consider the following statements for Examples 1 and 2:

```
CREATE TABLE emp . . . ;

CREATE PROCEDURE hire_emp BEGIN . . . END;

ALTER TABLE emp . . . ;
```

Example 1 Listing the Status of an Object

The ALL_OBJECTS data dictionary view lists information about all the objects available to the current user and the current status (that is, valid or invalid) of each object. For example, the following query lists the names, types, and current status of all objects available to the current user:

```
SELECT object_name, object_type, status
FROM all_objects;
```

The following results might be returned:

OBJECT_NAME	OBJECT_TYPE	STATUS
-----	-----	-----
EMP	TABLE	VALID
HIRE_EMP	PROCEDURE	INVALID

Example 2
Listing Dependencies

The DBA_DEPENDENCIES data dictionary view lists all dependent objects in the database and the objects on which they directly depend. For example, the following query lists all the dependent objects in JWARD's schema:

```
SELECT name, type, referenced_name, referenced_type
FROM sys.dba_dependencies
WHERE owner = 'JWARD';
```

If JWARD issued the example statements at the beginning of this section, the following results might be returned:

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
-----	-----	-----	-----
HIRE_EMP	PROCEDURE	EMP	TABLE

The Dependency Tracking Utility

The *_DEPENDENCIES data dictionary views provide information about only the direct dependencies of objects. As a supplement, you can use a special dependency tracking utility to list both direct and indirect dependents of an object.

To create the dependency tracking utility, you must run the SQL script UTLDTREE.SQL. The location of this file is operating system dependent. The UTLDTREE.SQL script creates the following schema objects:

Table
DEPTREE_TEMPTAB

Structure: object_id NUMBER
 referenced_object_id NUMBER
 nest_level NUMBER
 seq# NUMBER

A temporary table used to store dependency information returned by the DEPTREE_FILL procedure.

View DEPTREE

Column names: nested_level, object_type, owner, object_name, seq#

A view that lists dependency information in the DEPTREE_TEMPTAB table. The parent object is listed with a NESTED_LEVEL of 0, and dependent objects are listed with a nested level greater than 0.

View IDEPTREE

Column name: dependencies

A view that lists dependency information in the DEPTREE_TEMPTAB table. Output is in a graphical format, with dependent objects indented from the objects on which they depend.

Sequence DEPTREE_SEQ	A sequence used to uniquely identify sets of dependency information stored in the DEPTREE_TEMPTAB.
Procedure DEPTREE_FILL	<p>Syntax: DEPTREE_FILL(object_type CHAR, object_owner CHAR, object_name CHAR)</p> <p>A procedure that first clears the DEPTREE_TEMPTAB table in the executor's schema, then fills the same table to indicate the objects that directly or indirectly depend on (that is, reference) the specified object. All objects that recursively reference the specified object are listed, assuming the user has permission to know of their existence.</p>

Using UTLDTREE While Connected as INTERNAL

If you run the UTLDTREE.SQL script and use the utility while connected as INTERNAL, dependency information is gathered and displayed not only for dependent objects, but also for dependent cursors (shared SQL areas).

Example These SQL statements show how the UTLDTREE utility can be used to track dependencies of an object. Assume the following SQL statements:

```
CONNECT scott/tiger;

CREATE TABLE scott.emp ( .... );

CREATE SEQUENCE scott.emp_sequence;

CREATE VIEW scott.sales_employees AS
    SELECT * FROM scott.emp WHERE deptno = 10;

CREATE PROCEDURE scott.hire_salesperson (name VARCHAR2,
    job VARCHAR2, mgr NUMBER, hiredate DATE, sal NUMBER,
    comm NUMBER)
IS
BEGIN
    INSERT INTO scott.sales_employees
        VALUES (scott.emp_sequence.NEXTVAL, name, job, mgr,
            hiredate, sal, comm, 10;
END;

CREATE PROCEDURE scott.fire_salesperson (emp_id NUMBER) IS
BEGIN
    DELETE FROM scott.sales_employees WHERE empno = emp_id;
END;

SELECT * FROM scott.emp;

SELECT * FROM scott.sales_employees;
```

```
EXECUTE scott.hire_salesperson ('ARNALL', 'MANAGER', 7839, /
                                SYSDATE, 1000, 500);

EXECUTE scott.fire_salesperson (7934);
```

Assume that before SCOTT alters the EMP table, he would like to know all the dependent objects that will be invalidated as a result of altering the EMP table. The following procedure execution fills the DEPTREE_TEMPTAB table with dependency information regarding the EMP table (executed using Server Manager):

```
EXECUTE deptree_fill('TABLE', 'SCOTT', 'EMP');
```

The following two queries show the previously collected dependency information for the EMP table:

```
SELECT * FROM deptree;
```

NESTED_LEV	TYPE	OWNER	NAME	SEQ#
0	TABLE	SCOTT	EMP	0
1	VIEW	SCOTT	SALES_EMPLOYEES	1
2	PROCEDURE	SCOTT	FIRE_SALESPERSON	2
2	PROCEDURE	SCOTT	HIRE_SALESPERSON	3

```
SELECT * FROM ideptree;
```

```
DEPENDENCIES
-----
TABLE SCOTT.EMP
VIEW SCOTT.SALES_EMPLOYEES
PROCEDURE SCOTT.FIRE_SALESPERSON
PROCEDURE SCOTT.HIRE_SALESPERSON
```

Alternatively, you can reveal all of the cursors that depend on the EMP table (dependent shared SQL areas currently in the shared pool) using the UTLDTREE utility. After connecting as INTERNAL and collecting dependency information for the table SCOTT.EMP, issue the following two queries:

```
SELECT * FROM deptree;
```

NESTED_LEV	TYPE	OWNER	NAME	SEQ#
0	TABLE	SCOTT	EMP	0
1	CURSOR	<shared>	"select * from scott.emp	0.5
2	CURSOR	<shared>	"select * from scott.sa. . .	7.5
3	CURSOR	<shared>	"BEGIN hire_salesperson. . .	9.5
3	CURSOR	<shared>	"BEGIN fire_salesperson. . .	8.5

```
SELECT * FROM ideptree;
```

```
DEPENDENCIES
```

```
-----  
TABLE STEVE.EMP
```

```
    CURSOR <shared>."select * from scott.emp"
```

```
    CURSOR <shared>."select * from scott.sales_employee"
```

```
    CURSOR <shared>."BEGIN  hire_salesperson ('ARN. . .
```

```
    CURSOR <shared>."BEGIN  fire_salesperson (7934) END"
```

Signalling Events in the Database with Alerters

This chapter describes how to use the DBMS_ALERT package to provide notification, or “alerts”, of database events. Topics include the following:

- register interest in an alert
- disable notification from an alert
- signal an alert
- receive an alert
- example

See the *Trusted Oracle7 Server Administrator's Guide* for additional information about using alerts with Trusted Oracle.

Overview

The DBMS_ALERT package provides support for the asynchronous (as opposed to polling) notification of database events. By appropriate use of this package and database triggers, an application can cause itself to be notified whenever values of interest in the database are changed.

For example, suppose a graphics tool is displaying a graph of some data from a database table. The graphics tool can, after reading and graphing the data, wait on a database alert (DBMS_ALERT.WAITONE) covering the data just read. The tool automatically wakes up when the data is changed by any other user. All that is required is that a trigger be placed on the database table, which then performs a signal (DBMS_ALERT.SIGNAL) whenever the trigger is fired.

Alerts are transaction based. This means that the waiting session does not get alerted until the transaction signalling the alert commits.

There can be any number of concurrent signallers of a given alert, and there can be any number of concurrent waiters on a given alert.

A waiting application is blocked in the database and cannot do any other work.

Note: Because database alerters issue COMMITs, they cannot be used with Oracle Forms. For more information on restrictions on calling stored procedures while Oracle Forms (Runform) is active, refer to your Oracle Forms documentation.

The following procedures are callable from the DBMS_ALERT package:

Function/Procedure	Description	Refer to Page
REGISTER	Receive messages from an alert.	12 – 4
REMOVE	Disable notification from an alert.	12 – 5
SIGNAL	Signal an alert (send message to registered sessions).	12 – 5
WAITANY	Wait TIMEOUT seconds to receive alert message from an alert registered for session.	12 – 6
WAITONE	Wait TIMEOUT seconds to receive message from named alert.	12 – 6
SET_DEFAULTS	Set the polling interval.	12 – 8

Table 12 – 1 DBMS_ALERT Package Functions and Procedures

Creating the DBMS_ALERT Package

To create the DBMS_ALERT package, submit the DBMSALRT.SQL and PRVTALRT.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script. See page 7 – 39 for information on granting the necessary privileges to users who will be executing this package.

Security

Security on this package can be controlled by granting EXECUTE on this package to selected users or roles. You might want to write a cover package on top of this one that restricts the alert names used. EXECUTE privilege on this cover package can then be granted rather than on this package.

Errors

DBMS_ALERT raises the application error –20000 on error conditions. Table 12 – 2 shows the messages, and the procedures that can raise them.

Error Message	Procedure
ORU–10001 lock request error, status: N	SIGNAL
ORU–10015 error: N waiting for pipe status	WAITANY
ORU–10016 error: N sending on pipe 'X'	SIGNAL
ORU–10017 error: N receiving on pipe 'X'	SIGNAL
ORU–10019 error: N on lock request	WAIT
ORU–10020 error: N on lock request	WAITANY
ORU–10021 lock request error; status: N	REGISTER
ORU–10022 lock request error, status: N	SIGNAL
ORU–10023 lock request error; status N	WAITONE
ORU–10024 there are no alerts registered	WAITANY
ORU–10025 lock request error; status N	REGISTER
ORU–10037 attempting to wait on uncommitted signal from same session	WAITONE

Table 12 – 2 DBMS_ALERT Error Messages

Using Alerts

The application can register for multiple events and can then wait for any of them to occur using the WAITANY call.

An application can also supply an optional TIMEOUT parameter to the WAITONE or WAITANY calls. A TIMEOUT of 0 returns immediately if there is no pending alert.

The signalling session can optionally pass a message that will be received by the waiting session.

Alerts can be signalled more often than the corresponding application WAIT calls. In such cases, the older alerts are discarded. The application always gets the latest alert (based on transaction commit times).

If the application does not require transaction-based alerts, then the DBMS_PIPE package may provide a useful alternative; see page 8 – 2.

If the transaction is rolled back after the call to DBMS_ALERT.SIGNAL, no alert occurs.

It is possible to receive an alert, read the data, and find that no data has changed. This is because the data changed after the *prior* alert, but before the data was read for that *prior* alert.

REGISTER Procedure

The REGISTER procedure allows a session to register interest in an alert. The name of the alert is the IN parameter. A session can register interest in an unlimited number of alerts. Alerts should be deregistered when the session no longer has any interest, by calling REMOVE.



Warning: Alert names beginning with 'ORAS' are reserved for use for products provided by Oracle Corporation.

Syntax

The syntax for the REGISTER procedure is

```
DBMS_ALERT.REGISTER(name IN VARCHAR2);
```

REMOVE Procedure

The REMOVE procedure allows a session that is no longer interested in an alert to remove that alert from its registration list. Removing an alert reduces the amount of work done by signalers of the alert.

If a session dies without removing the alert, that alert is eventually (but not immediately) cleaned up.

Syntax

The syntax for the REMOVE procedure is

```
DBMS_ALERT.REMOVE (name IN VARCHAR2) ;
```

SIGNAL Procedure

Call SIGNAL to signal an alert. The effect of the SIGNAL call only occurs when the transaction in which it is made commits. If the transaction rolls back, the SIGNAL call has no effect.

All sessions that have registered interest in this alert are notified. If the interested sessions are currently waiting, they are awakened. If the interested sessions are not currently waiting, then they are notified the next time they do a wait call. Multiple sessions can concurrently perform signals on the same alert. Each session, as it signals the alert, blocks all other concurrent sessions until it commits. This has the effect of serializing the transactions.

Syntax

The parameters for the SIGNAL procedure are described in Table 12 – 3. The syntax for this procedure is

```
DBMS_ALERT.SIGNAL (name IN VARCHAR2 ,  
                   message IN VARCHAR2 ) ;
```

Parameter	Description
name	Specify the name of the alert to signal.
message	Specify the message, of 1800 bytes or less, to associate with this alert. This message is passed to the waiting session. The waiting session might be able to avoid reading the database after the alert occurs by using the information in the message.

Table 12 – 3 DBMS_ALERT.SIGNAL Procedure Parameters

WAITANY Procedure

Call WAITANY to wait for an alert to occur for any of the alerts for which the current session is registered. The same session that waits for the alert may also first signal the alert. In this case remember to commit after the signal and before the wait; otherwise, DBMS_LOCK.REQUEST (which is called by DBMS_ALERT) returns status 4.

Syntax

The parameters for the WAITANY procedure are described in Table 12 – 4. The syntax for this procedure is

```
DBMS_ALERT.WAITANY( name      OUT  VARCHAR2 ,
                    message    OUT  VARCHAR2 ,
                    status      OUT  INTEGER ,
                    timeout     IN   NUMBER DEFAULT MAXWAIT );
```

Parameter	Description
name	Returns the name of the alert that occurred.
message	Returns the message associated with the alert. This is the message provided by the SIGNAL call. Note that if multiple signals on this alert occurred before the WAITANY call, then the message corresponds to the most recent signal call. Messages from prior SIGNAL calls are discarded.
status	The values returned and their associated meanings are as follows: 0 – alert occurred 1 – timeout occurred
timeout	Specify the maximum time to wait for an alert. If no alert occurs before TIMEOUT seconds, this call returns with a status of 1.

Table 12 – 4 DBMS_ALERT.WAITANY Procedure Parameters

WAITONE Procedure

You call WAITONE to wait for a specific alert to occur. A session that is the first to signal an alert can also wait for the alert in a subsequent transaction. In this case, remember to commit after the signal and before the wait; otherwise, DBMS_LOCK.REQUEST (which is called by DBMS_ALERT) returns status 4.

The parameters for the WAITONE procedure are described in Table 12 – 5. The syntax for this procedure is

```
DBMS_ALERT.WAITONE(name      IN  VARCHAR2 ,
                    message   OUT VARCHAR2 ,
                    status    OUT  INTEGER ,
                    timeout    IN  NUMBER DEFAULT MAXWAIT);
```

Parameter	Description
name	Specify the name of the alert to wait for.
message	Returns the message associated with the alert. This is the message provided by the SIGNAL call. Note that if multiple signals on this alert occurred before the WAITONE call, then the message corresponds to the most recent signal call. Messages from prior SIGNAL calls are discarded.
status	The values returned and their associated meanings are as follows: 0 – alert occurred 1 – timeout occurred
timeout	Specify the maximum time to wait for an alert. If the named alert does not occurs before TIMEOUT seconds, this call returns with a status of 1.

Table 12 – 5 DBMS_ALERT.WAITONE Procedure Parameters

Checking for Alerts

Usually, Oracle is event-driven; that is, there are no polling loops. There are two cases where polling loops can occur:

- Shared mode. If your database is running in shared mode, a polling loop is required to check for alerts from another instance. The polling loop defaults to one second and can be set by the SET_DEFAULTS call.
- WAITANY call. If you use the WAITANY call, and a signalling session does a signal but does not commit within one second of the signal, then a polling loop is required so that this uncommitted alert does not camouflage other alerts. The polling loop begins at a one second interval and exponentially backs off to 30-second intervals.

SET_DEFAULTS Procedure

In case a polling loop is required, use the SET_DEFAULTS procedure to set the POLLING_INTERVAL. The POLLING_INTERVAL is the time, in seconds, to sleep between polls. The default interval is five seconds.

Syntax

The syntax for the SET_DEFAULTS procedure is

```
DBMS_ALERT.SET_DEFAULTS(polling_interval IN NUMBER);
```

Example of Using Alerts

Suppose you want to graph average salaries by department, for all employees. Your application needs to know whenever EMP is changed. Your application would look similar to the code below.

```
dbms_alert.register('emp_table_alert');
readagain:
/* ... read the emp table and graph it */
dbms_alert.waitone('emp_table_alert', :message, :status);
if status = 0 then goto readagain; else
/* ... error condition */
```

The EMP table would have a trigger similar to the following example:

```
CREATE TRIGGER emptrig AFTER INSERT OR UPDATE OR DELETE ON emp
BEGIN
    dbms_alert.signal('emp_table_alert', 'message_text');
END;
```

When the application is no longer interested in the alert, it makes the following request:

```
dbms_alert.remove('emp_table_alert');
```

This reduces the amount of work required by the alert signaller. If a session exits (or dies) while registered alerts exist, they are eventually cleaned up by future users of this package.

The above example guarantees that the application always sees the latest data, although it may not see every intermediate value.

Establishing a Security Policy

Given the many types of mechanisms available to maintain the security of an Oracle database, a discretionary security policy should be designed and implemented to determine

- the level of security at the application level
- system and object privileges
- database roles
- how to grant and revoke privileges and roles
- how to create, alter, and drop roles
- how role use can be controlled

These topics and guidance on developing security policies are discussed in this chapter. If you are using Trusted Oracle, see the *Trusted Oracle7 Server Administrator's Guide* for additional information about establishing an overall system security policy.

Application Security Policy

Draft a security policy for each database application. For example, each developed database application (such as a precompiler program or Oracle Forms form) should have one or more application roles that provide different levels of security when executing the application. The application roles can be granted to user roles or directly to specific usernames.

Applications that potentially allow unrestricted SQL statement execution (such as SQL*Plus or SQL*ReportWriter) also need tight control to prevent malicious access to confidential or important schema objects.

Application Administrators

In large database systems with many database applications (such as precompiler applications or Oracle Forms applications), it may be desirable to have application administrators. An application administrator is responsible for

- creating roles for an application and managing the privileges of each application role
- creating and managing the objects used by a database application
- maintaining and updating the application code and Oracle procedures and packages, as necessary

As the application developer, you might also assume the responsibilities of the application administrator. However, these jobs might be designated to another individual familiar with the database applications.

Roles and Application Privilege Management

Because most database applications involve many different privileges on many different schema objects, keeping track of which privileges are required for each application can be complex. In addition, authorizing users to run an application can involve many GRANT operations. To simplify application privilege management, a role should be created and granted all the privileges required to run each application. In fact, an application might have a number of roles, each granted a specific subset of privileges that allow fewer or more capabilities while running the application.

Example Assume that every administrative assistant uses the Vacation application to record vacation taken by members of the department. You should

1. Create a VACATION role.
2. Grant all privileges required by the Vacation application to the VACATION role.
3. Grant the VACATION role to all administrative assistants or to a role named ADMIN_ASSITS (if previously defined).

Grouping application privileges in a role aids privilege management. Consider the following administrative options:

- You can grant the role, rather than many individual privileges, to those users who run the application. Then, as employees change jobs, only one role grant or revoke (rather than many privilege grants and revokes) is necessary.
- You can change the privileges associated with an application by modifying only the privileges granted to the role, rather than the privileges held by all users of the application.
- You can determine which privileges are necessary to run a particular application by querying the ROLE_TAB_PRIVS and ROLE_SYS_PRIVS data dictionary views.
- You can determine which users have privileges on which applications by querying the DBA_ROLE_PRIVS data dictionary view.

Enabling Application Roles

A single user can use many applications and associated roles. However, you should only allow a user to have the privileges associated with the currently running application role. For example, consider the following scenario:

- The ORDER role (for the ORDER application) contains the UPDATE privilege for the INVENTORY table.
- The INVENTORY role (for the INVENTORY application) contains the SELECT privilege for the INVENTORY table.
- Several order entry clerks have been granted both the ORDER and INVENTORY roles.

Therefore, an order entry clerk who has been granted both roles can presumably use the privileges of the ORDER role when running the INVENTORY application to update the INVENTORY table. However, update modification to the INVENTORY table is not an authorized action when using the INVENTORY application, but only when using the ORDER application.

To avoid such problems, issue a SET ROLE statement at the beginning of each application to automatically enable its associated role and, consequently, disable all others. By using the SET ROLE command, each application dynamically enables particular privileges for a user only when required. A user can make use of an application's privileges only when running a given application, and is prevented from intentionally or unintentionally using them outside the context of an application.

The SET ROLE statement facilitates privilege management in that, in addition to letting you control what information a user can access, it allows you to control when a user can access it. In addition, the SET ROLE statement keeps users operating in a well defined privilege domain. If a user gets all privileges from roles, the user cannot combine them to perform unauthorized operations; see “Enabling and Disabling Roles” on page 13 – 10 for more information.

SET_ROLE Procedure

The DBMS_SESSIONS.SET_ROLE procedure behaves similarly to the SET ROLE statement and can be accessed from PL/SQL. You cannot call SET_ROLE from a stored procedure. This restriction prevents a stored procedure from changing its security domain during its execution. A stored procedure executes under the security domain of the creator of the procedure.

DBMS_SESSION.SET_ROLE is only callable from anonymous PL/SQL blocks. Because PL/SQL does the security check on SQL when an anonymous block is compiled, SET_ROLE will not affect the security role (that is, will not affect the roles enabled) for embedded SQL statements or procedure calls.

For example, if you have a role named ACCT that has been granted privileges that allow you to select from table FINANCE in the JOE schema, the following block will fail:

```
DECLARE
    n NUMBER
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct')
    SELECT empno INTO n FROM JOE.FINANCE
END;
```

This block fails because the security check that verifies that you have the SELECT privilege on table JOE.FINANCE happens at compile time. At compile time, you do not have the ACCT role enabled yet. The role is not enabled until the block is executed.

The DBMS_SQL package, however, is not subject to this restriction. When you use this package, the security checks are performed at runtime. Thus, a call to SET_ROLE would affect the SQL executed using calls to the DBMS_SQL package. The following block would therefore be successful:

```
DECLARE
    n NUMBER
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct');
    SYS.DBMS_SQL.PARSE
        ('SELECT empno FROM JOE.FINANCE');
    . . .
    --other calls to SYS.DBMS_SQL
    . . .
END;
```

Restricting Application Roles from Tool Users

Prebuilt database applications explicitly control the potential actions of a user, including the enabling and disabling of the user's roles while using the application. Alternatively, ad hoc query tools such as SQL*Plus allow a user to submit any SQL statement (which may or may not succeed), including the enabling and disabling of any granted role. This can pose a serious security problem. If you do not take precautions, an application user could have the ability to intentionally or unintentionally issue destructive SQL statements against database tables while using an ad hoc tool, using the privileges obtained through an application role.

For example, consider the following scenario:

- The Vacation application has a corresponding VACATION role.
- The VACATION role includes the privileges to issue SELECT, INSERT, UPDATE, and DELETE statements against the EMP table.
- The Vacation application controls the use of the privileges obtained via the VACATION role; that is, the application controls when statements are issued.

Now consider a user who has been granted the VACATION role. However, instead of using the Vacation application, the user executes SQL*Plus. At this point, the user is restricted only by the privileges granted to him explicitly or via roles, including the VACATION role. Because SQL*Plus is an ad hoc query tool, the user is not restricted to a set of predefined actions, as with designed database applications. The user can query or modify data in the EMP table as he or she chooses.

To avoid potential problems like the one above, consider the following policy for application roles:

1. Each application should have distinct roles:
 - One role should contain **all** privileges necessary to use the application successfully. Depending on the situation, there might be several roles that contain more or fewer privileges to provide tighter or less restrictive security while executing the application. Each application role should be protected by a password (or by operating system authentication) to prevent unauthorized use.
 - Another role should contain only non-destructive privileges associated with the application (that is, SELECT privileges for specific tables or views associated with the application). The read-only role allows the application user to generate custom reports using ad hoc tools such as SQL*Plus, SQL*ReportWriter, SQL*Graph, etc. However, this role does not allow the application user to modify table data outside the application itself. A role designed for an ad hoc query tool may or may not be protected by a password (or operating system authentication).

2. At startup, each application should use the SET ROLE command to enable one of the application roles associated with that application. If a password is used to authorize the role, the password must be included in the SET ROLE statement within the application (encrypted by the application, if possible); if the role is authorized by the operating system, the system administrator must have set up user accounts and applications so that application users get the appropriate operating system privileges when using the application.
3. At termination, each application should disable the previously enabled application role.
4. Application users should be granted application roles, as required. The administrator can prohibit a user from using application data with ad hoc tools by not granting the non-destructive role to the user.

Using this configuration, each application enables the proper role when the application is started, and disables the role when the application terminates. If an application user decides to use an ad hoc tool, the user can only enable the non-destructive role intended for that tool.

Additionally, you can

- Specify the roles to enable when a user starts SQL*Plus, using the PRODUCT_USER_PROFILE table. This functionality is similar to that of a precompiler or OCI application that issues a SET ROLE statement to enable specific roles upon application startup.
- Disable the use of the SET ROLE command for SQL*Plus users with the PRODUCT_USER_PROFILE table. This allows a SQL*Plus user only the privileges associated with the roles enabled when the user started SQL*Plus.

Other ad hoc query and reporting tools, such as SQL*ReportWriter, SQL*Graph, etc., can also make use of the PRODUCT_USER_PROFILE table to restrict the roles and commands that each user can use while running that product. For more information about these features, see the appropriate tool manual.

Schemas

Each database username is said to be a *schema*—a security domain that can contain schema objects. The access to the database and its objects is controlled by the privileges granted to each schema.

Most schemas can be thought of as usernames—the accounts set up to allow users to connect to a database and access the database's objects. However, *unique schemas* do not allow connections to the database, but are used to contain a related set of objects. Schemas of this sort are created as normal users, yet not granted the CREATE SESSION system privilege (either explicitly or via a role). However, you must temporarily grant the CREATE SESSION privilege to such schemas if you want to use the CREATE SCHEMA command to create multiple tables and views in a single transaction.

For example, the schema objects for a specific application might be owned by a schema. Application users can connect to the database using typical database usernames and use the application and the corresponding objects if they have the privileges to do so. However, no user can connect to the database using the schema set up for the application, thereby preventing access to the associated objects via this schema. This security configuration provides another layer of protection for objects.

Managing Privileges and Roles

As part of designing your application, you need to determine the types of users who will be working with the application and the level of access that they must be granted to accomplish their designated tasks. You must categorize these users into role groups and then determine the privileges that must be granted to each role.

Typically, end users are granted object privileges. An object privilege allows a user to perform a particular action on a specific table, view, sequence, procedure, function, or package. Depending on the type of object, there are different types of object privileges. Table 13 – 1 summarizes the object privileges available for each type of object.

Object Privilege	Table	View	Sequence	Procedure ⁽¹⁾
ALTER	✓		✓	
DELETE	✓	✓		
EXECUTE				✓
INDEX	✓ (2)			
INSERT	✓	✓		
REFERENCES	✓ (2)			
SELECT	✓	✓ (3)	✓	
UPDATE	✓	✓		

Table 13 – 1 Object Privileges

1 “Procedure” refers to standalone stored procedures, functions, and public package constructs.

2 Privilege cannot be granted to a role.

3 Can also be granted for snapshots.

Table 13 – 2 lists the SQL statements permitted by the object privileges listed in Table 13 – 1.

As you implement and test your application, you should create each of these roles and test the usage scenario for each role to be certain that the users of your application will have proper access to the database. After completing your tests, you should coordinate with the administrator of the application to ensure that each user is assigned the proper roles.

Object Privilege	SQL Statements Permitted
ALTER	ALTER object (table or sequence) CREATE TRIGGER ON object (tables only)
DELETE	DELETE FROM object (table or view)
EXECUTE	EXECUTE object (procedure or function) References to public package variables
INDEX	CREATE INDEX ON object (table or view)
INSERT	INSERT INTO object (table or view)
REFERENCES	CREATE or ALTER TABLE statement defining a FOREIGN KEY integrity constraint on object (tables only)
SELECT	SELECT...FROM object (table, view, or snapshot) SQL statements using a sequence

Table 13 – 2 SQL Statements Permitted by Database Object Privileges

Creating a Role

The use of a role can be protected by an associated password, as in the example below.

```
CREATE ROLE clerk IDENTIFIED BY bicentennial;
```

If you are granted a role protected by a password, you can enable or disable the role only by supplying the proper password for the role using a SET ROLE command (see page 13 – 12 for more information). Alternatively, roles can be created so that role use is authorized using information from the operating system. For more information about use of the operating system for role authorization, see the *Oracle7 Server Administrator's Guide*.

If a role is created without any protection, the role can be enabled or disabled by any grantee.

Database applications usually use the role authorization feature to specifically enable an application role and disable all other roles of a user. This way, the user cannot use privileges (from a role) intended for another application. With ad hoc query tools (such as SQL*Plus or Server Manager), users can explicitly enable only the roles for which they are authorized (that is, they know the password or are authorized by the operating system). See page 13 – 5 for more information.

When you create a new role, the name that you use must be unique among existing usernames and role names of the database. Roles are not contained in the schema of any user.

Immediately after creation, a role has no privileges associated with it. To associate privileges with a new role, you must grant privileges or other roles to the newly created role.

Privileges Required to Create Roles

To create a role, you must have the CREATE ROLE system privilege.

Enabling and Disabling Roles

Although a user can be granted a role, the role must be enabled before the privileges associated with it become available in the user's current session. Some, all, or none of the user's roles can be enabled or disabled. The following sections discuss when roles should be enabled and disabled and the different ways that a user can have roles enabled or disabled.

When to Enable Roles

In general, a user's security domain should always permit the user to perform the current task at hand, yet limit the user from having unnecessary privileges for the current job. For example, a user should have all the privileges to work with the database application currently in use, but not have any privileges required for any other database applications. Having too many privileges might allow users to access information through unintended methods.

Privileges granted directly to a user are always available to the user; therefore, directly granted privileges cannot be selectively enabled and disabled depending on a user's current task. Alternatively, privileges granted to a role can be selectively made available for any user granted the role. The enabling of roles **never** affects privileges explicitly granted to a user. The following sections explain how a user's roles can be selectively enabled (and disabled).

Default Roles

A default role is one that is automatically enabled for a user when the user creates a session. A user's list of default roles should include those roles that correspond to his or her typical job function.

Each user has a list of zero, or one or more default roles. Any role directly granted to a user can potentially be a default role of the user; an indirectly granted role (a role that is granted to a role) cannot be a default role; only directly granted roles can be default roles of a user.

The number of default roles for a user should not exceed the maximum number of enabled roles that are allowed per user (as specified by the initialization parameter `MAX_ENABLED_ROLES`); if the number of default roles for a user exceeds this maximum, errors are returned when the user attempts a connection, and the user's connection is not allowed.

Note: A default role is automatically enabled for a user when the user creates a session. Placing a role in a user's list of default roles bypasses authentication for the role, whether it is authorized using a password or the operating system.

A user's list of default roles can be set and altered using the SQL command `ALTER USER`. If the user's list of default roles is specified as `ALL`, every role granted to a user is automatically added to the user's list of default roles. Only subsequent modification of a user's default role list can remove newly granted roles from a user's list of default roles.

Modifications to a user's default role list only apply to sessions created after the alteration or role grant; neither method applies to a session in progress at the time of the user alteration or role grant.

Explicitly Enabling Roles

A user (or application) can explicitly enable a role using the SQL command SET ROLE. A SET ROLE statement enables all specified roles, provided that they have been granted to the user. All roles granted to the user that are not explicitly specified in a SET ROLE statement are disabled, including any roles previously enabled.

When you enable a role that contains other roles, all the indirectly granted roles are specifically enabled. Each indirectly granted role can be explicitly enabled or disabled for a user.

If a role is protected by a password, the role can only be enabled by indicating the role's password in the SET ROLE statement. If the role is not protected by a password, the role can be enabled with a simple SET ROLE statement. For example, assume that Morris' security domain is as follows:

- He is granted three roles: PAYROLL_CLERK (password BICENTENNIAL), ACCTS_PAY (password GARFIELD), and ACCTS_REC (identified externally). The PAYROLL_CLERK role includes the indirectly granted role PAYROLL_REPORT (identified externally).
- His only default role is PAYROLL_CLERK.

Morris' currently enabled roles can be changed from his default role, PAYROLL_CLERK, to ACCTS_PAY and ACCTS_REC, by the following statements:

```
SET ROLE accts_pay IDENTIFIED BY garfield, accts_rec;
```

Notice in the first statement that multiple roles can be enabled in a single SET ROLE statement. The ALL and ALL EXCEPT options of the SET ROLE command also allow several roles granted directly to the user to be enabled in one statement:

```
SET ROLE ALL EXCEPT payroll_clerk;
```

This statement shows the use of the ALL EXCEPT option of the SET ROLE command. Use this option when you want to enable most of a user's roles and only disable one or more. Similarly, all of Morris' roles can be enabled by the following statement:

```
SET ROLE ALL;
```

When using the ALL or ALL EXCEPT options of the SET ROLE command, all roles to be enabled either must not require a password, or must be authenticated using the operating system. If a role requires a password, the SET ROLE ALL or ALL EXCEPT statement is rolled back and an error is returned.

A user can also explicitly enable any indirectly granted roles granted to him or her via an explicit grant of another role. For example, Morris can issue the following statement:

```
SET ROLE payroll_report;
```

Privileges Required to Explicitly Enable Roles Any user can use the SET ROLE command to enable any granted roles, provided the grantee supplies role passwords, when necessary.

Enabling and Disabling Roles When OS_ROLES=TRUE

If OS_ROLES is set to TRUE, any role granted by the operating system can be dynamically enabled using the SET ROLE command. However, any role not identified in a user's operating system account cannot be specified in a SET ROLE statement (it is ignored), even if a role has been granted using a GRANT statement.

When OS_ROLES is set to TRUE, a user can enable as many roles as specified by the initialization parameter MAX_ENABLED_ROLES. For more information about use of the operating system for role authorization, see the *Oracle7 Server Administrator's Guide*.

Dropping Roles

When you drop a role, the security domains of all users and roles granted that role are immediately changed to reflect the absence of the dropped role's privileges. All indirectly granted roles of the dropped role are also removed from affected security domains. Dropping a role automatically removes the role from all users' default role lists.

Because the creation of objects is not dependent upon the privileges received via a role, no cascading effects regarding objects need to be considered when dropping a role (for example, tables or other objects are not dropped when a role is dropped).

Drop a role using the SQL command DROP ROLE, as shown in the following example.

```
DROP ROLE clerk;
```


Privileges Required to Drop Roles

To drop a role, you must have the DROP ANY ROLE system privilege or have been granted the role with the ADMIN OPTION.

Granting and Revoking Privileges and Roles

The following sections explain how to grant and revoke system privileges, roles, and object privileges.

Granting System Privileges and Roles

System privileges and roles can be granted to other roles or users using the SQL command GRANT, as shown in the following example:

```
GRANT create session, accts_pay  
    TO jward, finance;
```

Object privileges cannot be granted along with system privileges and roles in the same GRANT statement.

The ADMIN OPTION A system privilege or role can be granted with the ADMIN OPTION. (This option is not valid when granting a role to another role.) A grantee with this option has several expanded capabilities:

- The grantee can grant or revoke the system privilege or role to or from **any** user or other role in the database. (A user cannot revoke a role from himself.)
- The grantee can further grant the system privilege or role with the ADMIN OPTION.
- The grantee of a role can alter or drop the role.

A grantee without the ADMIN OPTION cannot perform the above operations.

When a user creates a role, the role is automatically granted to the creator with the ADMIN OPTION.

Assume that you grant the NEW_DBA role to MICHAEL with the following statement:

```
GRANT new_dba TO michael WITH ADMIN OPTION;
```

The user MICHAEL cannot only use all of the privileges implicit in the NEW_DBA role, but can grant, revoke, or drop the NEW_DBA role as necessary.

Privileges Required to Grant System Privileges or Roles To grant a system privilege or role, the grantor requires the ADMIN OPTION for all system privileges and roles being granted. Additionally, any user with the GRANT ANY ROLE system privilege can grant any role in a database.

Granting Object Privileges

Grant object privileges to roles or users using the SQL command GRANT. The following statement grants the SELECT, INSERT, and DELETE object privileges for all columns of the EMP table to the users JWARD and TSMITH:

```
GRANT select, insert, delete ON emp TO jward, tsmith;
```

To grant the INSERT object privilege for only the ENAME and JOB columns of the EMP table to the users JWARD and TSMITH, enter the following statement:

```
GRANT insert(ename, job) ON emp TO jward, tsmith;
```

To grant all object privileges on the SALARY view to the user WALLEN, use the ALL short cut, as in

```
GRANT ALL ON salary TO wallen;
```

System privileges and roles cannot be granted along with object privileges in the same GRANT statement.

The GRANT OPTION An object privilege can be granted to a user with the GRANT OPTION. This special privilege allows the grantee several expanded privileges:

- The grantee can grant the object privilege to any user or any role in the database.
- The grantee can also grant the object privilege to other users, with or without the GRANT OPTION.
- If the grantee receives object privileges for a table with the GRANT OPTION and the grantee has the CREATE VIEW or the CREATE ANY VIEW system privilege, the grantee can create views on the table and grant the corresponding privileges on the view to any user or role in the database.

The user whose schema contains an object is automatically granted all associated object privileges with the GRANT OPTION.

Specifically note that the GRANT OPTION is not valid when granting an object privilege to a role. Oracle prevents the propagation of object privileges via roles so that grantees of a role cannot propagate object privileges received via roles.

Privileges Required to Grant Object Privileges To grant an object privilege, the grantor must either

- be the owner of the object being specified, or
- have been granted the object privileges being granted with the GRANT OPTION

Revoking System Privileges and Roles

System privileges and roles can be revoked using the SQL command REVOKE, as shown in the following example:

```
REVOKE create table, accts_rec FROM tsmith, finance;
```

The ADMIN OPTION for a system privilege or role cannot be selectively revoked; the privilege or role must be revoked and then the privilege or role regranted without the ADMIN OPTION.

Privileges Required to Revoke System Privileges and Roles Any user with the ADMIN OPTION for a system privilege or role can revoke the privilege or role from any other database user or role (the user does not have to be the user that originally granted the privilege or role). Additionally, any user with the GRANT ANY ROLE can revoke **any** role.

Revoking Object Privileges

Object privileges can be revoked using the SQL command REVOKE. For example, assuming you are the original grantor, to revoke the SELECT and INSERT privileges on the EMP table from the users JWARD and TSMITH, enter the following statement:

```
REVOKE select, insert ON emp  
FROM jward, tsmith;
```

A grantor could also revoke all privileges on the table DEPT (even if only one privilege was granted) that he or she granted to the role HUMAN_RESOURCES by entering the following statement:

```
REVOKE ALL ON dept FROM human_resources;
```

This statement would only revoke the privileges that the grantor authorized, not the grants made by other users. The GRANT OPTION for an object privilege cannot be selectively revoked; the object privilege must be revoked and then regranted without the GRANT OPTION. A user cannot revoke object privileges from him or herself.

Revoking Column-Selective Object Privileges Recall that column-specific INSERT, UPDATE, and REFERENCES privileges can be granted for tables or views; however, it is not possible to revoke column-specific privileges selectively with a similar REVOKE statement. Instead, the grantor must first revoke the object privilege for all columns of a table or view, and then selectively grant the new column-specific privileges again.

For example, assume the role HUMAN_RESOURCES has been granted the UPDATE privilege on the DEPTNO and DNAME columns of the table DEPT. To revoke the UPDATE privilege on just the DEPTNO column, enter the following two statements:

```
REVOKE UPDATE ON dept FROM human_resources;  
GRANT UPDATE (dname) ON dept TO human_resources;
```

The REVOKE statement revokes the UPDATE privilege on all columns of the DEPT table from the role HUMAN_RESOURCES. The GRANT statement regrants the UPDATE privilege on the DNAME column to the role HUMAN_RESOURCES.

Revoking the REFERENCES Object Privilege If the grantee of the REFERENCES object privilege has used the privilege to create a foreign key constraint (that currently exists), the grantor can only revoke the privilege by specifying the CASCADE CONSTRAINTS option in the REVOKE statement:

```
REVOKE REFERENCES ON dept FROM jward CASCADE CONSTRAINTS;
```

Any foreign key constraints currently defined that use the revoked REFERENCES privilege are dropped when the CASCADE CONSTRAINTS option is specified.

Privileges Required to Revoke Object Privileges To revoke an object privilege, the revoker must be the original grantor of the object privilege being revoked.

Cascading Effects of Revoking Privileges

Depending on the type of privilege, there may or may not be cascading effects if a privilege is revoked. The following sections explain several cascading effects.

System Privileges There are no cascading effects when revoking a system privilege related to DDL operations, regardless of whether the privilege was granted with or without the ADMIN OPTION. For example, assume the following:

1. You grant the CREATE TABLE system privilege to JWARD with the WITH OPTION.
2. JWARD creates a table.
3. JWARD grants the CREATE TABLE system privilege to TSMITH.
4. TSMITH creates a table.
5. You revoke the CREATE TABLE system privilege from JWARD.
6. JWARD's table continues to exist. TSMITH continues to have the CREATE TABLE system privilege and his table still exists.

Cascading effects can be observed when revoking a system privilege related to a DML operation. For example, if SELECT ANY TABLE is granted to a user, and that user has created any procedures, all procedures contained in the user's schema must be reauthorized before they can be used again (after the revoke).

Object Privileges Revoking an object privilege can have several types of cascading effects that should be investigated before a REVOKE statement is issued:

- Object definitions that depend on a DML object privilege can be affected if the DML object privilege is revoked. For example, assume the procedure body of the TEST procedure includes a SQL statement that queries data from the EMP table. If the SELECT privilege on the EMP table is revoked from the owner of the TEST procedure, the procedure can no longer be executed successfully.
- Object definitions that require the ALTER and INDEX DDL object privileges are not affected if the ALTER or INDEX object privilege is revoked. For example, if the INDEX privilege is revoked from a user that created an index on someone else's table, the index continues to exist after the privilege is revoked.

- When a REFERENCES privilege for a table is revoked from a user, any foreign key integrity constraints defined by the user that require the dropped REFERENCES privilege are automatically dropped. For example, assume that the user JWARD is granted the REFERENCES privilege for the DEPTNO column of the DEPT table and creates a foreign key on the DEPTNO column in the EMP table that references the DEPTNO column. If the REFERENCES privilege on the DEPTNO column of the DEPT table is revoked, the foreign key constraint on the DEPTNO column of the EMP table is dropped in the same operation.
- The object privilege grants propagated using the GRANT OPTION are revoked if a grantor's object privilege is revoked. For example, assume that USER1 is granted the SELECT object privilege with the GRANT OPTION, and grants the SELECT privilege on EMP to USER2. Subsequently, the SELECT privilege is revoked from USER1. This revoke is cascaded to USER2 as well. Any objects that depended on USER1's and USER2's revoked SELECT privilege can also be affected.

Granting to, and Revoking from, the User Group PUBLIC

Privileges and roles can also be granted to and revoked from the user group PUBLIC. Because PUBLIC is accessible to every database user, all privileges and roles granted to PUBLIC are accessible to every database user.

You should only grant a privilege or role to PUBLIC if every database user requires the privilege or role. This recommendation restates the general rule that at any given time, each database user should only have the privileges required to successfully accomplish the current task.

Revokes from PUBLIC can cause significant cascading effects, depending on the privilege that is revoked. If any privilege related to a DML operation is revoked from PUBLIC (for example, SELECT ANY TABLE, UPDATE ON emp), all procedures in the database (including functions and packages) must be reauthorized before they can be used again. Therefore, use caution when granting DML-related privileges to PUBLIC.

When Do Grants and Revokes Take Effect?

Depending upon what is granted or revoked, a grant or revoke takes effect at different times:

- All grants/revokes of privileges (system and object) to anything (users, roles, and PUBLIC) are immediately observed.
- All grants/revokes of roles to anything (users, other roles, PUBLIC) are only observed when a current user session issues a SET ROLE statement to re-enable the role after the grant/revoke, or when a new user session is created after the grant/revoke.

How Do Grants Affect Dependent Objects?

Issuing a GRANT statement against a database object causes the “last DDL time” attribute of the object to change. This can invalidate any dependent objects, in particular PL/SQL package bodies that refer to the object. These then must be recompiled.

Index

Symbols

%ROWTYPE attribute, 7 – 7, 11 – 4
 used in stored functions, 7 – 8
%TYPE attribute, 7 – 7, 11 – 4

A

access

 database

 granting privileges, 13 – 14
 revoking privileges, 13 – 16

 objects

 remote integrity constraints and, 6 – 13
 sequences, 4 – 26
 triggers, 9 – 3, 9 – 33 to 9 – 36

access, object

 granting privileges, 13 – 15
 revoking privileges, 13 – 16

adding labels, 5 – 14

ADMIN option, 13 – 14

AFTER triggers

 auditing and, 9 – 22, 9 – 24
 correlation names and, 9 – 7
 specifying, 9 – 4

alerters, 12 – 2

ALL_ERRORS view, 7 – 35

 for debugging stored procedures, 7 – 35

ALL_LABELS view, 5 – 14

allocation, extents, 4 – 41

ALTER CLUSTER command, 4 – 5

 ALLOCATE EXTENT option, 4 – 41

ALTER FUNCTION command, 11 – 5

ALTER INDEX command, 4 – 5

ALTER PACKAGE command, 11 – 5

ALTER PROCEDURE command, 11 – 5

ALTER SEQUENCE command, 4 – 26

ALTER TABLE command, 4 – 5, 4 – 8

 defining integrity constraints, 6 – 16

 DISABLE ALL TRIGGERS option, 9 – 19

 DISABLE integrity constraint option, 6 – 21

 DROP integrity constraint option, 6 – 25

 ENABLE ALL TRIGGERS option, 9 – 19

 ENABLE integrity constraint option, 6 – 21

 INITRANS parameter in, 3 – 30

ALTER TRIGGER command, 11 – 6

 DISABLE option, 9 – 18

 ENABLE option, 9 – 19

ALTER VIEW command, 11 – 5

ALTER_COMPILE procedure, 7 – 53

altering

 storage parameters, 4 – 8

 tables, 4 – 8

American National Standards Institute (ANSI),

 ANSI-compatible locking, 3 – 16

anonymous PL/SQL blocks

 about, 7 – 2, 10 – 2

 compared to triggers, 7 – 4

 dynamic SQL and, 10 – 3

ANSI SQL92, FIPS flagger, 3 – 2

- applications
 - calling stored procedures and packages, 7 – 38
 - designing, 2 – 2, 2 – 4
 - designing database, 2 – 2
 - maintaining, 2 – 10
 - roles, 13 – 3
 - security and, 13 – 2 to 13 – 8
 - tuning, 2 – 8
 - unhandled exceptions in, 7 – 32
- arrays of C structs, 2 – 7
- auditing, triggers and, 9 – 21 to 9 – 36

B

- BEFORE triggers
 - correlation names and, 9 – 7
 - specifying, 9 – 4
- BIND_VARIABLE procedure, 10 – 5
- BIND_VARIABLE procedures, 10 – 10
- blank padding data,
 - performance considerations, 5 – 2
- body, triggers, 9 – 7 to 9 – 36
- business rules, 1 – 2

C

- CACHE option,
 - CREATE SEQUENCE command, 4 – 30
- caches
 - sequence cache, 4 – 29
 - sequence numbers, 4 – 25
- CASCADE option, integrity constraints, 4 – 42
- CASE tools, 1 – 2
- CATPROC.SQL file, 3 – 20, 8 – 22, 9 – 3
- CHAR datatype, 5 – 3
 - increasing column length, 4 – 8
 - when to use, 5 – 2
- CHARARR datatype,
 - in DBMS_OUTPUT, 8 – 25
- CHECK constraint
 - data integrity, 6 – 20
 - designing, 6 – 14
 - NOT NULL constraint and, 6 – 15

- CHECK constraint *continued*
 - number of, 6 – 15
 - restricting nulls using, 6 – 15
 - restrictions on, 6 – 14
 - triggers and, 9 – 32 to 9 – 36
 - when to use, 6 – 13
- client-side development tools, 1 – 2
- CLOSE_CURSOR procedure, 10 – 6, 10 – 20
- CLOSE_DATABASE_LINK procedure, 7 – 53
- clusters
 - allocating extents, 4 – 41
 - choosing data, 4 – 38, 4 – 39
 - creating, 4 – 40
 - dropped tables and, 4 – 9
 - dropping, 4 – 41
 - index creation, 4 – 40
 - integrity constraints and, 4 – 40
 - keys, 4 – 38
 - performance considerations, 4 – 39
 - privileges for creating, 4 – 41
- COLUMN_VALUE procedure, 10 – 16
- COLUMN_VALUE_LONG procedure,
 - 10 – 6, 10 – 17
- columns
 - accessing in triggers, 9 – 7 to 9 – 36
 - default values, 6 – 4
 - generating derived values with triggers, 9 – 34
 - granting privileges for selected, 13 – 15
 - increasing length, 4 – 8
 - listing in an UPDATE trigger, 9 – 5, 9 – 9
 - multiple FOREIGN KEY constraints, 6 – 10
 - number of CHECK constraints limit, 6 – 15
 - revoking privileges from, 13 – 17
- comments, on Oracle documentation, vi
- COMMIT command, 3 – 5
- communication, between sessions, 8 – 2
- comparison operators, comparing labels, 5 – 24
- COMPILE option,
 - of ALTER PROCEDURE command, 11 – 5
- compliance with industry standards, 2 – 7
- composite keys, restricting nulls in, 6 – 15
- concurrency, 3 – 27

- conditional predicates
 - in trigger bodies, 9 – 7
 - trigger bodies, 9 – 8 to 9 – 36
- consistency, read-only transactions, 3 – 8
- constraining tables, 9 – 12
- constraints, composite UNIQUE keys, 6 – 6
- conventions, used in this Guide, iv
- conversion functions
 - TO_CHAR function, 5 – 22
 - TO_LABEL function, 5 – 22
- converting data, 5 – 19
- correlation names, 9 – 6, 9 – 7, 9 – 8
 - NEW, 9 – 7
 - OLD, 9 – 7
 - REFERENCING option and, 9 – 8
 - when preceded by a colon, 9 – 8
- CREATE CLUSTER command, 4 – 5, 4 – 40
 - hash clusters, 4 – 43
 - HASH IS option, 4 – 44
 - HASHKEYS option, 4 – 44
- CREATE INDEX command, 4 – 5, 4 – 37
 - ON CLUSTER option, 4 – 40
- CREATE PACKAGE BODY command, 7 – 13
- CREATE PACKAGE command, 7 – 13
- CREATE ROLE command, 13 – 10
- CREATE SCHEMA command, 4 – 47
 - privileges required, 4 – 47
- CREATE SEQUENCE command
 - CACHE option, 4 – 25
 - CACHE parameter, 4 – 30
 - examples, 4 – 30
 - NOCACHE option, 4 – 30
- CREATE TABLE command, 4 – 2, 4 – 3, 4 – 5
 - CLUSTER option, 4 – 40
 - defining integrity constraints, 6 – 16
 - INITRANS parameter in, 3 – 30
- CREATE TRIGGER command, 9 – 3
 - REFERENCING option, 9 – 8
- CREATE VIEW command, 4 – 10
 - OR REPLACE option, 4 – 13
 - WITH CHECK OPTION, 4 – 11, 4 – 14
- CREATE_PIPE procedure, 8 – 4

- creating
 - clusters, 4 – 40
 - hash clusters, 4 – 43
 - indexes, 4 – 36
 - integrity constraints, 6 – 2
 - multiple objects, 4 – 47
 - packages, 7 – 13
 - sequences, 4 – 30
 - synonyms, 4 – 31
 - tables, 4 – 2, 4 – 3
 - triggers, 9 – 3 to 9 – 15
 - views, 4 – 10
- CURRVAL pseudo-column, 4 – 27
 - restrictions, 4 – 28
- cursor variables, 7 – 26
 - declaring and opening, 7 – 26
- cursors, 3 – 9
 - closing, 3 – 10
 - maximum number of, 3 – 9
 - pointers to, 7 – 26
 - private SQL areas and, 3 – 9

D

- data blocks, factors affecting size of, 4 – 5
- data conversion, 5 – 19
- data dictionary
 - dropped tables and, 4 – 9
 - integrity constraints in, 6 – 28
 - schema object views, 4 – 50
- database
 - administrator,
 - application administrator vs., 13 – 2
 - designing, 2 – 2
 - global name in a distributed system, 4 – 48
 - normalizing, 2 – 3
 - security
 - applications and, 13 – 2
 - schemas and, 13 – 8
 - triggers, using in applications, 2 – 5
- datatypes
 - ANSI, 5 – 18
 - CHAR, 5 – 3
 - column lengths for character types, 5 – 4
 - DATE, 5 – 5

datatypes continued

- DB2, 5 – 18
- LONG, 5 – 7
- LONG RAW, 5 – 9
- NUMBER, 5 – 4
- PL/SQL, numeric codes for, 7 – 64
- RAW, 5 – 9
- ROWID, 5 – 9
- SQL/DS, 5 – 18
- summary of, 5 – 13
- VARCHAR, 5 – 3
- VARCHAR2, 5 – 3

date arithmetic, 5 – 6

DATE datatype, 5 – 5

DBA_ERRORS view, 7 – 35

- for debugging stored procedures, 7 – 35

DBA_ROLE_PRIVS view, 13 – 3

DBMS labels

- DBMS MAC mode
 - displaying, 5 – 16
 - retrieving, 5 – 16
- OS MAC mode
 - displaying, 5 – 16
 - retrieving, 5 – 16

DBMS_ALERT package, 7 – 56

- about, 12 – 2
- creating, 12 – 3

DBMS_APPLICATION_INFO package, 7 – 56

DBMS_DEFER package, 7 – 57

DBMS_DEFER_QUERY package, 7 – 57

DBMS_DEFER_SYS package, 7 – 57

DBMS_DESCRIBE package, 7 – 56, 7 – 57

- creating, 7 – 57

DBMS_JOB package, 7 – 56

DBMS_LOCK package, 3 – 18, 3 – 19, 7 – 56

- creating, 3 – 20
- security, 3 – 19

DBMS_OUTPUT package, 7 – 56, 8 – 21

- creating, 8 – 22
- examples, 8 – 25
- GET_LINE procedure, 8 – 21
- NEW_LINE procedure, 8 – 21
- PUT procedure, 8 – 21
- PUT_LINE procedure, 8 – 21

DBMS_PIPE package, 7 – 56, 8 – 2

- creating, 8 – 3

DBMS_REFRESH package, 7 – 56

DBMS_REPCAT package, 7 – 57

DBMS_REPCAT_ADMIN package, 7 – 57

DBMS_REPCAT_AUTH package, 7 – 57

DBMS_SHARED_POOL package, 7 – 56

DBMS_SNAPSHOT package, 7 – 57

DBMS_SPACE package, 7 – 56

DBMS_SQL functions, 10 – 5 to 10 – 6

DBMS_SQL package, 7 – 56, 10 – 2

- creating, 10 – 2

DBMS_SYSTEM package, 7 – 56

DBMSALRT.SQL file, 12 – 3

DBMSDESC.SQL file, 7 – 57

DBMSLOCK.SQL file, 3 – 19, 3 – 20

DBMSOTPT.SQL file, 8 – 22

DBMSPIPE.SQL file, 8 – 3

DBMSSQL.SQL file, 10 – 2

DDL statements,

- package state and, 7 – 15

debugging

- stored procedures, 7 – 36
- triggers, 9 – 17

DECLARE,

- not used in stored procedures, 7 – 9

default

- column values, 6 – 4
- maximum savepoints, 3 – 6
- PCTFREE option, 4 – 4
- PCTUSED option, 4 – 5
- role, 13 – 11

DEFINE_COLUMN procedure, 10 – 5, 10 – 12

DEFINE_COLUMN_LONG procedure, 10 – 6, 10 – 14

DELETE command

- column values and triggers, 9 – 7
- data consistency, 3 – 11
- triggers for referential integrity, 9 – 28, 9 – 29

- dependencies
 - among PL/SQL library objects, 7 – 16
 - in stored triggers, 9 – 16
 - listing information about, 11 – 6 to 11 – 11
 - object
 - trigger management, 9 – 11
 - UTLDTREE.SQL, 11 – 7
 - the release 7.2 model, 7 – 16
 - the timestamp model, 7 – 16
- DESCRIBE PROCEDURE procedure, 7 – 59
- Designer/2000, 1 – 2
- designing applications, 2 – 4
 - assessing needs, 2 – 2
- Developer/2000, 1 – 2
- DISABLE procedure, 8 – 21, 8 – 23
- disabling
 - integrity constraints, 6 – 19
 - triggers, 9 – 18 to 9 – 36
- distributed databases
 - referential integrity and, 6 – 13
 - remote stored procedures, 7 – 40, 7 – 41
 - triggers and, 9 – 11
- distributed queries, 7 – 33 to 7 – 36
- distributed transactions,
 - LOCK TABLE command, 3 – 12
- DISTRIBUTED_LOCK_TIMEOUT parameter,
 - LOCK TABLE command and, 3 – 12
- DROP CLUSTER command, 4 – 42, 4 – 44
- DROP INDEX command, 4 – 37
 - privileges required, 4 – 38
- DROP ROLE command, 13 – 13
- DROP TABLE command, 4 – 9
- DROP TRIGGER command, 9 – 18
- dropping
 - clusters, 4 – 41
 - hash clusters, 4 – 44
 - indexes, 4 – 37
 - integrity constraints, 6 – 25
 - packages, 7 – 14
 - procedures, 7 – 11
 - roles, 13 – 13
 - sequences, 4 – 31
 - tables, 4 – 9
 - triggers, 9 – 18
 - views, 4 – 15

- dynamic SQL
 - anonymous blocks and, 10 – 3
 - DBMS_SQL functions, using, 10 – 2
 - DBMS_SQL package, 10 – 2 to 10 – 24
 - errors, locating, 10 – 21
 - examples, 10 – 22 to 10 – 25
 - execution flow in, 10 – 4 to 10 – 6
 - LAST_ERROR_POSITION function,
 - 10 – 21 to 10 – 25
 - LAST_ROW_COUNT function,
 - 10 – 21 to 10 – 25
 - LAST_ROW_ID function, 10 – 21 to 10 – 25
 - LAST_SQL_FUNCTION_CODE function,
 - 10 – 21 to 10 – 25
 - security, 10 – 7

E

- embedded SQL, 7 – 2
- ENABLE procedure, 8 – 21, 8 – 22
- enabling
 - integrity constraints, at creation, 6 – 19
 - integrity constraints, 6 – 20
 - at creation, 6 – 18
 - reporting exceptions, 6 – 23
 - when violations exist, 6 – 19
 - roles, 13 – 12
 - triggers, 9 – 19 to 9 – 36
- Entity–Relationship model, 2 – 2
- errors
 - application errors raised by Oracle packages,
 - 7 – 30
 - creating views with errors, 4 – 12
 - locating in dynamic SQL, 10 – 21
 - remote procedures and, 7 – 33 to 7 – 36
 - returned by DBMS_ALERT package, 12 – 3
 - returned by DBMS_DESCRIBE package,
 - 7 – 58
 - returned by DBMS_OUTPUT, 8 – 22
 - returned by DBMS_PIPE package, 8 – 4
 - user-defined, 7 – 30, 7 – 31
- events, signalling with alerters, 12 – 2
- exception handlers, in PL/SQL, 7 – 2
- exceptions
 - defining, 7 – 31
 - during trigger execution, 9 – 9

exceptions *continued*

- effects on applications, 7 – 32
- remote procedures and, 7 – 33 to 7 – 36
- unhandled, 7 – 32

exclusive locks,

- LOCK TABLE command, 3 – 15

EXECUTE function, 10 – 6, 10 – 14

EXECUTE_AND_FETCH function, 10 – 15

execution flow, in dynamic SQL, 10 – 4

explicit locking, manual locking, 3 – 11

extents

- allocating, 4 – 41
- dropped tabled and, 4 – 9

F

FCLOSE procedure, 8 – 34

FCLOSE_ALL procedure, 8 – 34

features, 2 – 4

feedback, on Oracle documentation, vi

FETCH_ROWS function, 10 – 15

FETCH_ROWS procedure, 10 – 6

FFLUSH procedure, 8 – 40

file I/O, in PL/SQL, 8 – 28

file ownership,

- with the UTL_FILE package, 8 – 30

FIPS flagger,

- interactive SQL statements and, 3 – 2

FIXED_DATE parameter, 5 – 6

FOPEN function, 8 – 32

FOR EACH ROW clause, 9 – 5

FOREIGN KEY constraint

- defining, 6 – 26, 6 – 27
- enabling, 6 – 20, 6 – 27
- NOT NULL constraint and, 6 – 9
- number of rows referencing parent table, 6 – 9
- one-to-many relationship, 6 – 9
- one-to-one relationship, 6 – 9
- UNIQUE key constraint and, 6 – 9
- updating tables, 6 – 10, 6 – 11

G

GET_LINE procedure, 8 – 21, 8 – 24, 8 – 35

GET_LINES procedure, 8 – 21, 8 – 24

GRANT command, 13 – 14

- ADMIN option, 13 – 14
- object privileges, 13 – 15
- system privileges, 13 – 14
- when in effect, 13 – 20
- WITH GRANT option, 13 – 15

granting privileges and roles. *See* GRANT command

H

hash clusters

- choosing key, 4 – 44
- creating, 4 – 43
- dropping, 4 – 44
- root block, 4 – 43
- when to use, 4 – 43

hiding PL/SQL code, 7 – 29

I

IN OUT parameter mode, 7 – 6

IN parameter mode, 7 – 6

indexes

- creating, 4 – 36
- dropped tables and, 4 – 9
- dropping, 4 – 37
- guidelines, 4 – 34
- order of columns, 4 – 35
- privileges, 4 – 37
- specifying PCTFREE for, 4 – 5
- SQL*Loader and, 4 – 34
- temporary segments and, 4 – 34
- when to create, 4 – 33

industry standards compliance, 2 – 7

initialization part of package,

- avoiding problems with, 7 – 51

INITTRANS parameter, 3 – 30

- INSERT command
 - column values and triggers, 9 – 7
 - read consistency, 3 – 11
- integrity constraints
 - altering, 6 – 25
 - application uses, 6 – 2
 - clusters and, 4 – 40
 - defining, 6 – 15
 - disabling, 6 – 18, 6 – 20
 - dropping, 6 – 25
 - enabling, 6 – 18
 - enabling at creation, 6 – 18
 - enabling when violations exist, 6 – 19
 - examples, 6 – 2
 - exceptions to, 6 – 23
 - listing definitions of, 6 – 28
 - naming, 6 – 17
 - performance considerations, 6 – 3
 - privileges required for creating, 6 – 17
 - restrictions for adding or dropping, 6 – 16
 - triggers vs., 9 – 2, 9 – 25
 - using in applications, 2 – 4
 - violations, 6 – 19
 - when to disable, 6 – 19
 - when to use, 6 – 2
- interactive block execution, 7 – 37
- invalid views, 4 – 15
- IS_OPEN function, 8 – 33, 10 – 20
- IS_ROLE_ENABLED procedure, 7 – 53
- ISOLATION LEVEL
 - changing, 3 – 30
 - SERIALIZABLE, 3 – 30

J

- join view, 4 – 16
 - DELETE statements, 4 – 19
 - key-preserved tables in, 4 – 18
 - mergeable, 4 – 17
 - modifying, rule for, 4 – 18
 - UPDATE statements, 4 – 19
 - when modifiable, 4 – 17
- Julian dates, using, 5 – 6

K

- key-preserved tables
 - in join views, 4 – 18
 - in outer joins, 4 – 22
- keys
 - foreign keys, 6 – 25
 - unique, composite, 6 – 6

L

- labels
 - adding, 5 – 14
 - altering DBMS labels, 5 – 17
 - comparing, 5 – 24
 - converting, 5 – 22
 - defining, guidelines, 5 – 25
 - displaying DBMS labels, 5 – 16
 - formatting, 5 – 23
 - installation-specific component, 5 – 22
 - retrieving from table or view, 5 – 15
- LAST_ERROR_POSITION function,
 - 10 – 21 to 10 – 25
- LAST_ROW_COUNT function,
 - 10 – 21 to 10 – 25
- LAST_ROW_ID function, 10 – 21 to 10 – 25
- LAST_SQL_FUNCTION_CODE function,
 - 10 – 21 to 10 – 25
- local procedures, in a package body, 7 – 14
- LOCK TABLE command, 3 – 12, 3 – 13
- locking
 - application design and, 2 – 6
 - indexed foreign keys and, 6 – 11
 - manual (explicit), 3 – 11
 - unindexed foreign keys and, 6 – 10
- locks
 - distributed, 3 – 11
 - LOCK TABLE command, 3 – 12, 3 – 13
 - monitoring, 3 – 27
 - privileges for manual acquirement, 3 – 15
 - user locks, 3 – 18
 - UTLLOCKT.SQL script, 3 – 27

LONG datatype, 5 – 7
 use in triggers, 9 – 11
LONG RAW datatype, 5 – 9
 use in triggers, 9 – 11

M

maintaining applications, 2 – 10
manual locking, 3 – 11
 LOCK TABLE command, 3 – 12
MAX_ENABLED_ROLES parameter,
 default roles and, 13 – 11
MAXTRANS option, 4 – 5
messages, between sessions, 8 – 2
MLSLABEL datatype, 5 – 13
modes, of parameters, 7 – 6
modifiable join view, definition of, 4 – 17
mutating tables, 9 – 12

N

name resolution, 4 – 48
national language support, 2 – 6
NEW, correlation name, 9 – 7
NEW_LINE procedure, 8 – 21, 8 – 37
NEXT_ITEM_TYPE function, 8 – 9
NEXTVAL pseudo-column, 4 – 27
 restrictions, 4 – 28
NOCACHE option, CREATE SEQUENCE
 statement, 4 – 30
normalization, 2 – 3
NOT NULL constraint
 CHECK constraint and, 6 – 15
 data integrity, 6 – 20
 when to use, 6 – 3
NOWAIT option, 3 – 12
NUMBER datatype, 5 – 4

O

objects, schema
 granting privileges, 13 – 15
 listing information, 4 – 50
 name resolution, 4 – 48
 renaming, 4 – 50
 revoking privileges, 13 – 16
 when revoking object privileges, 13 – 18
OCI. *See* Oracle Call Interface
OLD, correlation name, 9 – 7
one-to-many relationship, with foreign keys,
 6 – 9
one-to-one relationship, with foreign keys,
 6 – 9
OPEN_CURSOR function, 10 – 5, 10 – 9
OPEN_CURSORS parameter, 3 – 9
operating system, roles and, 13 – 13
optimizer, using hints in applications, 2 – 5
OR REPLACE clause,
 for creating packages, 7 – 13
Oracle Call Interface, 7 – 2
 applications, 7 – 4
 new functionality in, 2 – 7
Oracle errors, 7 – 3
Oracle Precompilers, calling stored procedures
 and packages, 7 – 38
Oracle Procedure Builder, 1 – 2
Oracle-supplied packages,
 where documented, 1 – 4
OUT parameter mode, 7 – 6
outer joins, 4 – 21
 key-preserved tables in, 4 – 22
overloading
 of packaged functions, 7 – 52
 stored procedure names, 7 – 5
 using RESTRICT_REFERENCES, 7 – 52

P

- PACK_MESSAGE procedure, 8 – 6
- package. *See* PL/SQL package
- package body, 7 – 11
- package specification, 7 – 11
- packages
 - compilation,
 - avoiding runtime, 11 – 2 to 11 – 4
 - creating, 7 – 13
 - DBMS_OUTPUT, example of use, 7 – 3
 - dropping, 7 – 14
 - in PL/SQL, 7 – 11
 - listing information about, 7 – 65 to 7 – 67
 - minimizing object dependencies and, 11 – 3
 - naming of, 7 – 14
 - privileges, 11 – 5
 - privileges for execution, 7 – 39
 - privileges required to create, 7 – 13
 - privileges required to create procedures in, 7 – 10
 - recompiling, 11 – 4 to 11 – 6
 - session state and, 7 – 15
 - supplied by Oracle, 7 – 53
 - synonyms, 7 – 42
 - using in applications, 2 – 4
 - where documented, 1 – 4, 7 – 53
- parallel server
 - distributed locks, 3 – 11
 - sequence numbers and, 4 – 26
- parameter
 - default values, 7 – 9
 - with stored functions, 7 – 46
 - file, INIT.ORA, 8 – 29, 8 – 30
 - modes, 7 – 6
- PARSE procedure, 10 – 5, 10 – 9
- parse tree, 9 – 16
- pcode, when generated for triggers, 9 – 16
- PCTFREE storage parameter
 - altering, 4 – 8
 - block overhead and, 4 – 6
 - default, 4 – 4
 - guidelines for setting, 4 – 4, 4 – 6
 - indexes for, 4 – 5
 - non-clustered tables, 4 – 5
- PCTUSED storage parameter
 - altering, 4 – 8
 - block overhead and, 4 – 6
 - default, 4 – 5
 - guidelines for setting, 4 – 5, 4 – 6
 - non-clustered tables, 4 – 5
- performance
 - ROW_LOCKING parameter, 3 – 16
 - SERIALIZABLE parameter, 3 – 16
- performance, database
 - clusters and, 4 – 39
 - index column order, 4 – 35
- pipes
 - between sessions, 8 – 2
 - private, 8 – 2
 - public, 8 – 2
 - domain of, 8 – 3
- PL/SQL, 7 – 2
 - anonymous blocks, 7 – 2
 - calling remote stored procedures, 7 – 41
 - cursor variables, 7 – 26
 - datatypes, 7 – 62
 - numeric codes for, 7 – 64
 - dependencies among library units, 7 – 16
 - exception handlers, 7 – 2
 - file I/O, 8 – 28
 - file I/O security, 8 – 30
 - functions
 - arguments, 7 – 45
 - overloading, 7 – 52
 - parameter default values, 7 – 46
 - RESTRICT_REFERENCES pragma, 7 – 48
 - using, 7 – 43
 - hiding source code, 7 – 29
 - packages, 7 – 11
 - program units, 7 – 2
 - dropped tables and, 4 – 9
 - replaced views and, 4 – 13
 - RAISE statement, 7 – 31
 - tables, 7 – 8
 - of records, 7 – 8
 - trigger bodies, 9 – 7 to 9 – 36
 - user-defined errors, 7 – 31
- pragma. *See* RESTRICT_REFERENCES pragma
- precompiler, applications, 7 – 4

precompilers. *See* Oracle Precompilers

PRIMARY KEY constraint

- altering, 6 – 25

- choosing a primary key, 6 – 5

- disabling, 6 – 20

- enabling, 6 – 20

- multiple columns in, 6 – 6

- UNIQUE key constraint vs., 6 – 6

private SQL areas, cursors and, 3 – 9

privileges

- altering sequences, 4 – 26

- altering tables, 4 – 9

- cluster creation, 4 – 41

- creating integrity constraints, 6 – 17

- creating tables, 4 – 7

- disabling triggers, 9 – 19

- dropping a view, 4 – 15

- dropping sequences, 4 – 31

- dropping tables, 4 – 10

- dropping triggers, 9 – 18

- enabling roles and, 13 – 11

- granting, 13 – 14, 13 – 15

- index creation, 4 – 37

- managing, 13 – 8 to 13 – 21

- manually acquiring locks, 3 – 15

- on selected columns, 13 – 17

- recompiling packages or procedures, 11 – 5

- recompiling triggers, 11 – 6

- recompiling views, 11 – 5

- renaming objects, 4 – 50

- replacing views, 4 – 13

- revoking, 13 – 14, 13 – 16

- sequence creation, 4 – 26

- stored procedure execution, 7 – 39

- synonym creation, 4 – 31

- triggers, 9 – 15

- using a view, 4 – 15

- using sequences, 4 – 31

- view creation, 4 – 12

- when revoking object privileges, 13 – 18

procedures

- compilation,

 - avoiding runtime, 11 – 2 to 11 – 4

- listing information about, 7 – 65 to 7 – 67

- local, 7 – 14

- supplied, 7 – 53

- using in applications, 2 – 4

profiles, application design and, 2 – 6

program units, in PL/SQL, 7 – 2

PUBLIC user group

- granting and revoking privileges to, 13 – 19

- procedures and, 13 – 19

purity level, 7 – 47

PUT procedure, 8 – 21, 8 – 23, 8 – 36

- maximum output size for, 8 – 38

PUT_LINE procedure, 8 – 21, 8 – 23, 8 – 38

- maximum output size for, 8 – 38

PUTF procedure, 8 – 38

Q

queries, distributed, 7 – 33 to 7 – 36

R

RAISE statement, 7 – 31

RAISE_APPLICATION_ERROR procedure,
7 – 30

- remote procedures and, 7 – 33 to 7 – 36

raising exceptions, triggers, 9 – 9

RAW datatype, 5 – 9

read-only transactions, 3 – 8

RECEIVE_MESSAGE function, 8 – 8

recompilation, avoiding runtime,
11 – 2 to 11 – 11

REFERENCING option, 9 – 8

referential integrity

- distributed databases and, 6 – 13

- one-to-many relationship, 6 – 9

- one-to-one relationship, 6 – 9

- privileges required to create foreign keys,
6 – 26

- self-referential constraints, 9 – 29

- triggers and, 9 – 26 to 9 – 36

REGISTER procedure, 12 – 4

remote exception handling, 9 – 9

REMOVE procedure, 12 – 5

REMOVE_PIPE procedure, 8 – 10

RENAME command, 4 – 49

renaming objects, 4 – 49

repeatable reads, 3 – 8, 3 – 11

- RESET_PACKAGE procedure, 7 – 53
- RESTRICT_REFERENCES pragma
 - syntax for, 7 – 48
 - using to control side effects, 7 – 47, 7 – 50
 - variant, 7 – 50
- REVOKE command, 13 – 16
 - when in effect, 13 – 20
- revoking privileges and roles
 - on selected columns, 13 – 17
 - REVOKE command, 13 – 16
- RNDS argument, 7 – 48
- RNPS argument, 7 – 48
- ROLE_SYS_PRIVS view, 13 – 3
- ROLE_TAB_PRIVS view, 13 – 3
- roles
 - ADMIN OPTION and, 13 – 14
 - advantages, 13 – 3
 - application, 13 – 2 to 13 – 21
 - creating, 13 – 10
 - default, 13 – 11
 - dropping, 13 – 13
 - enabling, 13 – 3 to 13 – 21
 - GRANT and REVOKE commands, 13 – 13
 - granting, 13 – 14
 - managing, 13 – 8 to 13 – 21
 - operating system granting of, 13 – 13
 - privileges for creating, 13 – 10
 - security policy for application, 13 – 5 to 13 – 21
 - SET ROLE command, 13 – 13
 - user, 13 – 3 to 13 – 21
 - user privileges and enabling, 13 – 11
 - when to enable, 13 – 11
 - WITH GRANT OPTION and, 13 – 15
- ROLLBACK command, 3 – 6
- rolling back transactions, to savepoints, 3 – 6
- row exclusive locks (RX),
 - LOCK TABLE command, 3 – 13
- row labels, modifying, 5 – 16
- row locking, manually locking, 3 – 15
- row share locks (RS),
 - LOCK TABLE command, 3 – 13

- row triggers
 - defining, 9 – 5
 - REFERENCING option, 9 – 8
 - timing, 9 – 4
 - UPDATE statements and, 9 – 5, 9 – 9
- ROW_LOCKING parameter, 3 – 16
 - locking and, 3 – 16
- ROWID datatype, 5 – 9
 - changes in, 5 – 10
- ROWLABEL pseudo-column
 - DBMS MAC mode, definition, 5 – 14
 - OS MAC mode, definition, 5 – 14
- rows
 - chaining across blocks, 4 – 5
 - format, 4 – 2
 - header, 4 – 2
 - size, 4 – 2
 - violating integrity constraints, 6 – 19
- ROWTYPE_MISMATCH exception, 7 – 29
- RS locks, LOCK TABLE command, 3 – 13
- RX locks, LOCK TABLE command, 3 – 13

S

- S locks, LOCK TABLE command, 3 – 13
- sample programs
 - daemon.pc, 8 – 17
 - daemon.sql, 8 – 15
- SAVEPOINT command, 3 – 6
- savepoints
 - maximum number of, 3 – 6
 - rolling back to, 3 – 6
- schemas, 13 – 8
- security
 - enforcing in applications, 2 – 8
 - in PL/SQL file I/O, 8 – 30
 - policy for applications, 13 – 2 to 13 – 8
 - roles, advantages, 13 – 3
 - when using the UTL_FILE package, 8 – 29
- SELECT ... FOR UPDATE command, 3 – 15
- SELECT command
 - OS MAC mode, ROWLABEL pseudo-column and, 5 – 14
 - read consistency, 3 – 11

- SEND_MESSAGE function, 8 – 6
- SEQUENCE_CACHE_ENTRIES parameter, 4 – 30
- sequences
 - accessing, 4 – 26
 - altering, 4 – 26
 - caching numbers, 4 – 25
 - caching sequence numbers, 4 – 29
 - creating, 4 – 25, 4 – 30
 - CURRVAL, 4 – 26, 4 – 28
 - dropping, 4 – 31
 - initialization parameters, 4 – 25
 - NEXTVAL, 4 – 27
 - parallel server, 4 – 26
 - privileges for creating, 4 – 26
 - privileges to alter, 4 – 26
 - privileges to drop, 4 – 31
 - privileges to use, 4 – 31
 - reducing serialization, 4 – 27
 - using in applications, 2 – 6
- SERIALIZABLE option,
 - for ISOLATION LEVEL, 3 – 30
- SERIALIZABLE parameter, 3 – 16
 - locking and, 3 – 16
- serializable transactions, 3 – 27
- Server Manager,
 - SET SERVEROUTPUT ON command, 8 – 21
- sessions
 - communicating between, 8 – 2
 - package state and, 7 – 15
- SET ROLE command, 13 – 4, 13 – 12
 - when using operating system roles, 13 – 13
- SET SERVEROUTPUT ON, 8 – 21
- SET TRANSACTION command, 3 – 8
- SET_DEFAULTS procedure, 12 – 8
- SET_LABEL procedure, 7 – 53
- SET_MLS_LABEL_FORMAT procedure, 7 – 53
- SET-NLS procedure, 7 – 53
- SET_ROLE procedure, 7 – 53
- SET_SQL_TRACE procedure, 7 – 53
- SGA. *See* system global area
- share locks (S), LOCK TABLE command, 3 – 13
- share row exclusive locks (SRX),
 - LOCK TABLE command, 3 – 14
- shared SQL areas, using in applications, 2 – 5
- SHOW command, LABEL option, 5 – 17
- side effects, 7 – 6, 7 – 47
- SIGNAL procedure, 12 – 5
- signatures
 - for dependencies among PL/SQL library units, 7 – 16
 - to manage remote dependencies, 7 – 17
- SORT_AREA_SIZE parameter,
 - index creation and, 4 – 34
- SQL statements
 - access in PL/SQL, 7 – 53
 - application design and, 2 – 7
 - execution, 3 – 2
 - in trigger bodies, 9 – 7 to 9 – 36
 - not allowed in triggers, 9 – 10
 - privileges required for, 13 – 9
 - when constraint checking occurs, 6 – 15
- SQL*Loader, indexes and, 4 – 34
- SQL*Module
 - applications, 7 – 4
 - calling stored procedures from, 7 – 5
- SQL*Plus, 7 – 3, 7 – 4, 7 – 9, 7 – 12, 7 – 34, 7 – 35, 7 – 36, 8 – 22, 8 – 23
 - SET SERVEROUTPUT ON command, 8 – 21
- SRX locks, LOCK TABLE command, 3 – 14
- standards
 - ANSI, 3 – 16
 - compliance, 2 – 7
- state, session, of package objects, 7 – 15
- statement triggers
 - conditional code for statements, 9 – 8 to 9 – 36
 - defining, 9 – 4 to 9 – 36
 - specifying, 9 – 5
 - specifying SQL statement, 9 – 4 to 9 – 36
 - timing, 9 – 4
 - UPDATE statements and, 9 – 5, 9 – 9
- storage parameters
 - PCTFREE, 4 – 8
 - PCTUSED, 4 – 8
- stored functions, 7 – 5
 - creating, 7 – 9

- stored procedures, 7 – 5
 - argument values, 7 – 39 to 7 – 67
 - compilation, avoiding runtime,
 - 11 – 2 to 11 – 4
 - creating, 7 – 9
 - distributed query creation, 7 – 33 to 7 – 36
 - exceptions, 7 – 31
 - exceptions in, 7 – 30
 - invoking, 7 – 36
 - listing information about, 7 – 65 to 7 – 67
 - names of, 7 – 5
 - overloading names of, 7 – 5
 - parameter, default values, 7 – 9
 - privileges, 7 – 39, 11 – 5
 - recompiling, 11 – 4 to 11 – 6
 - remote, 7 – 40
 - remote objects and, 7 – 41 to 7 – 67
 - storing, 7 – 9
 - supplied, 7 – 53
 - synonyms, 7 – 42
 - using in applications, 2 – 4
 - using privileges granted to PUBLIC, 13 – 19
- structs, arrays of in C, 2 – 7
- supplied procedures, 7 – 53
- synonyms
 - creating, 4 – 31
 - dropped tables and, 4 – 9
 - dropping, 4 – 32
 - privileges, 4 – 31, 4 – 32
 - stored procedures and packages, 7 – 42
 - using, 4 – 32
- SYSDATE function, 5 – 6
- system global area
 - buffers DBMS_OUTPUT data, 8 – 23
 - buffers pipes information, 8 – 2
 - holds sequence number cache, 4 – 29
- system-specific Oracle documentation, 3 – 19,
 - 3 – 20, 7 – 57, 8 – 3, 8 – 22, 9 – 3, 10 – 2,
 - 11 – 7, 12 – 3
 - PL/SQL wrapper, 7 – 29
 - UTLDTREE.SQL script, 11 – 3

T

- tables
 - altering, 4 – 8
 - constraining, 9 – 12
 - creating, 4 – 2, 4 – 3
 - designing, 4 – 2
 - dropping, 4 – 9
 - guidelines, 4 – 2, 4 – 3
 - in PL/SQL, 7 – 8
 - increasing column length, 4 – 8
 - key-preserved, 4 – 18
 - location, 4 – 3
 - mutating, 9 – 12
 - privileges for creation, 4 – 7
 - privileges for dropping, 4 – 10
 - privileges to alter, 4 – 9
 - schema of clustered, 4 – 40
 - specifying PCTFREE for, 4 – 5
 - specifying PCTUSED for, 4 – 5
 - specifying tablespace, 4 – 3
 - truncating, 4 – 9
- temporary segments, index creation and, 4 – 34
- text, conventions in this Guide, iv
- thread safety, in OCI applications, 2 – 7
- timestamps,
 - for dependencies among PL/SQL library
 - units, 7 – 16
- TO_CHAR function, 5 – 22
- TO_LABEL function, 5 – 22
- transactions
 - manual locking, 3 – 11
 - read-only, 3 – 8
 - serializable, 3 – 27
 - SET TRANSACTION command, 3 – 8
- triggers
 - about, 7 – 4
 - accessing column values, 9 – 7 to 9 – 36
 - AFTER, 9 – 4
 - auditing with, 9 – 21 to 9 – 36
 - BEFORE, 9 – 4
 - body, 9 – 7 to 9 – 36
 - check constraints, 9 – 32 to 9 – 36
 - column list in UPDATE, 9 – 5, 9 – 9
 - compiled, 9 – 16
 - conditional predicates, 9 – 7, 9 – 8 to 9 – 36

triggers *continued*

- creating, 9 – 3 to 9 – 15
 - data access and, 9 – 33 to 9 – 36
 - debugging, 9 – 17
 - designing, 9 – 2
 - disabling, 9 – 18 to 9 – 36
 - distributed query creation, 7 – 33
 - dropped tables and, 4 – 9
 - enabling, 9 – 19 to 9 – 36
 - error conditions and exceptions, 9 – 9
 - events, 9 – 4 to 9 – 36
 - examples, 9 – 21 to 9 – 36
 - FOR EACH ROW clause, 9 – 5
 - generating derived column values, 9 – 34
 - illegal SQL statements, 9 – 10
 - integrity constraints vs., 9 – 2, 9 – 25
 - listing information about, 9 – 20 to 9 – 21
 - migration issues, 9 – 17
 - modifying, 9 – 18
 - multiple same type, 9 – 12
 - mutating tables and, 9 – 12
 - naming, 9 – 4
 - package variables and, 9 – 11
 - prerequisites before creation, 9 – 3
 - privileges, 9 – 15, 11 – 6
 - privileges to disable, 9 – 19
 - privileges to drop, 9 – 18
 - procedures and, 9 – 10
 - recompiling, 9 – 17, 11 – 6
 - REFERENCING option, 9 – 8
 - referential integrity and, 9 – 26
 - remote dependencies and, 9 – 11
 - remote exceptions, 9 – 9
 - restrictions, 9 – 6 to 9 – 36
 - row, 9 – 5
 - scan order, 9 – 11
 - stored, 9 – 16
 - use of LONG and LONG RAW datatypes, 9 – 11
 - username reported in, 9 – 15
 - using in applications, 2 – 5
 - WHEN clause, 9 – 6 to 9 – 36
- TRUNCATE TABLE command, 4 – 9
- Trusted Oracle7 Server, dynamic SQL, 10 – 7
- tuning
- overview, 2 – 8
 - using LONGs, 5 – 8

U

- unhandled exceptions, 7 – 32
- UNIQUE key constraints
- altering, 6 – 25
 - combining with NOT NULL constraint, 6 – 4
 - composite keys and nulls, 6 – 6
 - data integrity, 6 – 25
 - disabling, 6 – 20
 - enabling, 6 – 20
 - PRIMARY KEY constraint vs., 6 – 6
 - when to use, 6 – 6
- UNIQUE_SESSION_ID procedure, 7 – 53
- UNPACK_MESSAGE procedures, 8 – 10
- UPDATE command
- column values and triggers, 9 – 7
 - data consistency, 3 – 11
 - triggers and, 9 – 5, 9 – 9
 - triggers for referential integrity, 9 – 28, 9 – 29
- updating applications, 2 – 10
- updating tables, with parent keys, 6 – 10, 6 – 11
- USER function, 6 – 4
- user locks, requesting, 3 – 18
- user-defined errors, 7 – 30, 7 – 31
- declaring, 7 – 31
- USER_ERRORS view, 7 – 35
- for debugging stored procedures, 7 – 35
- USERENV function, LABEL option, 5 – 16
- username, as reported in a trigger, 9 – 15
- usernames, database, schemas and, 13 – 8
- users, database, dropped roles and, 13 – 13
- UTL_FILE package, 8 – 28
- security issues, 8 – 29
- UTLDTREE.SQL file, 11 – 3, 11 – 7
- UTLEXCPT.SQL file, 6 – 23
- UTLLOCKT.SQL script, 3 – 27

V

- VARCHAR datatype, 5 – 3
- VARCHAR2 datatype, 5 – 3
- when to use, 5 – 2
- VARIABLE_VALUE procedure, 10 – 6, 10 – 18

views

- creating, 4 – 10
 - creating with errors, 4 – 12
 - dropped tables and, 4 – 9
 - dropping, 4 – 15
 - FOR UPDATE clause and, 4 – 11
 - invalid, 4 – 15
 - join views, 4 – 16
 - ORDER BY clause and, 4 – 11
 - privileges, 4 – 12, 11 – 5
 - recompiling, 11 – 4 to 11 – 6
 - replacing, 4 – 13
 - restrictions, 4 – 14
 - using, 4 – 14
 - when to use, 4 – 10
 - WITH CHECK OPTION, 4 – 11
- violating integrity constraints, 6 – 19

W

- WAITANY procedure, 12 – 6
- WAITONE procedure, 12 – 6
- WHEN clause, 9 – 6 to 9 – 36
 - cannot contain PL/SQL expressions, 9 – 6
- WITH GRANT OPTION, 13 – 15
- WNDS argument, 7 – 48
- WNPS argument, 7 – 48
- wrapper, 7 – 29
 - to hide PL/SQL code, 7 – 29

X

- X locks, LOCK TABLE command, 3 – 15

Reader’s Comment Form

Name of Document: Oracle7™ Server Application Developer’s Guide
Part No. A32536–1

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

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

If you find any errors or have any other suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065 U.S.A.
Fax: (415) 506–7200

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

Thank you for helping us improve our documentation.