Oracle8™

Application Developer's Guide

Release 8.0

December, 1997

Part No. A58241-01

ORACLE®

Enabling the Information Age™

Oracle8 Application Developer's Guide

Part No. A58241-01

Release 8.0

Primary Author: Denis Raphaely

Contributing Authors: Paul Lane, Lefty Leverenz, Richard Mateosian

Contributors: Richard Allen, Neerja Bhatt, Steven Bobrowski, George Buzsaki, Sashi Chandrasekaran, Atif Chaudry, Greg Doherty, Dieter Gawlick, Gary Hallmark, Michael Hartstein, Chin Hong, Kenneth Jacobs, Hakan Jakobsson, Amit Jasuja, Robert Jenkins, Jr., Jonathan Klein, Robert Kooi, Susan Kotsovolos, Vishu Krishnamurthy, Ramkumar Krishnan, Juan Loaiza, William Maimone, Andrew Mendelsohn, Mark Moore, Ravi Narayaran, Goran Olsson, Edward Peeler, Thomas Portfolio, Maria Pratt, Tuomas Pystynen, Mark Ramacher, Madhu Reddy, Hasan Rizvi, Premal Shah, Timothy Smith, Alvin To, Usha Sundaram, Scott Urman, Peter Vasterd, Joyo Wijaya

Graphic Designer: Valarie Moore

Contents

2 The Application Developer

3 Processing SQL Statements

4 Managing Schema Objects

6 Large Objects (LOBs)

7 User-Defined Datatypes — An Extended Example

8 Object Views—An Extended Example

9 Maintaining Data Integrity

10 Using Procedures and Packages

11 Advanced Queuing

12 PL/SQL Input/Output

13 Using Database Triggers

14 Using Dynamic SQL

15 Dependencies Among Schema Objects

16 Signalling Database Events with Alerters

17 Establishing a Security Policy

18 Oracle XA

Index

Send Us Your Comments

Oracle8 Application Developer's Guide, Release 8.0

Part No. A58241-01

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

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

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

- electronic mail - infodev@us.oracle.com
- FAX - (650)506-7228
- postal service:
  Oracle Corporation
  Oracle Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

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

Preface

This Guide describes features of application development for the Oracle Server, Release 8.0. Information in this Guide applies to versions of the Oracle Server that run on all platforms, and does not include system-specific information.

The Preface includes the following sections:

- Information in This Guide
- How This Book Is Organized
- Conventions Used in this Guide
- Your Comments Are Welcome

Information in This Guide

As an application developer, you should learn about the many Oracle Server features that can ease application development and improve performance. This Guide describes Oracle Server features that relate to application development. It 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 more information about the material covered. Chapter 1 also points you to other Oracle documentation that contains related information.

Audience

The *Oracle8 Application Developer's Guide* is intended for programmers developing new applications or converting existing applications to run in the Oracle environment. This Guide will also be valuable to systems analysts, project managers, and others interested in the development of database applications.

This guide assumes that you have a working knowledge of application programming, and that you are familiar with the use of Structured Query Language (SQL) to access information in relational database systems.

Certain sections of this Guide also assume a knowledge of the basic concepts of object oriented programming.

Feature Coverage and Availability

The *Oracle8 Application Developer's Guide* contains information that describes the features and functionality of the Oracle8 and the Oracle8 Enterprise Edition products. Oracle8 and Oracle8 Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use object functionality, you must have the Enterprise Edition and the Objects Option.

For information about the differences between Oracle8 and the Oracle8 Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8 and the Oracle8 Enterprise Edition*.

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 Oracle Call Interface (OCI) is described ins:

- *Oracle Call Interface Programmer's Guide*

You can use the OCI to build third-generation language (3GL) applications that access the Oracle Server.

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 *Pro*C/C++ Precompiler Programmer's Guide*.

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 Oracle Tools documentation.

For SQL information, see the *Oracle8 SQL Reference* and *Oracle8 Administrator's Guide*. For basic Oracle concepts, see *Oracle8 Concepts*.

How This Book Is Organized

The *Oracle8 Application Developer's Guide* contains eighteen chapters. A brief summary of what you will find in each chapter follows:

Chapter 1: Information Sources for Application Developers

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 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. It also discusses performance enhancements to data retrieval through the use of indexes and clusters.

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, raw binary data, and row identifiers (ROWIDs).

Chapter 6: Large Objects (LOBs)

This chapter describes the extended SQL commands and PL/SQL interface for the LOB datatypes, which include BLOBs for unstructured binary data, CLOBs and NCLOBs for character data, and BFILEs for data stored in an external file.

Chapter 7: User-Defined Datatypes — An Extended Example

This chapter explains how to define and use the composite datatypes and collection datatypes (varying-length arrays and nested tables) that can be created for particular application requirements.

Chapter 8: Object Views—An Extended Example

This chapter explains how to define and use object views.

Chapter 9: Maintaining Data Integrity

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

Chapter 10: 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 in this chapter.

Chapter 11: Advanced Queuing

This chapter describes how to use advanced queuing to defer or regulate the execution of work in a client/server environment.

Chapter 12: PL/SQL Input/Output

This chapter describes how to use public and private pipes to allow sessions in the same Oracle Server instance to communicate with one another or with a disk file.

Chapter 13: Using Database Triggers

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

Chapter 14: Using Dynamic SQL

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

Chapter 15: Dependencies Among Schema Objects

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

Chapter 16: Signalling Database Events with Alerters

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

Chapter 17: Establishing a Security Policy

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

Chapter 18: Oracle XA

This chapter describes how to use the Oracle XA library.

Conventions Used in this Guide

The following notational and text formatting conventions are used in this guide:

[ ]
Square brackets indicate that the enclosed item is optional. Do not type the brackets.

{ }
Braces enclose items of which only one is required.

|
A vertical bar separates items within braces, and may also be used to indicate that multiple values are passed to a function parameter.

...
In code fragments, an ellipsis means that code not relevant to the discussion has been omitted.

`font change`
SQL or C code examples are shown in monospaced font.

italics
Italics are used for OCI parameters, OCI routines names, file names, and data fields.

UPPERCASE
Uppercase is used for SQL keywords, like SELECT or UPDATE.

This guide uses special text formatting to draw the reader's attention to some information. A paragraph that is indented and begins with a bold text label may have special meaning. The following paragraphs describe the different types of information that are flagged this way.

Note: The "Note" flag indicates that the reader should pay particular attention to the information to avoid a common problem or increase understanding of a concept.

Warning: An item marked as "Warning" indicates something that an OCI programmer must be careful to do or not do in order for an application to work correctly.

See Also: Text marked "See Also" points you to another section of this guide, or to other documentation, for additional information about the topic being discussed.

Your Comments Are Welcome

We value and appreciate your comment as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the following e-mail address:

infodev@us.oracle.com

If you prefer, you can send letters or faxes containing your comments to the following address:

Server Technologies Documentation Manager

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065

Fax: (650) 506-7200

1

Information Sources for Application Developers

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 Oracle 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 Net8 to connect client applications to Oracle 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 Oracle. Visit your local technical or university bookstore to discover the titles available.

Specific Topics

This section tells you where to get information about the following topics:

- Business Rules
- Client-Side Tools
- Communicating with 3GL Programs
- Database Constraints
- Database Design
- Datatypes
- Debugging
- Error Handling
- Gateways
- Oracle-Supplied Packages
- PL/SQL
- Schema Objects
- Security
- SQL Statements

Business Rules

You can enforce business rules in your Oracle application using integrity constraints on columns of a table, or by using triggers. See

See Also:

- Chapter 9, "Maintaining Data Integrity" in this Guide for a description of integrity constraints
- Chapter 13, "Using Database Triggers" for a discussion of database triggers
- *Oracle8 Concepts* 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_PIPES package.

See Also: Chapter 12, "PL/SQL Input/Output" for a detailed discussion of this package.

WARNING: The `DBMS_PIPES` package is not transaction safe, so it must be used with care.

It was not possible to call a 3GL routine directly from PL/SQL in release 7.3 of the Oracle Server. For information about how to do this in Oracle8, refer to the *PL/SQL User's Guide and Reference*.

Database Constraints

How to use database constraints such as NOT NULL, PRIMARY KEY, and FOREIGN KEY is described in Chapter 9, "Maintaining Data Integrity" of this Guide. See *Oracle8 Concepts* for a basic introduction to constraints.

Database Design

Database design is not discussed exhaustively in this Guide. See the *Oracle8 Concepts* manual for a basic discussion, and refer to the *Oracle8 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, "Selecting a Datatype" of this Guide. For a more comprehensive treatment of datatypes, see the *Oracle Call Interface Programmer's Guide.*

Debugging

You can use the DBMS_OUTPUT and the DBMS_PIPES packages for first-level debugging of your PL/SQL code. See Chapter 12, "PL/SQL Input/Output" 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 *Oracle8 Error Messages* manual for a complete listing or 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 Chapter 10, "Using Procedures and Packages", Chapter 12, "PL/SQL Input/Output", Chapter 13, "Using Database Triggers", Chapter 14, "Using Dynamic SQL", and Chapter 16, "Signalling Database Events with Alerters".

> **See Also:** "Supplied Packages" on page 10-59 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 *Oracle8 Concepts* manual for introductory material. For example, you might want to read the chapter in the *Oracle8 Concepts* manual called "Procedures and Packages" before reading Chapter 10, "Using Procedures and Packages" in this Guide.

Security

Your primary reference for security in this Guide is Chapter 17, "Establishing a Security Policy". Security issues are also discussed in the *Oracle8 Concepts* manual.

SQL Statements

Your primary reference for the SQL language is the *Oracle8 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 Also: “Client-Side Tools” on page 1-3.

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:

1. Identify the entities of your system and construct a table to represent each entity.
2. Identify the relationships between the entities and either extend the current tables or create new tables to represent these relationships.
3. 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 can help improve, automate, and document designs. The Oracle database design products are *Designer/2000* and *Object Database Designer.*

Designer/2000 is a business and application modeling toolset which generates both servers and applications from graphical models. *Designer/2000* release 2.0 supports all the scalability features of Oracle8 such as partitioned tables, LOB datatypes, index-organized tables, and deferred constraint checking as well as the object features such as user-defined views, object tables, and referenced and embedded object types.

Object Database Designer supports all aspects of the design and creation of an Object-Relational Database Management System (ORDBMS). Type modeling forms the core of an object-oriented development. Object Database Designer implements type modeling using UML and then uses the type model to drive generation of Oracle8 database designs and C++ classes with transparent persistence—thus supporting both the database designer and the application developer.

See the *Designer/2000* and *Object Database Designer* manuals for additional information about these products.

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, "Selecting a Datatype" 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 Also: *Oracle8 Distributed Database Systems.*

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 Also:** Chapter 9, "Maintaining Data Integrity", 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 Also:** Chapter 10, "Using Procedures and Packages".

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 Also:** Chapter 13, "Using Database Triggers".

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 *Oracle8 Tuning* 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 *Oracle8 Tuning* 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 *Oracle8 Reference* 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, “Processing SQL Statements”.

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 *Oracle8 SQL Reference* 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 offers calls that provide:

- 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

See *Oracle Call Interface Programmer's Guide* for more information.

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 manual.

> **See Also:** *Oracle8 SQL Reference* manual for more detailed information.

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 *Oracle8 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 Also:** Chapter 17, "Establishing 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 *Oracle8 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 Also:** "Using Integrity Constraints" on page 9-2 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 *Oracle8 Tuning* manual.
- Use packages to further improve performance and reduce runtime recompilations. Packages are described in Chapter 10, "Using Procedures and Packages".
- Use cached sequence numbers to generate primary key values; see "Creating Sequences" on page 4-23.
- Use array processing to reduce the number of calls to Oracle; see *Oracle8 Tuning*.
- Use VARCHAR2 to store character data instead of CHAR, which blank-pads data; see "Using Character Datatypes" on page 5-5.
- Use LOB datatypes instead of LONG or LONG RAW datatypes; see Chapter 6, "Large Objects (LOBs)" for more information.
- If you use LONG or LONG RAW datatypes, store the 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 "Using the LONG Datatype" on page 5-10 and "Using RAW and LONG RAW Datatypes" on page 5-12.
- 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 *Oracle8 Tuning*.

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 *Oracle8 Tuning*.

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.

3

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
- Explicit Data Locking
- Explicitly Acquiring Row Locks
- SERIALIZABLE and ROW_LOCKING Parameters
- Creating User Locks
- Sample User Locks
- Viewing and Monitoring Locks
- Concurrency Control Using Serializable Transactions

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 stages 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 *Oracle8 Concepts* for a description of each stage 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.

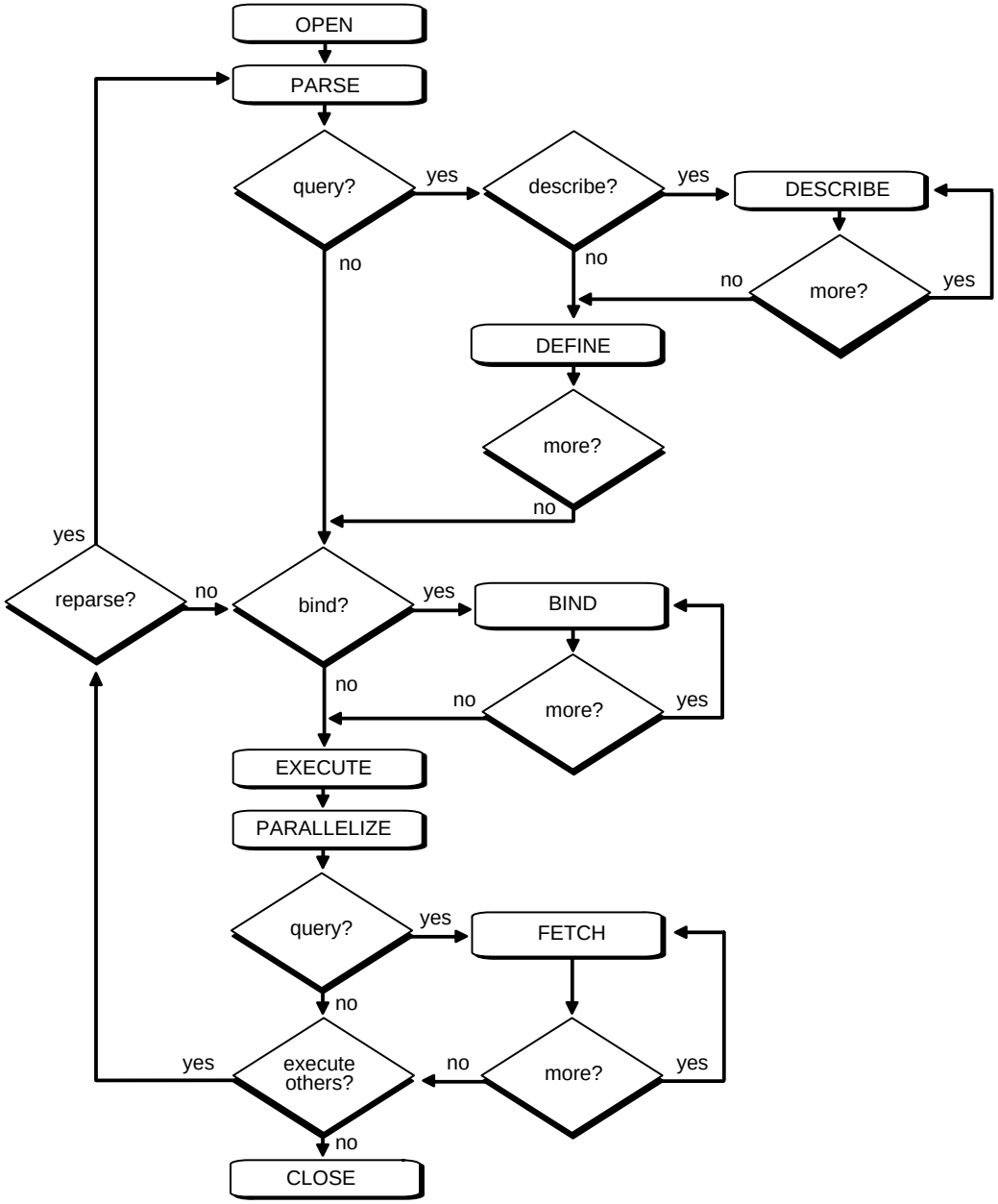
The FIPS flagging feature supports flagging through interactive SQL statements submitted using Enterprise Manager or SQL*Plus. The Oracle Precompilers and SQL*Module also support FIPS flagging of embedded and module language SQL.

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 *Oracle8 Tuning*.

- 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 Also:

  - "Serializable Transaction Interaction" on page 3-31.
  - *Oracle8 Concepts*.

- 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 10-42.
- 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 *Oracle8 Distributed Database Systems*.

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 Also: *Oracle8 Distributed Database Systems.*

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:

SQL Statement	Results
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 *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 *Oracle8 Reference*.

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.)

Cancelling Cursors

Cancelling a cursor frees resources from the current fetch.The information currently in the associated private area is lost but the cursor remains open, parsed, and associated with its bind variables.

Note: You cannot cancel cursors using Pro*C or PL/SQL.

See Also: For more information about cancelling cursors, see *Oracle Call Interface Programmer's Guide*.

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 *Oracle8 Reference* 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 *Oracle8 Distributed Database Systems*.

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 totsal = totsal * 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 13, "Using Database Triggers", show how to implement referential integrity. In the EMP_DEPT_CHECK trigger (see page 13-28), 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 3-11, a distributed transaction waiting for a row lock can time-out 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

Two factors determine how an instance handles locking: the SERIALIZABLE option of the SET TRANSACTION or ALTER SESSION command and the ROW_LOCKING initialization parameter. 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.

> **See Also:** For detailed explanations of these parameters, see *Oracle8 Reference.*

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

Three combinations of settings for SERIALIZABLE and ROW_LOCKING, other than the default settings, are available to change the way locking occurs for transactions. Table 3–1 summarizes the non-default settings and why you might choose to execute your transactions in a non-default way.

Table 3–1 Summary of Non-Default Locking Options

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

Table 3–2 illustrates the difference in locking behavior resulting from the three possible settings of the SERIALIZABLE option and ROW_LOCKING initialization parameter, as shown in Table 3–1.

Table 3–2 Non-default Locking Behavior

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 . .						
ROW SHARE MODE	RS	RS	RS	RS	RS	RS
ROW EXCLUSIVE MODE	RX	RX	RX	RX	RX	RX
SHARE MODE	S	S	S	S	S	S
SHARE ROW EXCLUSIVE MODE	SRX	SRX	SRX	SRX	SRX	SRX
EXCLUSIVE MODE	X	X	X	X	X	X
DDL statements	-	X	-	X	-	X

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 3–3Summarizes the procedures available in the DBMS_LOCK package.

Table 3–3 DBMS_LOCK Package Functions and Procedures

Function/Procedure	Description	Refer to
ALLOCATE_UNIQUE	Allocate a unique lock ID to a named lock.	page 3-19
REQUEST	Request a lock of a specific mode.	page 3-20
CONVERT	Convert a lock from one mode to another.	page 3-23
RELEASE	Release a lock.	page 3-25
SLEEP	Put a procedure to sleep for a specified time.	page 3-25

User locks never conflict with Oracle locks because they are identified with the prefix “UL”. You can view these locks using the Enterprise 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 Also: 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 Also: See page 10-59 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);
```

Table 3–4 DBMS_LOCK.ALLOCATE_UNIQUE Procedure Parameters

Parameter	Description
LOCKNAME	Specify the name of the lock for which you want to generate a unique ID. The first session to call ALLOCATE_UNIQUE with a new lock name causes a unique lock ID to be generated and stored in the DBMS_LOCK_ALLOCATED table. The handle to this ID is then returned for this call, and all subsequent calls (usually by other sessions). Lock IDs assigned by ALLOCATE_UNIQUE are in the range of 1073741824 to 1999999999. Do not use lock names beginning with ORA$; these names are reserved for products supplied by Oracle Corporation.
LOCKHANDLE	Returns to the caller the handle to the lock ID generated by ALLOCATE_UNIQUE. You can use this handle in subsequent calls to REQUEST, CONVERT, and RELEASE. LOCKHANDLE can be up to VARCHAR2(128). 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. All sessions using a lock handle returned by ALLOCATE_UNIQUE 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.
EXPIRATION_SECS	Specify the number of seconds to wait after the last ALLOCATE_UNIQUE has been performed on a given lock, before allowing that lock to be deleted from the DBMS_LOCK_ALLOCATED table. The default waiting period is 10 days. You should not delete locks from this table. Subsequent calls to ALLOCATE_UNIQUE may delete expired locks to recover space.

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 online, for the current default values.

Table 3–5 DBMS_LOCK.REQUEST Function Parameters

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	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 Enterprise Manager monitors are shown in parentheses. 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) Each of these lock modes is explained in *Oracle8 Concepts*.
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).
RELEASE_ON_COMMIT	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 3–6 DBMS_LOCK.REQUEST Function Return Values

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

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;
```

Table 3–7 DBMS_LOCK.CONVERT Function Parameters

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	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 Enterprise Manager monitors are shown in parentheses. 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) Each of these lock modes is explained in *Oracle8 Concepts*.
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 3–8 DBMS_LOCK.CONVERT Function Return Values

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

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;
```

Table 3–9 DBMS_LOCK.RELEASE Function Parameter

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 3–10 DBMS_LOCK.RELEASE Function Return Values

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

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);
```

Table 3–11 DBMS_LOCK.SLEEP Procedure Parameters

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.

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

Enterprise Manager Monitors (Lock and Latch Monitors)	The Monitor feature of Enterprise Manager provides two monitors for displaying lock information of an instance. Refer to *Oracle Server Manager User's Guide* for complete information about the Enterprise Manager monitors.
`UTLLOCKT.SQL`	The `UTLLOCKT.SQL` script displays a simple character lock wait-for graph in tree structured fashion. Using any *ad hoc* 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.

Table 3–12 ANSI Isolation Levels

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.

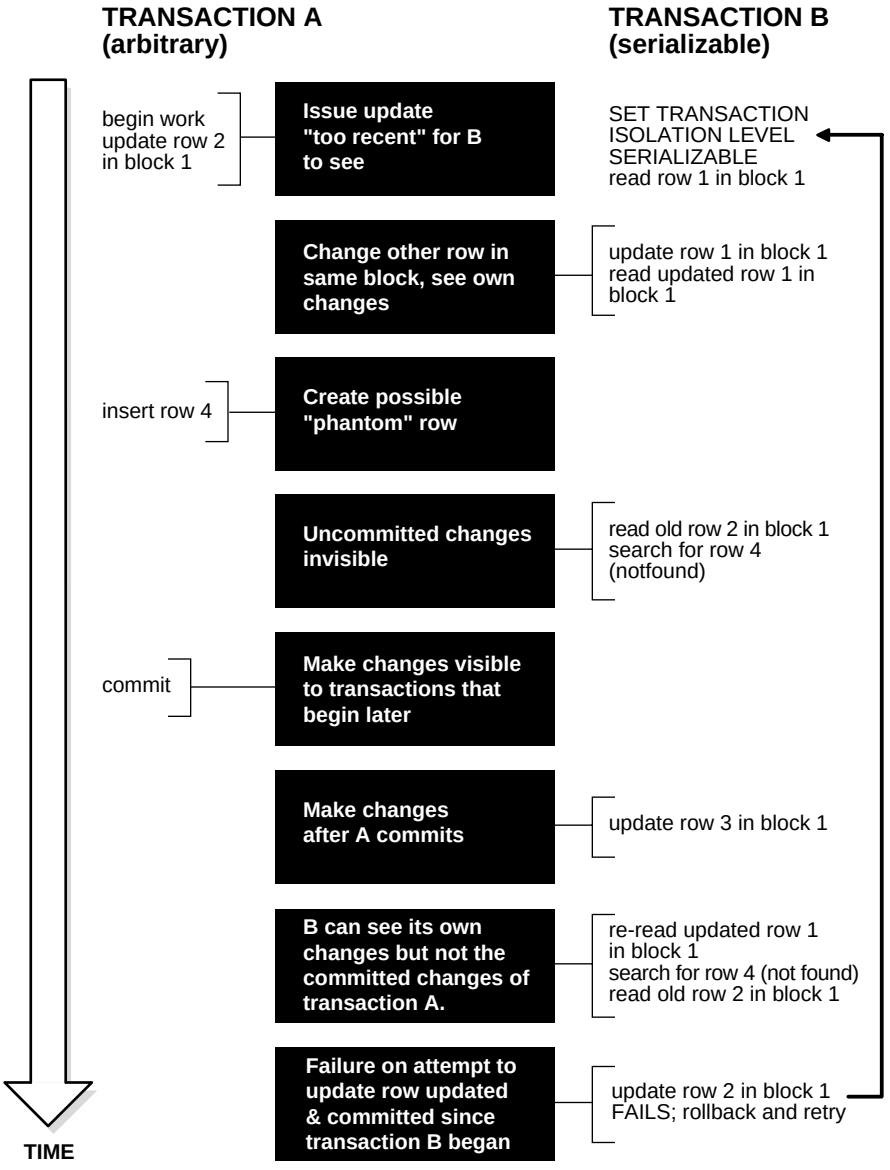
(2) A transaction re-read data committed by another transaction and sees the new data

(3) A transaction can re-execute a query, and discover new rows inserted by another committed transaction

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 3-31.

Figure 3–2 Time Line for Two Transactions



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).

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 Also:** *Oracle8 SQL Reference* for the complete syntax of the SET TRANSACTION and ALTER SESSION commands.

The INITRANS Parameter

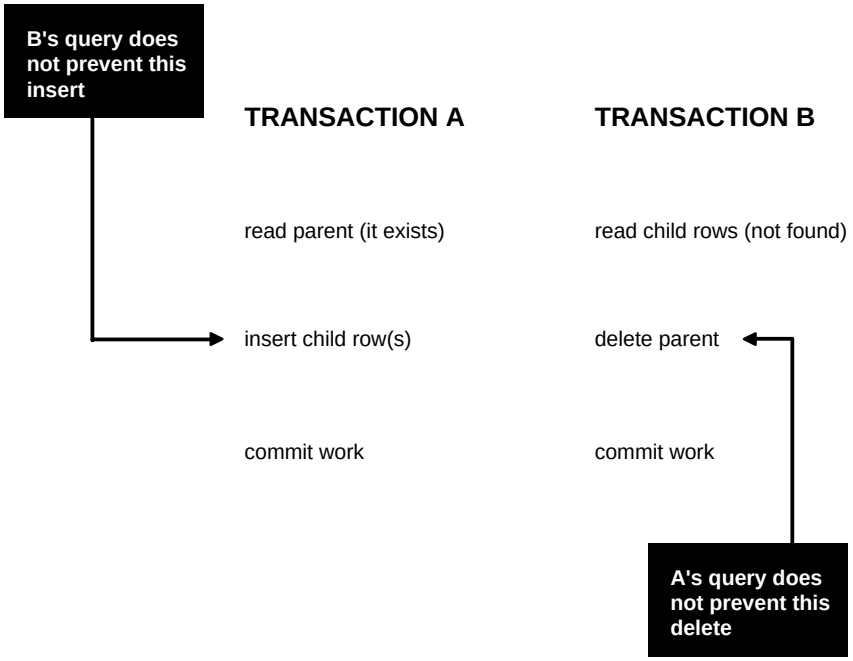
Oracle stores control information in each data block to manage access by concurrent transactions. Therefore, if you set the transaction isolation level to serializable, you must use the ALTER TABLE command to set INITRANS to at least 3. This parameter will cause Oracle to allocate sufficient storage 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 Oracle 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 Oracle 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 Oracle 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

Oracle 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 Oracle 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.

Oracle 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, Oracle 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 Oracle), 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. Oracle'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.

Table 3–13 Read Committed vs. Serializable Transaction

	Read Committed	Serializable
Dirty write	Not Possible	Not Possible
Dirty read	Not Possible	Not Possible

Table 3–13 (Cont.) Read Committed vs. Serializable Transaction

	Read Committed	Serializable
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

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 trade-off between consistency and concurrency (transaction throughput).

Both Oracle 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 Oracle, while queries still see consistent data, both `READ COMMIT-`

TED 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.

4

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:

- Managing Tables
- Managing Views
- Modifying a Join View
- Managing Sequences
- Managing Synonyms
- Managing Indexes
- Managing Clusters, Clustered Tables, and Cluster Indexes
- Managing Hash Clusters and Clustered Tables
- Miscellaneous Management Topics for Schema Objects

> **See Also:** Specific information is described in the following locations:
>
> - Procedures, functions, and packages — Chapter 10
> - Object types — Chapter 7
> - Dependency information — Chapter 15.
> - If you use symmetric replication, see *Oracle8 Replication* for information on managing schema objects, such as snapshots.
> - If you use Trusted Oracle, there are additional privileges required and issues to consider when managing schema objects; see the *Trusted Oracle* documentation.

Managing Tables

A table is the data structure that holds data in a relational database. A table is composed 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 *Oracle8 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 9, “Maintaining Data Integrity” 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

Specifying PCTFREE

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 the 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 trade-off 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 over-

head, 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.

Example1

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.

Example2

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.

Example3

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 *Oracle8 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 4-13 and page 4-15 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 allows you, with some restrictions, to modify views that involve joins. 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 that involves a join operation, 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 there are restrictions on modifying either the EMP or the DEPT base table through this view, for example, using a statement such as:

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

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 *Oracle8 Concepts* 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.
- 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 in "Modifying a Join View" on page 4-15, 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 4-15):

```
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

Three views you can use for modifying join views are shown in Table 4–1.

Table 4–1 UPDATABLE_COLUMNS Views

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.

Outer Joins

Views that involve outer joins are modifiable in some cases. For example:

```
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.

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 *Oracle8 Parallel Server Concepts and Administration*. *G*eneral infor-

mation about caching sequence numbers is included in “Caching Sequence Numbers” on page 4-27.

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 stand-alone procedure or package procedure, you could execute it in SQL*Plus or Enterprise 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 syn-

onym, 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 *Oracle8 Tuning* 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 *Oracle8 Utilities* 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 9-10 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 trade-off 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 sig-

nificant 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.

Figure 4–1 The VENDOR_PARTS Table

Table VENDOR_PARTS

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

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 *Oracle8 Tuning*.

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 "Index the Correct Tables and Columns" on page 4-31 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 trade-offs 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
    INITRANS 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 *Oracle8 Parallel Server Concepts and 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 "Dropping Tables" on page 4-8 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 "Dropping Indexes" on page 4-35 for more information.

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 condi-

tions 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 4-39 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 4-36 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 *Oracle8 Tuning*.

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
- Naming Schema Objects
- Name Resolution in SQL Statements
- Renaming Schema Objects
- Listing Information about Schema Objects

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 Schema 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 schema object in the database then has a global object name consisting of the schema object name and the global database name.

Because Oracle ensures that the schema 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 *Oracle8 Concepts* for more information about 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

Note: 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.

If you use the RENAME command to rename a table, view, sequence, or a private synonym of a table, view, or sequence, grants made for the object are carried forward for the new name, ad the next 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 schema objects . 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

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
```

Note: 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
```

5

Selecting a Datatype

This chapter discusses how to use Oracle *built-in* datatypes in applications. Topics include:

- Oracle Built-In Datatypes
- ROWIDs and the ROWID Datatype
- Trusted Oracle MLSLABEL Datatype
- ANSI/ISO, DB2, and SQL/DS Datatypes
- Data Conversion

> **See Also:** For information about *user-defined* datatypes, refer to *Oracle8 Concepts* and to Chapter 7, "User-Defined Datatypes — An Extended Example" in this manual.

Oracle Built-In Datatypes

A datatype associates a fixed set of properties with the values that can be used in a column of a table or in an argument of a procedure or function. These properties cause Oracle to treat values of one datatype differently from values of another datatype; for example, Oracle can add values of NUMBER datatype but not values of RAW datatype.

Oracle supplies the following built-in datatypes:

- character datatypes
  - CHAR
  - NCHAR
  - VARCHAR2 and VARCHAR
  - NVARCHAR2
  - CLOB
  - NCLOB
  - LONG
- NUMBER datatype
- DATE datatype
- binary datatypes
  - BLOB
  - BFILE
  - RAW
  - LONG RAW

Another datatype, ROWID, is used for values in the ROWID pseudocolumn, which represents the unique address of each row in a table.

> **See Also:** Figure 5–2 summarizes the information about each Oracle built-in datatype. See *Oracle8 Concepts* for general descriptions of these datatypes, and see Chapter 6, "Large Objects (LOBs)" in this Guide for information about the LOB datatypes.

Table 5–1 Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length and Default
CHAR (*size*)	Fixed-length character data of length *size* bytes.	Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (one-byte or multibyte) before setting *size.*
VARCHAR2 (*size*)	Variable-length character data.	Variable for each row, up to 4000 bytes per row. Consider the character set (one-byte or multibyte) before setting *size.* A maximum *size* must be specified.
NCHAR(*size*)	Fixed-length character data of length *size* characters or bytes, depending on the national character set.	Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 2000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 (*size*)	Variable-length character data of length *size* characters or bytes, depending on national character set. A maximum *size* must be specified.	Variable for each row. Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
CLOB	Single-byte character data.	Up to 2^32 - 1 bytes, or 4 gigabytes.

Table 5–1 (Cont.) Summary of Oracle Built-In Datatypes

NCLOB	Single-byte or fixed-length multi-byte national character set (NCHAR) data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
LONG	Variable-length character data.	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility.
NUMBER (p, s)	Variable-length numeric data. Maximum precision *p* and / or scale *s* is 38.	Variable for each row. The maximum space required for a given column is 21 bytes per row.
DATE	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E.	Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-YY) specified by NLS_DATE_FORMAT parameter.
BLOB	Unstructured binary data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
BFILE	Binary data stored in an external file.	Up to 2^32 - 1 bytes, or 4 gigabytes.
RAW (size)	Variable-length raw binary data.	Variable for each row in the table, up to 2000 bytes per row. A maximum *size* must be specified. Provided for backward compatibility.
LONG RAW	Variable-length raw binary data.	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility.
ROWID	Binary data representing row addresses.	Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.
MLSLABEL	Trusted Oracle datatype.	See the *Trusted Oracle* documentation.

Using Character Datatypes

Use the character datatypes to store alphanumeric data.

- CHAR and NCHAR datatypes store fixed-length character strings.
- VARCHAR2 and NVARCHAR2 datatypes store variable-length character strings. (The VARCHAR datatype is synonymous with the VARCHAR2 datatype.)
- CLOB and NCLOB datatypes store single-byte and multibyte character strings of up to four gigabytes.

  See Also: Chapter 6, "Large Objects (LOBs)")

- The LONG datatype stores variable-length character strings containing up to two gigabytes, but with many restrictions..

  See Also: "Restrictions on LONG and LONG RAW Data" on page 5-10

  This datatype is provided for backward compatibility with existing applications; in general, new applications should use CLOB and NCLOB datatypes to store large amounts of character data.

When deciding which datatype to use for a column that will store alphanumeric data in a table, consider the following points of distinction:

Space Usage

- To store data more efficiently, use the VARCHAR2 datatype. The CHAR datatype blank-pads and stores trailing blanks up to a fixed column length for all column values, while the VARCHAR2 datatype does not blank-pad or store trailing blanks for column values.

Comparison Semantics

- Use the CHAR datatype when you require ANSI compatibility in comparison semantics, that is, when trailing blanks are not important in string comparisons. Use the VARCHAR2 when trailing blanks are important in string comparisons.

Future Compatibility

- The CHAR and VARCHAR2 datatypes are and will always be fully supported. At this time, the VARCHAR datatype automatically corresponds to the VARCHAR2 datatype and is reserved for future use.

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_LANGUAGE parameter, where these are different.

Column Lengths for Single-Byte and Multibyte Character Sets

The lengths of CHAR and VARCHAR2 columns are specified in bytes rather than characters, and are constrained as such. The lengths of NCHAR and NVARCHAR2 columns are specified either in bytes or in characters, depending on the national character set being used.

When using a multibyte database character encoding scheme, consider carefully the space required for tables with character columns. 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 multibyte, there generally is no such correspondence. A character might consist of one or more bytes depending upon the specific multibyte encoding scheme, and whether shift-in/shift-out control codes are present.

> **See Also:** *Oracle8 Reference* for information about National Language Support features of Oracle and support for different character encoding schemes.

Comparison Semantics

Oracle compares CHAR and NCHAR values using *blank-padded comparison semantics*. If two values have different lengths, Oracle adds blanks at the end of the shorter value, until the two values are the same length. Oracle then compares the values character-by-character up to the first character that differs. The value with the greater character in the first differing position is considered greater. Two values that differ only in the number of trailing blanks are considered equal.

Oracle compares VARCHAR2 and NVARCHAR2 values using *non-padded comparison semantics*. Two values are considered equal only if they have the same characters and are of equal length. Oracle compares the values character-by-character up to the first character that differs. The value with the greater character in that position is considered greater.

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.

> **See Also:** For more information on comparison semantics for these datatypes, see the *Oracle8 SQL Reference*.

Using the NUMBER Datatype

Use the NUMBER datatype to store real numbers in a fixed-point or floating-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 x 10^-130 to 9.99...x10^125, as well as zero, in a NUMBER column.

For numeric columns you can specify the column as a floating-point number:

```
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)
```

Although not required, specifying the precision and scale for numeric fields provides extra integrity checking on input. If a precision is not specified, the column stores values as given. Table 5–2 shows examples of how data would be stored using different scale factors.

Table 5–2 How Scale Factors Affect Numeric Data Storage

Input Data	Stored As	Specified 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

Table 5–2 How Scale Factors Affect Numeric Data Storage

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

See Also: For information about the internal format for the NUMBER datatype, see *Oracle8 Concepts*.

Using the 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 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 the *Oracle Call Interface Programmer's Guide* for a complete description of the Oracle internal date format.

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')
```

Note: Oracle Julian dates might not be compatible with Julian dates generated by other date algorithms. For information about Julian dates, see *Oracle8 Concepts*.

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 Format

Time is stored in 24-hour format#HH:MM:SS. By default, the time in a date field is 12:00:00 A.M. (midnight) 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.

Centuries and the Year 2000

Oracle stores year data with the century information. For example, the Oracle database stores 1996 or 2001, and not just 96 or 01. The DATE datatype always stores a four-digit year internally, and all other dates stored internally in the database have four digit years. Oracle utilities such as import, export, and recovery also deal properly with four-digit years.

However, some applications might be written with an assumption about the year (such as assuming that everything is 19xx). The application might hand over a two-digit year to the database, and the procedures that Oracle uses for determining the century could be different from what the programmer expects. Application programmers should therefore review and test their code with regard to the year 2000.

The RR date format element of the TO_DATE and TO_CHAR functions allows a database site to default the century to different values depending on the two-digit year, so that years 50 to 99 default to 19*xx* and years 00 to 49 default to 20*xx*. This can help applications make the conversion to the new century easily.

The CC date format element of the TO_CHAR function sets the century value to one greater than the first two digits of a four-digit year (for example, '20' from '1900'). For years that are a multiple of 100, this is not the true century. Strictly speaking, the century of '1900' is not the twentieth century (which began in 1901) but rather the nineteenth century.

The following workaround computes the correct century for any Common Era (CE, formerly known as AD) date. If *userdate* is a CE date for which you want the true century, use the expression:

```
DECODE (TO_CHAR (userdate, 'YY'),
```

```
'00', TO_CHAR (userdate - 366, 'CC'),
TO_CHAR (userdate, 'CC'))
```

This expression works as follows: Get the last two digits of the year. If it is '00', then it is a year in which the Oracle century is one year too large so compute a date in the preceding year (whose Oracle century is the desired true century). Otherwise, use the Oracle century.

> **See Also:** For more information about date format codes, see *Oracle8 SQL Reference*.

Using the LONG Datatype

> **Note:** The LONG datatype is provided for backward compatibility with existing applications. For new applications, you should use the CLOB and NCLOB datatypes for large amounts of character data. See Chapter 6, "Large Objects (LOBs)" for information about the CLOB and NCLOB datatypes.

The LONG datatype can store variable-length character data containing up to two gigabytes of information. The length of LONG values might be limited by the memory available on your computer.

You can use columns defined as LONG in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements. LONG columns have many of the characteristics of VARCHAR2 columns.

Restrictions on LONG and LONG RAW Data

Although LONG (and LONG RAW; see below) columns have many uses, their use has some restrictions:

- 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.
- Variables in database triggers cannot be declared using the LONG or LONG RAW datatypes.
- References to :NEW and :OLD in database triggers cannot be used with LONG or LONG RAW columns.
- LONG and LONG RAW columns cannot be used in distributed SQL statements.
- LONG and LONG RAW columns cannot be replicated.

  Note: If 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 of LONG Datatype

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 two 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.

Using RAW and LONG RAW Datatypes

Note: The RAW and LONG RAW datatypes are provided for backward compatibility with existing applications. For new applications, you should use the BLOB and BFILE datatypes for large amounts of binary data. See Chapter 6, “Large Objects (LOBs)” for information about the BLOB and BFILE datatypes.

The RAW and LONG RAW datatypes store data that is not to be interpreted by Oracle (that is, not to be converted when moving data between different systems). 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.

Net8 and the Export and Import utilities do not perform character conversion when transmitting RAW or LONG RAW data. 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. For more information about restrictions on LONG RAW data, see "Restrictions on LONG and LONG RAW Data" on page 5-10.

ROWIDs and the ROWID Datatype

Every row in a nonclustered 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 pseudocolumn named ROWID.

> **See Also:** *Oracle8 Concepts* for general information about the ROWID pseudocolumn and the ROWID datatype.

Extended ROWID Format

The Oracle Server uses an *extended ROWID* format, which supports features such as table partitions, index partitions, and clusters.

The extended ROWID includes the following information:

- data object (segment) identifier
- datafile identifier
- block identifier
- row identifier

The data object identifier is an identification number that Oracle assigns to schema objects in the database, such as nonpartitioned tables or partitions. For example, the query

```
SELECT DATA_OBJECT_ID FROM DBA_OBJECTS
    WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP';
```

returns the data object identifier for the EMP table in the SCOTT schema. "The DBMS_ROWID Package" on page 10-79 describes other ways to get the data object identifier, using the DBMS_ROWID package functions.

Different Forms of the ROWID

Oracle documentation uses the term ROWID in different ways, depending on context. These uses are explained in this section.

Internal ROWID The internal ROWID format is an internal structure which holds information that the server code needs to access a row. The restricted internal ROWID is 6 bytes on most platforms; the extended ROWID is 10 bytes on these platforms.

ROWID Pseudocolumn Each table and nonjoined view has a pseudocolumn called ROWID. Statements such as

```
CREATE TABLE T1 (col1 ROWID);

INSERT INTO T1 SELECT ROWID FROM EMP WHERE empno = 7499;
```

return the ROWID pseudocolumn of the row of the EMP table that satisfies the query, and insert it into the T1 table.

External Character ROWID The extended ROWID pseudocolumn is returned to the client in the form of an 18-character string (for example, "AAAA8mAALAAAAQkAAA"), which represents a base 64 encoding of the components of the extended ROWID in a four-piece format, OOOOOOFFFBBBBBBRRR:

- OOOOOO: The **data object number** identifies the database segment (AAAA8m in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- FFF: The **datafile** that contains the row (file AAL in the example). File numbers are unique within a database.
- BBBBBB: The **data block** that contains the row (block AAAAQk 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.
- RRR: The **row** in the block (row AAA in the example).

There is no need to decode the external ROWID; you can use the functions in the DBMS_ROWID package to obtain the individual components of the extended ROWID.

> **See Also:** "The DBMS_ROWID Package" on page 10-79.

The restricted ROWID pseudocolumn is returned to the client in the form of an 18-character string with a hexadecimal encoding of the datablock, row, and datafile components of the ROWID.

External Binary ROWID Some client applications use a binary form of the ROWID. For example, OCI and some precompiler applications can map the ROWID to a 3GL structure on bind or define calls. The size of the binary ROWID is the same for extended and restricted ROWIDs. The information for the extended ROWID is included in an unused field of the restricted ROWID structure.

The format of the extended binary ROWID, expressed as a C struct, is:

```
struct riddef {
    ub4    ridobjnum; /* data obj#--this field is
                         unused in restricted ROWIDs */
    ub2    ridfilenum;
    ub1    filler;
    ub4    ridblocknum;
    ub2    ridslotnum;
}
```

ROWID Migration and Compatibility Issues

For backward compatibility, the restricted form of the ROWID is still supported. These ROWIDs exist in massive amounts of Oracle7 data, and the extended form of the ROWID is required only in global indexes on partitioned tables. New tables always get extended ROWIDs. .

> **See Also:** *Oracle8 Administrator's Guide.*

It is possible for an Oracle7 client to access an Oracle8 database. Similarly, an Oracle8 client can access an Oracle7 Server. A client in this sense can include a remote database accessing a server using database links, as well as a client 3GL or 4GL application accessing a server.

> **See Also:** FThe description of "ROWID_TO_EXTENDED Function" on page 10-85 has more information, as has *Oracle8 Migration*.

Accessing an Oracle7 Database from an Oracle8 Client The ROWID values that are returned are always restricted ROWIDs. Also, Oracle8 uses restricted ROWIDs when returning a ROWID value to an Oracle7 or earlier server.

The following ROWID functionality works when accessing an Oracle7 Server:

- selecting a ROWID and using the obtained value in a WHERE clause

- WHERE CURRENT OF cursor operations
- storing ROWIDs in user columns of ROWID or CHAR type
- interpreting ROWIDs using the hexadecimal encoding (not recommended, use the DBMS_ROWID functions)

Accessing an Oracle8 Database from an Oracle7 Client Oracle8 returns ROWIDs in the extended format. This means that you can only:

- select a ROWID and use it in a WHERE clause
- use WHERE CURRENT OF cursor operations
- store ROWIDs in user columns of CHAR(18) datatype

Import and Export It is not possible for an Oracle7 client to import an Oracle8 table that has a ROWID column (not the ROWID pseudocolumn), if any row of the table contains an extended ROWID value.

Trusted Oracle MLSLABEL Datatype

Trusted Oracle provides the MLSLABEL datatype, which stores Trusted Oracle's internal representation of labels generated by multilevel secure (MLS) operating systems. Trusted Oracle uses labels to control database access.

You can define a column using the MLSLABEL datatype for compatibility with Trusted Oracle applications, but the only valid value for the column in Oracle8 is NULL.

When you create a table in Trusted Oracle, a column called ROWLABEL is automatically appended to the table. This column contains a label of the MLSLABEL datatype for every row in the table.

See Also: *Trusted Oracle* documentation for more information about the MLSLABEL datatype, the ROWLABEL column, and Trusted Oracle.

ANSI/ISO, DB2, and SQL/DS Datatypes

You can define columns of tables in an Oracle database using ANSI/ISO, DB2, and SQL/DS datatypes. Oracle internally converts such datatypes to Oracle datatypes.

The ANSI datatype conversions to Oracle datatypes are shown in Table 5–3. The ANSI/ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, s defaults to 0.

Table 5–3 ANSI Datatype Conversions to Oracle Datatypes

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)

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.

Table 5–4 shows the DB2 and SQL/DS conversions.

Table 5–4 SQL/DS, DB2 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

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 below.

> **See Also:** If you are using Trusted Oracle, see "Data Conversion for Trusted Oracle" on page 5-21 for information about data conversions and the MLSLABEL datatype.

Rule 1: Assignments

For assignments, Oracle can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- NUMBER to VARCHAR2
- VARCHAR2 or CHAR to DATE
- DATE to VARCHAR2
- VARCHAR2 or CHAR to ROWID
- ROWID to VARCHAR2
- VARCHAR2 or CHAR to MLSLABEL
- MLSLABEL to VARCHAR2
- VARCHAR2 or CHAR to HEX
- HEX to VARCHAR2

The assignment succeeds if Oracle can convert the datatype of the value used in the assignment to that of the assignment's target.

For the examples in the following list, assume a package with a public variable and a table declared as in the following statements:

```
var1  CHAR(5);
CREATE TABLE table1 (col1 NUMBER);
```

- `variable := expression`

  The datatype of *expression* must be either 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*, and so on, must be either the same as or convertible to the datatypes of the 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 be either 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 be either 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.

Oracle can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- VARCHAR2 or CHAR to DATE

Character to NUMBER conversions succeed only if the character string represents a valid number. Character to DATE conversions succeed only if the character string satisfies the session default format, which is specified by the initialization parameter NLS_DATE_FORMAT.

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('1997-01-01','yyyy-mm-dd')
  ```

- WHERE clause conditions, such as

  ```
  WHERE rowid = 'AAAAaoAATAAAADAAA'
  ```

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.

Data Conversion for Trusted Oracle

In Trusted Oracle, labels are stored internally as compact binary structures. Trusted Oracle provides the TO_LABEL function that enables you to convert a label from its internal binary format to an external character format. To convert a label from character format to binary format in Trusted Oracle, you use the TO_CHAR function.

The TO_LABEL function is provided for compatibility with Trusted Oracle applications. It returns the NULL value in Oracle8.

> **See Also:** The *Trusted Oracle* documentation has more information about using the TO_LABEL and TO_CHAR functions to convert label formats.

6

Large Objects (LOBs)

Oracle8 provides support for defining and manipulating large objects (LOBs). Oracle8 extends SQL DDL and DML commands to create and update LOB columns in a table or LOB attributes of an object type. Further, Oracle8 provides Oracle Call Interface (OCI) and PL/SQL package APIs to perform random, piecewise operations on LOBs.

This chapter documents the extended SQL commands and the PL/SQL package API for LOBs. It also briefly mentions the OCI API for LOB manipulation, which is described in the *Oracle Call Interface Programmer's Guide*.

This chapter has two sections:

- *Introduction to LOBs* — introduces LOBs in Oracle8, and discusses the main aspects of LOB handling
- *LOB Reference* — contains the detailed list of all of the technical specifications for the PL/SQL `DBMS_LOB` package

Introduction to LOBs

Introduction Overview

This section introduces the treatment of LOBs in Oracle8 under the headings that are also laid out below. Although it is not made explicit in the text, the various issues can be grouped under a number of umbrella topics.

The first topic is one of general introduction:

- What are LOBs?
- Internal LOBs and External LOBs (BFILEs)
  - Internal LOBs
  - External LOBs (BFILEs)
- LOB Datatypes
  - Internal LOB datatypes
  - External LOB datatypes
- LOBs in comparison to LONG and LONG RAW types
- Packages for Working with LOBs

The second topic discusses steps involved in beginning to work with LOBs:

- Defining Internal and External LOBs for Tables (SQL DDL)
- Stipulating Tablespace and Storage Characteristics for Internal LOBs
  - Tablespace and LOB Index
  - PCTVERSION
  - CACHE / NO CACHE
  - LOGGING / NO LOGGING
  - CHUNK
- Initializing Internal LOBs (SQL DML)

The third topic deals with issues specific to handling external LOBs (BFILEs):

- Accessing External LOBs (SQL DML)

  - Initializing BFILES using BFILENAME() and OCIFileSetName()
  - The DIRECTORY object
  - DIRECTORY Name Specification
- BFILE Security
  - Ownership and security
  - SQL DDL for BFILE security
  - SQL DML for BFILE security
- Catalog Views on Directories
- Guidelines for DIRECTORY Usage
- Maximum number of open BFILES
- BFILEs in MTS mode
- Closing BFILEs after Program Termination

The fourth topic considers how LOBs are handled by way of *locators*:

- LOB Values and Locators
  - Inline storage of LOB values
  - LOB locators
  - Internal LOB locators
  - BFILE locators
- LOB Locator Operations
  - Setting the LOB Column / Attribute to contain a Locator
  - Accessing a LOB through a locator
  - Read consistent locators
  - Updated locators
  - LOB bind variables

The fifth topic is concerned with basic manipulation of LOBs:

- Efficient reads and writes of large amounts of LOB data

- Copying LOBs
  - Copying internal LOBs
  - Copying external LOBs
- Deleting LOBs
  - Deleting internal LOBs
  - Deleting external LOBs
- Copying Data from LONGs to LOBs

Finally, the last topic considers performance and optimization issues in a client/server environment:

- LOBs in the Object Cache
- The LOB Buffering Subsystem
- User Guidelines for Best Performance Practices
- Working with Varying-Width Character Data

What Are LOBs?

Consider the following application scenarios:

Application Scenario 1: A law firm wishes to manage production of a significant case by means of a database. The lawyers are aware that the information will include x-rays (image data), expert analysis (character text), depositions (audio/video), and drawings (graphics).During the course of the trial they also come to utilize computer-simulated events (animation).

Application Scenario 2: A broadcast station wishes to manage production of its feature programs by means of a database. The program managers are aware that this information commonly includes photographs (image data), interviews (audio/video), sound-effects (sound waveforms), music (sound waveforms), and script (character text). With the advance of digitizing and storage technology, they also find it possible to include legacy silent-film (video).

Application Scenario 3: A geological survey team looking for oil under the sea wishes to manage its projects by means of a database. The project managers are aware that the information will include satellite pictures (image data) with complex overlay drawings (image data), sonar recordings along with their graphic representations (sound wave forms and image data), and chemical analysis (image data and character text).During the course of the project they also come to employ computer modeling of likely weather conditions (character text and image data).

Although each of these scenarios is drawn from a different domain, it is easy to see how management of multiple media is becoming commonplace in business applications. This is relevant to this chapter because Oracle8 supports LOBs — *large objects* which can hold up to 4 gigabytes of RAW, binary data (e.g., graphic images, sound waveforms, video clips, etc.) or character text data.

Oracle8 regards LOBs as being of two kinds depending on their location with regard to the database — *internal* LOB*s* and *external* LOB*s (BFILEs)*. When the term *LOB* is used without an identifying prefix term, it refers to both internal and external LOBs. Data stored in a LOB is termed the LOB's *value*.

Internal LOBs and External LOBs (BFILEs)

Internal LOBs

Internal LOBs, as their name suggests, are stored in the database tablespaces in a way that optimizes space and provides efficient access. Internal LOBs use copy semantics and participate in the transactional model of the server. Internal LOBs are also recoverable in the event of transaction or media failure. That is, all the ACID properties that pertain to using database objects pertain to internal LOBs also. This means that any changes to a internal LOB value can be committed or rolled back.

External LOBs (BFILEs)

External LOBs, also referred to as *BFILES*, are large binary data objects stored in operating system files outside of the database tablespaces. These files use reference semantics. They may be located on hard disks, CDROMs, PhotoCDs or any such device, but a single LOB may not extend from one device to another. The SQL datatype *BFILE* is supported in Oracle8 SQL and PL/SQL to enable *read-only* byte stream I/O access to large files existing on the filesystem of the *database server*. The Oracle Server can access them provided the underlying server operating system supports a stream-mode access to these files.

WARNING: External LOBs do not participate in transactions. Any support for integrity and durability must be provided by the underlying file system as governed by the operating system.

LOBs in Comparison to LONG and LONG RAW Types

LOBs are similar to LONG and LONG RAW types, but differ in the following ways:

- Multiple LOBs are allowed in a single row.
- LOBs can be attributes of a user-defined datatype (object).
- Only the LOB locator is stored in the table column; BLOB and CLOB data can be stored in separate tablespaces and BFILE data is stored as an external file.
- When you access a LOB column, it is the locator which is returned.
- A LOB can be up to 4 gigabytes in size. The BFILE maximum is operating system dependent, but cannot exceed 4 gigabytes. The valid accessible range is 1 to (2^{32}-1).
- LOBs let you access and manipulate data in a random, piece-wise manner.

Packages for Working with LOBs

You can make changes to the entire values of internal LOBs through direct SQL DML. You can make to an entire internal LOB, or piecewise to the beginning, middle or end of an internal LOB through the OCI, or through the PL/SQL DBMS_LOB APIs. It is possible to access both internal and external LOBs for read purposes and also write to internal LOBs.

- The OCI LOB interface is described briefly in "Using the OCI to Manipulate LOBs" on page 6-63, and more extensively in the *Oracle Call Interface Programmer's Guide*.
- The PL/SQL DBMS_LOB API is described in "DBMS_LOB Package" on page 6-66.

LOB Datatypes

Internal LOB Datatypes

There are three SQL datatypes for defining instances of internal LOBs:

- BLOB, a LOB whose value is composed of unstructured binary ("raw") data.

- CLOB, a LOB whose value is composed of single-byte fixed-width character data that corresponds to the database character set defined for the Oracle8 database.
- NCLOB, a LOB whose value is composed of fixed-width multi-byte character data that corresponds to the national character set defined for the Oracle8 database.

Varying width character data is not supported for BLOBs, CLOBs and NCLOBs.

> **See Also:** "Working with Varying-Width Character Data" on page 6-57

External LOB Datatype

There is one external SQL LOB datatype:

- BFILE, a LOB whose value is composed of binary ("raw") data, and is stored outside of the database tablespaces in a server-side operating system file.

Defining Internal and External LOBs for Tables

It is possible to incorporate LOBs into tables in two ways.

- LOBs may be columns in a table — the case in which the large object is 'in relation' with other data entities.
- LOBs may be attributes of an object — the case in which a data entity (i.e. an object type) has one or more LOBs as attributes.

In both cases SQL DDL is used — to define LOB columns in a table and LOB attributes in an object type. Refer to the *Oracle8 SQL Reference* for information about using LOBs in the following DDL commands:

- CREATE TABLE and ALTER TABLE
  - BLOB, CLOB, NCLOB and BFILE columns
  - LOB storage clause for internal LOB columns / attributes
- CREATE TYPE and ALTER TYPE
  - BLOB, CLOB, and BFILE attributes (noting that NCLOBs cannot be attributes in an object type).

The following code fragment describes creating the table, lob_table. We refer to this example throughout the text.

```
CREATE TABLE lob_table (
  key_value      INTEGER,
  b_lob          BLOB,
  c_lob          CLOB,
  n_lob          NCLOB,
  f_lob          BFILE);
```

Stipulating Tablespace and Storage Characteristics for Internal Lobs

When defining LOBs in a table, you can explicitly indicate the tablespace and storage characteristics for each internal LOB. There are no extra tablespace or storage characteristics for external LOBs since they are not stored in the database.

Specifying a name for the LOB data segment and the LOB index makes for a much more intuitive working environment. When querying the LOB data dictionary views USER_LOBS, ALL_LOBS, DBA_LOBS (see *Oracle8 Reference*), you see the LOB data segment and LOB index names that you chose instead of system-generated names that are non-intuitive.

The LOB storage characteristics that can be specified for a LOB column or a LOB attribute include PCTVERSION, CACHE, NOCACHE, LOGGING, NOLOGGING, CHUNK and ENABLE / DISABLE STORAGE IN ROW. For most users, defaults for these storage characteristics will be sufficient. If you want to fine-tune LOB storage, you should consider the following guidelines.

Tablespace and LOB Index

Best performance for LOBs can be achieved by specifying storage for LOBs in a tablespace that is different from the one used for the table that contains the LOB. If many different LOBs will be accessed frequently, it may also be useful to specify a separate tablespace for each LOB column / attribute in order to reduce device contention.

The LOB index is an internal structure that is strongly associated with the LOB storage. This implies that a user may not drop the LOB index and rebuild it. Note that the LOB index cannot be altered through the ALTER INDEX statement although you can alter it through the ALTER TABLE statement. However, you may not rename the LOB index. The system determines which tablespace to use for the LOB data and LOB index depending on the user specification in the LOB storage clause:

- If you do not specify a tablespace for the LOB data nor for the LOB index, the table's tablespace is used for both the LOB data and the LOB index.
- If you specify a tablespace for the LOB data but not for the LOB index, both the LOB data and index use the tablespace that was specified for the LOB data.
- If you specify a tablespace for the LOB index but not the LOB data, the LOB index uses the specified tablespace and the LOB data uses the table's tablespace.
- If you specify a tablespace for the LOB data and the LOB index, the LOB data and index use the specified tablespaces respectively.

Specifying a separate tablespace for the LOB storage segments will allow for a decrease in contention on the table's tablespace. In some extreme cases, it may even be beneficial to use three separate tablespaces — one for the table data, one for the LOB data segments, and one for the LOB index segments. This would be useful if certain LOB data is to be accessed very frequently. Normally, using two tablespaces

— one for the table data, and one for the LOB data and LOB index — should be sufficient.

PCTVERSION

When a LOB is modified, a new version of the LOB page is made in order to support consistent read of prior versions of the LOB value.

PCTVERSION is the percent of all used LOB data space that can be occupied by old versions of LOB data pages. As soon as old versions of LOB data pages start to occupy more than the PCTVERSION amount of used LOB space Oracle will try to reclaim the old versions and reuse them. In other words, PCTVERSION is the percent of used LOB data blocks that is available for versioning of old LOB data.

Default: 10 (%) Minimum: 0 (%) Maximum: 100 (%)

One way of approximating PCTVERSION is to set PCTVERSION =% of LOBs updated at any given point in time * % of each LOB updated whenever a LOB is updated * % of LOBs being read at any given point in time. Basically, the idea is to allow for a percentage of LOB storage space to be used as old versions of LOB pages so that readers will be able to get consistent reads of data that has been updated.

Example 1: *Several LOB updates concurrent with heavy reads of LOBs.*

set PCTVERSION = 20%

Setting PCTVERSION to twice the default allows more free pages to be used for old versions of data pages. Since large queries may require consistent reads of LOBs, it is useful to keep more old versions of LOB pages around. Of course, LOB storage may grow some because Oracle will not be reusing free pages aggressively.

Example 2: *LOBs are created and written just once and are primarily read-only afterwards. Updates are infrequent.*

set PCTVERSION = 5% or lower

The more infrequent and smaller the LOB updates are, the less space that needs to be reserved for old copies of LOB data. If existing LOBs are known to be read-only, we could safely set PCTVERSION to 0% since there would never be any pages needed for old versions of data.

CACHE / NOCACHE

Use the CACHE option on LOBs if the same LOB data will be accessed frequently. Use the NOCACHE option (the default) if LOB data will be read only once, or infrequently.

LOGGING / NOLOGGING

[NO] LOGGING has a similar application with regard to using LOBs as it does for other table operations. In the normal case, if the [NO]LOGGING clause is omitted, this means that neither NO LOGGING nor LOGGING is specified and the logging attribute of the table or table partition defaults to the logging attribute of the tablespace in which it resides.

For LOBs, there is a further alternative depending on how CACHE is stipulated.

- If the [NO]LOGGING clause is omitted and CACHE is specified, LOGGING is automatically implemented (because you cannot have CACHE NOLOGGING).
- If the [NO]LOGGING clause is omitted and CACHE is not specified, the process defaults in the same way as it does for tables and partitioned tables i.e.,the [NO]LOGGING value is obtained from the tablespace in which the LOB value resides.

The following issues should also be kept in mind.

- LOBs will always generate undo for LOB index pages. Regardless of whether LOGGING or NOLOGGING is set LOBs will never generate rollback information (undo) for LOB data pages because old LOB data is stored in versions. Rollback information that is created for LOBs tends to be small because it is only for the LOB index page changes.
- When LOGGING is set Oracle will generate full redo for LOB data pages. NOLOGGING is intended to be used when a customer does not care about media recovery. Thus, if the disk / tape / storage media fails, you will not be able to recover your changes from the log since the changes were never logged.

  An example of when NOLOGGING is useful is bulk loads or inserts. For instance, when loading data into the LOB, if you don't care about redo and can just start the load over if it fails, set the LOB's data segment storage characteristics to NOCACHE NOLOGGING. This will give good performance for the initial load of data. Once you have completed loading the data, you can use ALTER TABLE to modify the LOB storage characteristics for the LOB data segment to be what you really want for normal LOB operations -- i.e. CACHE or NOCACHE LOGGING.

  Note: CACHE implies that you also get LOGGING.

CHUNK

Set CHUNK to the number of blocks of LOB data that will be accessed at one time i.e. the number of blocks that will be read / written via OCILobRead(), OCILob-

Write(), DBMS_LOB.READ(), or DBMS_LOB.WRITE() during one access of the LOB value. For example, if only 1 block of LOB data is accessed at a time, set CHUNK to the size of one block. For example, if the database block size is 2K, then set CHUNK to 2K.

If you explicitly specify the storage characteristics for the LOB, make sure that INITIAL and NEXT for the LOB data segment storage are set to a size that is larger than the CHUNK size. For example, if the database block size is 2K and you specify a CHUNK of 8K, make sure that the INITIAL and NEXT are bigger than 8K and preferably considerably bigger (for example, at least 16K).

Put another way: If you specify a value for INITIAL, NEXT or the LOB CHUNK size, make sure that:

- CHUNK <= NEXT

and

- CHUNK <= INITIAL

ENABLE | DISABLE STORAGE IN ROW

You use the ENABLE | DISABLE STORAGE IN ROW clause to indicate whether the LOB should be stored inline (i.e. in the row) or out of line. You may not alter this specification once you have made it: if you ENABLE STORAGE IN ROW, you cannot alter it to DISABLE STORAGE IN ROW and vice versa. The default is ENABLE STORAGE IN ROW.

The maximum amount of LOB data that will be stored in the row is the maximum VARCHAR size (4000). Note that this includes the control information as well as the LOB value. If the user indicates that the LOB should be stored in the row, once the LOB value and control information is larger than 4000, the LOB value is automatically moved out of the row.

This suggests the following guideline. If the LOB is small (i.e. < 4000 bytes), then storing the LOB data out of line will decrease performance. However, storing the LOB in the row increases the size of the row. This will impact performance if the user is doing a lot of base table processing, such as full table scans, multi-row accesses (range scans) or many UPDATE/SELECT to columns other than the LOB columns. If the user doesn't expect the LOB data to be < 4000, i.e. if all LOBs are big, then the default is the best choice since

(a) the LOB data is automatically moved out of line once it gets bigger than 4000 (which will be the case here since the LOB data is big to begin with), and

(b) performance will be slightly better since we still store some control information in the row even after we move the LOB data out of the row.

Initializing Internal LOBs (SQL DML)

You can set an internal LOB — that is, a LOB column in a table, or a LOB attribute in an object type defined by you— to be empty, or NULL. An empty LOB stored in a table is a LOB of zero length that has a locator. If you SELECT from an empty LOB column / attribute, you get back a locator which you can use to populate the LOB with data via the OCI or DBMS_LOB routines. This is discussed in more detail below.

Alternatively, LOB columns, but not LOB attributes, may be initialized to a value. Which is to say — internal LOB attributes differ from internal LOB columns in that LOB attributes may not be initialized to a value other than NULL or empty. As discussed below, an external LOB (i.e. BFILE) can be initialized to NULL or to a filename.

For example, let us say that you create the table, *lob_table:*

```
CREATE TABLE lob_table (
  key_value      INTEGER,
  b_lob          BLOB,
  c_lob          CLOB,
  n_lob          NCLOB,
  f_lob          BFILE);
```

You can initialize the LOBs by using the following SQL INSERT statement:

```
INSERT INTO lob_table VALUES (1001, EMPTY_BLOB(), NULL,
   EMPTY_CLOB(), NULL);
```

This sets the value of *b_lob* and *n_lob* to an empty value, and sets *c_lob* and *f_lob* to NULL.

Setting the LOB to NULL

You may want to set the internal LOB value to NULL upon inserting the row in cases where you do not have the LOB data at the time of the INSERT and / or if you want to issue a SELECT statement thereafter such as:

```
SELECT * FROM a_table WHERE a_lob_col != NULL;
```

or

```
SELECT * FROM a_table WHERE a_lob_col == NULL;
```

However, the drawback to this approach is that you must then issue a SQL UPDATE statement to set the NULL LOB column to EMPTY_BLOB() / EMPTY_CLOB() or to a value (e.g. 'abc') for internal LOBs or to a filename for external LOBs. You cannot call

the OCI or the PL/SQL DBMS_LOB functions on a NULL LOB. These functions only work with a locator and if the LOB column is NULL, there is no locator in the row.

Setting the internal LOB to empty

The other option is for you to set the LOB value to empty by using the function EMPTY_BLOB () / EMPTY_CLOB() in the INSERT statement:

```
INSERT INTO a_table values (empty_blob());
```

Even better is to use the RETURNING clause (thereby eliminating a round trip that is necessary for the subsequent SELECT), and then immediately call OCI or the PL/SQL DBMS_LOB functions to populate the LOB with data.

> **See Also:** "EMPTY_BLOB() and EMPTY_CLOB() Functions" on page 6-59

Accessing External LOBs (SQL DML)

Directory Object

The DIRECTORY object enables administering the access and usage of BFILEs in an Oracle8 Server (see the CREATE DIRECTORY command in the *Oracle8 Reference).* A DIRECTORY specifies a *logical alias name* for a physical directory on the server's file-system under which the file to be accessed is located. You can access a file in the server's filesystem only if granted the required access privilege on the DIRECTORY object.

The DIRECTORY object also provides the flexibility to manage the locations of the files, instead of forcing you to hardcode the absolute pathnames of the physical files in your applications. A DIRECTORY alias is used in conjunction with the BFILENAME() function (in SQL and PL/SQL), or the OCILobFileSetName() (in OCI) for initializing a BFILE locator.

WARNING: Oracle does not verify that the directory and pathname you specify actually exist. You should take care to specify a valid directory in your operating system. If your operating system uses case-sensitive pathnames, be sure you specify the directory in the correct format. There is no need to specify a terminating slash (e.g., *`/tmp/`* is not necessary, simply use *`/tmp`*).

Initializing BFILES using BFILENAME()

In order to associate an operating system file to a BFILE, it is necessary to first create a DIRECTORY object which is an alias for the full pathname to the operating system file.

See Also: "Directory Object" on page 6-15.

You use Oracle8 SQL DML to associate existing operating system files with the relevant database records of a particular table. You can use the SQL INSERT statement to initialize a BFILE column to point to an existing file in the server's filesystem, and you can use a SQL UPDATE statement to change the reference target of the BFILE. You can also initialize a BFILE to NULL and then update it later to refer to an operating system file via the BFILENAME() function. OCI users can also use OCILobFIleSetName() to initialize a BFILE locator variable that is then used in the VALUES clause of an INSERT statement.

For example, the following statements associate the files *image1.gif* and *image2.gif* with records having key_value of 21 and 22 respectively. 'IMG' is a DIRECTORY object that represents the physical directory under which *image1.dif* and i*mage2.dif* are stored.

```
INSERT INTO lob_table VALUES
     (21, NULL, NULL, NULL, BFILENAME('IMG', 'image1.gif'));
 INSERT INTO lob_table VALUES
     (22, NULL, NULL, NULL, BFILENAME('IMG', 'image2.gif'));
```

The UPDATE statement below changes the target file to *image3.gif* for the row with key_value 22**.**

```
UPDATE lob_table SET f_lob = BFILENAME('IMG', 'image3.gif')
    WHERE key_value = 22;
```

BFILENAME() is a built-in function that is used to initialize the BFILE column to point to the external file.

See Also: "BFILENAME() Function" on page 6-60.

Once physical files are associated with records using SQL DML, subsequent read operations on the BFILE can be performed using PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILES, and so they cannot be updated or deleted through BFILES.

As a consequence of the reference-based semantics for BFILEs, it is possible to have multiple BFILE columns in the same record or different records referring to the same file. For example, the UPDATE statements below set the BFILE column of the

row with key_value 21 in lob_table to point to the same file as the row with key_value 22.

```
UPDATE lob_table
   SET f_lob = (SELECT f_lob FROM lob_table WHERE key_value = 22)
   WHERE key_value = 21;
```

DIRECTORY Name Specification

The naming convention followed by Oracle8 for DIRECTORY objects is the same as that done for tables and indexes. That is, normal identifiers are interpreted in uppercase, but delimited identifiers are interpreted as is. For example, the following statement

```
CREATE DIRECTORY scott_dir AS '/usr/home/scott';
```

creates a directory object whose name is 'SCOTT_DIR' (in uppercase). But if a delimited identifier is used for the DIRECTORY name, as shown in the following statement

```
CREATE DIRECTORY "Mary_Dir" AS '/usr/home/mary';
```

the directory object's name is 'Mary_Dir'. Use 'SCOTT_DIR' and 'Mary_Dir' when calling BFILENAME(). For example:

```
BFILENAME('SCOTT_DIR', 'afile')
BFILENAME('Mary_Dir', 'afile')
```

BFILE Security

This section introduces the BFILE security model and the associated SQL DDL and DML. The main features for BFILE security in Oracle 8.0 are:

- SQL DDL to CREATE and REPLACE / ALTER a DIRECTORY object.
- SQL DML to GRANT and REVOKE the READ system and object privileges on DIRECTORY objects.

Ownership and Privileges

The DIRECTORY is a *system owned* object. For more information on system owned objects, see *Oracle8 SQL Reference*. Oracle8 supports two new system privileges, which are granted only to the DBA account:

- CREATE ANY DIRECTORY — for creating or altering the directory object creation

- DROP ANY DIRECTORY — for deleting the directory object

The READ privilege on the DIRECTORY object allows you to read files located under that directory. The creator of the DIRECTORY object automatically earns the READ privilege. If you have been granted the READ privilege with GRANT option, you may in turn grant this privilege to other users/roles and add them to your privilege domains.

It is important to note that the READ privilege is defined only on the DIRECTORY object. The physical directory that it represents may or may not have the corresponding operating system privileges (*read* in this case) for the Oracle Server process. It is the DBA's responsibility to ensure that the physical directory exists, and *read* permission for the Oracle Server process is enabled on the file, the directory, and the path leading to it. It is also the DBA's responsibility to make sure that the directory remains available, and the *read* permission remains enabled, for the entire duration of file access by database users.

The privilege just implies that as far as the Oracle8 Server is concerned, you may read from files in the directory. These privileges are checked and enforced by the PL/SQL DBMS_LOB package and OCI APIs at the time of the actual file operations.

WARNING: Since the CREATE ANY DIRECTORY and DROP ANY DIRECTORY privileges potentially expose the server filesystem to all database users, the DBA should be prudent in granting these privileges to normal database users to prevent any accidental or malicious security breach.

SQL DDL for BFILE security

Refer to the *Oracle8 SQL Reference* for information about the following SQL DDL commands that create, replace, and drop directory objects:

- CREATE DIRECTORY
- DROP DIRECTORY

SQL DML for BFILE security

Refer to the *Oracle8 SQL Reference* for information about the following SQL DML commands that provide security for BFILEs:

- GRANT (system privilege)
- GRANT (object privilege)

- REVOKE (system privilege)
- REVOKE (object privilege)
- AUDIT (new statements)
- AUDIT (schema objects)

Catalog Views on Directories

Catalog views are provided for directory objects to enable users to view object names and their corresponding paths and privileges. The supported views are:

- ALL_DIRECTORIES (OWNER, DIRECTORY_NAME, DIRECTORY_PATH)

  This view describes all the directories accessible to the user.

- DBA_DIRECTORIES(OWNER, DIRECTORY_NAME, DIRECTORY_PATH)

  This view describes all the directories specified for the entire database.

Guidelines for DIRECTORY Usage

The main goal of the DIRECTORY feature in Oracle8 is to enable a simple, flexible, non-intrusive, yet secure mechanism for the DBA to manage access to large files in the server filesystem. But to realize this goal, it is very important that the DBA follow these guidelines when using directory objects:

- A DIRECTORY should not be mapped to physical directories which contain Oracle datafiles, control files, log files, and other system files. Tampering with these files (accidental or otherwise) could potentially corrupt the database or the server operating system.
- The system privileges such as CREATE ANY DIRECTORY (granted to the DBA initially) should be used carefully and not granted to other users indiscriminately. In most cases, only the database administrator should have these privileges.
- Privileges on directory objects should be granted to different users carefully. The same holds for the use of the WITH GRANT OPTION clause when granting privileges to users.
- DIRECTORY objects should not be arbitrarily dropped or replaced when the database is in operation. If this were to happen, DBMS_LOB or OCI operations

from all sessions on all files associated with this directory object will fail. Further, if a DROP or REPLACE command is executed before these files could be successfully closed, the references to these files will be lost in the programs, and system resources associated with these files will not be released until the session(s) is shutdown.

The only recourse left to PL/SQL users, for example, will be to either execute a program block that calls DBMS_LOB FILECLOSEALL() and restart their file operations, or exit their sessions altogether. Hence, it is imperative that you use these commands with prudence, and preferably during maintenance downtimes.

> **See Also:** "DBMS_LOB.FILECLOSEALL() Procedure" on page 6-81

- Similarly, revoking an user's privilege on a directory using the REVOKE statement causes all subsequent operations on dependent files from the user's session to fail. Either you must re-acquire the privileges to close the file, or execute a FILECLOSEALL() in the session and restart the file operations.

In general, using DIRECTORY objects for managing file access is an extension of system administration work at the operating system level. With some planning, files can be logically organized into suitable directories that have read privileges for the Oracle process, DIRECTORY objects can be created with READ privileges that map to these physical directories, and specific database users granted access to these directories.

Maximum Number of Open BFILEs

A limited number of BFILEs can be open simultaneously per session. The maximum number is specified by a new initialization parameter, the SESSION_MAX_OPEN_FILES parameter.

SESSION_MAX_OPEN_FILES defines an upper limit on the number of simultaneously open files in a session. The default value for this parameter is 10. That is, a maximum of 10 files can be opened simultaneously per session if the default value is utilized. The database administrator can change the value of this parameter in the init.ora file**.** For example:

```
SESSION_MAX_OPEN_FILES=20
```

BFILEs in MTS Mode

Oracle8 release 8.0 does not support session migration for BFILEs in MTS mode. This implies that operations on open BFILEs can persist beyond the end of a call to an MTS server. Sessions involving BFILE operations need to be bound to one shared server, they cannot migrate from one server to another.

Closing BFILEs after Program Termination

It is the user's responsibility to close any opened file(s) after normal or abnormal termination of a PL/SQL program block or OCI program. So, for instance, for every DBMS_LOB FILEOPEN call, there must be a matching DBMS_LOB FILECLOSE call. You should close open files before the termination of a PL/SQL block or OCI program, and also in situations which have raised errors. The exception handler should make provision to close any files that were opened before the occurrence of the exception or abnormal termination.

If this is not done, Oracle will consider these files unclosed, and if the number of unclosed files exceeds the SESSION_MAX_OPEN_FILES value then you will not be able to open any more files in the session. To close all open files, use the FILECLOSEALL call.

> **See Also:** "DBMS_LOB General Usage Notes" on page 6-69 for more details on PL/SQL programming

LOB Value and Locators

Inline storage of the LOB value

Data stored in a LOB is termed the LOB's *value*. The value of an internal LOB may or may not be stored inline with the other row data. If the internal LOB value is less than approximately 4000 bytes, then the value is stored inline; otherwise it is stored outside the row. Since LOBs are intended to be large objects, inline storage will only be relevant if your application mixes 'small' and 'large' LOBs.

As mentioned above ("ENABLE | DISABLE STORAGE IN ROW" on page 6-12), the LOB value is automatically moved out of the row once it extends beyond approximately 4000 bytes.

LOB locators

Regardless of where the value of the internal LOB is stored, a *locator* is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value. A LOB *locator* is a locator to an internal LOB while a *BFILE locator* is a locator

to an external LOB. When the term *locator* is used without an identifying prefix term, it refers to both LOB locators and BFILE locators.

Internal LOB Locators

For internal LOBs, the LOB column stores a locator to the LOB's value which is stored in a database tablespace. Each LOB column / attribute for a given row has its own distinct LOB locator and copy of the LOB value stored in the database tablespace.

External LOB Locators (BFILE Locators)

For BFILEs, the value is stored in a server-side operating system file, i.e. external to the database. The BFILE locator that refers to that file is stored in the row. If a BFILE locator variable that is used in a DBMS_LOB FILEOPEN() (for example L1) is assigned to another locator variable, (for example L2), both L1 and L2 point to the same file. This means that two rows in a table with a BFILE column can refer to the same file or to two distinct files — a fact that the canny developer might turn to advantage, but which could well be a pitfall for the unwary.

A BFILE locator variable in a PL / SQL or OCI program behaves like any other automatic variable. With respect to file operations, it behaves like a *file descriptor* available as part of the standard I / O library of most conventional programming languages. This implies that once you define and initialize a BFILE locator, and open the file pointed to by this locator, all subsequent operations until the closure of this file must be done from within the same program block using this locator or local copies of this locator.

The BFILE locator variable can be used, just as any scalar, as a parameter to other procedures, member methods, or external function callouts. However, it is recommended that you open and close a file from the same program block at the same nesting level, in PL / SQL and OCI programs.

LOB Locator Operations

Setting the LOB Column/Attribute to contain a locator

Before you can start writing data to a internal LOB, the LOB column / attribute must be made non-null, that is, it must contain a locator. Similarly, before you can start accessing the BFILE value, the BFILE column / attribute must be made non-null.

- For internal LOBs, you can accomplish this by initializing the internal LOB to empty in an INSERT / UPDATE statement using the functions EMPTY_BLOB() for BLOBs or EMPTY_CLOB() for CLOBs and NCLOBs.

> **See Also:** "EMPTY_BLOB() and EMPTY_CLOB() Functions" on page 6-59.

- For external LOBs, you can initialize the BFILE column to point to an external file by using the BFILENAME() function.

  > **See Also:** "BFILENAME() Function" on page 6-60.

Invoking the EMPTY_BLOB() or EMPTY_CLOB() function in and of itself does not raise an exception. However, using a LOB locator that was set to empty to access or manipulate the LOB value in any PL/SQL DBMS_LOB or OCI routine will raise an exception. Valid places where empty LOB locators may be used include the VALUES clause of an INSERT statement and the SET clause of an UPDATE statement.

The following INSERT statement

- sets *b_lob* to NULL,
- populates *c_lob* with the character string 'abcde',
- sets n_lob to NULL, and
- initializes *f_lob* to point to the file 'scott.dat' located under the logical directory 'SCOTT_DIR' (see the CREATE DIRECTORY command in the *Oracle8 Reference)*. Character strings are inserted using the default character set for the instance.

```
INSERT INTO lob_table VALUES (1002, NULL 'abcde',
    NULL,BFILENAME('SCOTT_DIR', 'scott.dat'));
```

Similarly, given a table *person_objcol_table* one of whose columns is an object with LOB attributes, the LOB attributes can be initialized to NULL or set to empty as shown below:

```
INSERT INTO person_objcol_table VALUES (1001,person_type
    ('Scott', EMPTY_CLOB(), EMPTY_BLOB(),
    BFILENAME('SCOTT_DIR', 'scott.dat')));
```

Accessing a LOB through a locator

SELECTing a LOB Performing a SELECT on a LOB returns the locator instead of the LOB value. In the following PL/SQL fragment you select the LOB locator for *b_lob* and place it in the PL/SQL locator variable *image1* defined in the program block. When you use PL/SQL DBMS_LOB functions to manipulate the LOB value, you refer to the LOB using the locator.

```
DECLARE
    image1       BLOB;
    image_no     INTEGER := 101;
BEGIN
    SELECT b_lob INTO image1 FROM lob_table
        WHERE key_value = image_no;
    DBMS_OUTPUT.PUT_LINE('Size of the Image is: ' ||
        DBMS_LOB.GETLENGTH(image1));
    -- more LOB routines
END;
```

In using OCI, locators are mapped to locator pointers which are used to manipulate the LOB value. As mentioned before, the OCI LOB interface is described briefly in "Using the OCI to Manipulate LOBs" on page 6-63, and more extensively in the *Oracle Call Interface Programmer's Guide.*

Locking an Internal LOB before Updating Prior to *updating* a LOB value via the PL/SQL DBMS_LOB package or the OCI, *you must lock the row* containing the LOB. While the SQL INSERT and UPDATE statements implicitly lock the row, locking is done explicitly by means of a SQL SELECT FOR UPDATE statement in SQL and PL/SQL programs, or by using an OCI pin or lock function in OCI programs.

Read consistent locators

Oracle provides the same read consistency mechanisms for LOBs as for all other database reads and updates (refer to *Oracle8 Concepts* for general information about read consistency). However, read consistency has some special applications to LOB locators that need to be clearly understood.

A SELECTed locator, regardless of the existence of the FOR UPDATE clause, becomes a *read consistent locator,* and remains a read consistent locator until the LOB value is updated through that locator. A read consistent locator contains the snapshot environment as of the point in time of the SELECT.

This has some complex implications. Let us say that you have created a read consistent locator (L1) by way of a SELECT operation. In reading the value of the internal LOB through L1, the LOB is read as of the point in time of the SELECT statement even if the SELECT statement includes a FOR UPDATE. Further, if the LOB value is updated through a different locator (L2) in the same transaction, L1 does not see L2's updates. In addition, L1 will not see committed updates made to the LOB through *another* transaction.

Furthermore, if the read consistent locator L1 is copied to another locator L2 (for example, by a PL/SQL assignment of two locator variables — L2:= L1), then L2 becomes a read consistent locator along with L1 and any data read is read *as of the point in time of the SELECT for L1*.

Clearly you can utilize the existence of multiple locators to access different transformations of the LOB value. However, in taking this course, you must be careful to keep track of the different values accessed by different locators. The following code demonstrates the relationship between read-consistency and updating in a simple example.

Using *lob_table* as defined above and PL/SQL, three CLOBs are created as potential locators: clob_selected, clob_updated and clob_copied.

- At the time of the first SELECT INTO (at t1), the value in *c_lob* is associated with the locator *clob_selected.*
- In the second operation (at t2), the value in *c_lob* is associated with the locator *clob_updated.* Since there has been no change in the value of *c_lob* between t1 and t2, both *clob_selected* and *clob_updated* are read consistent locators that effectively have the same value even though they reflect snapshots taken at different moments in time.
- The third operation (at t3) copies the value in *clob_selected* to *clob_copied.* At this juncture, all three locators see the same value. The example demonstrates this with a series of dbms_lob.read calls.
- At this juncture (at t4), the program utilizes dbms_lob.write to alter the value in *clob_updated,* and a dbms_lob.read reveals a new value.
- However, a dbms_lob.read of the value through *clob_selected* (at t5) reveals that it is a read consistent locator, continuing to refer to the same value as of the time of its SELECT.
- Likewise, a dbms_lob.read of the value through *clob_copied* (at t6) reveals that it is a read consistent locator, continuing to refer to the same value as *clob_selected.*

Example of a Read Consistent Locator

```
INSERT INTO lob_table
   VALUES (1, NULL, 'abcd', NULL, NULL);

COMMIT;

DECLARE
```

```
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);

BEGIN
  -- At time t1:
  SELECT c_lob INTO clob_selected
     FROM lob_table
     WHERE key_value = 1;

  -- At time t2:
  SELECT c_lob INTO clob_updated
     FROM lob_table
     WHERE key_value = 1
     FOR UPDATE;

  -- At time t3:
  clob_copied := clob_selected;
  -- After the assignment, both the clob_copied and the
  -- clob_selected have the same snapshot as of the point in time
  -- of the SELECT into clob_selected

  -- Reading from the clob_selected and the clob_copied will
  -- return the same LOB value. clob_updated also sees the same
  -- LOB value as of its select:
  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
```

```
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t4:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
       buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'

  -- At time t5:
  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t6:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

Updated locators

When you update the value of the internal LOB through the LOB locator (L1), L1 (that is, the *locator* itself) is updated to contain the current snapshot environment *as of the point in time after the operation was completed* on the LOB value through the locator L1. L1 is then termed an *updated locator*. This operation allows you to see your own changes to the LOB value on the next read through the *same locator, L1*.

> **Note:** the snapshot environment in the locator is *not* updated if the locator is used to merely read the LOB value. It is only updated *when you modify* the LOB value through the locator via the PL/SQL DBMS_LOB package or the OCI LOB APIs.

Any committed updates made by a different transaction are seen by L1 only if your transaction is a read-committed transaction and if you use L1 to update the LOB value after the other transaction committed.

> **Note:** When you update an internal LOB's value, the modification is always made to the most current LOB value.

Updating the value of the internal LOB through the OCI LOB APIs or the PL/SQL DBMS_LOB package can be thought of as updating the LOB value *and then reselecting* the locator that refers to the new LOB value.

Note that updating the LOB value through SQL is merely an UPDATE statement. It is up to you to do the reselect of the LOB locator or use the RETURNING clause in the UPDATE statement (see the *PL/SQL User's Guide and Reference*) so that the locator can see the changes made by the UPDATE statement. Unless you reselect the LOB locator or use the RETURNING clause, you may think you are reading the latest value when this is not the case. For this reason you should avoid mixing SQL DML with OCI and DBMS_LOB piecewise operations.

Using *lob_table* as defined above, a CLOB locator is created: clob_selected.

- At the time of the first SELECT INTO (at t1), the value in *c_lob* is associated with the locator *clob_selected.*
- In the second operation (at t2), the value in *c_lob* is modified through the SQL UPDATE command, bypassing the *clob_selected* locator. The locator still sees the value of the LOB as of the point in time of the original SELECT. In other words, the locator does not see the update made via the SQL UPDATE command. This is illustrated by the subsequent dbms_lob.read call.
- The third operation (at t3) re-selects the LOB value into the locator *clob_selected*. The locator is thus updated with the latest snapshot environment which allows the locator to see the change made by the previous SQL UPDATE command. Therefore, in the next dbms_lob.read, an error is returned because the LOB value is empty (i.e., it does not contain any data).

Example of Repercussions of Mixing SQL DML with DMBS_LOB

```
INSERT INTO lob_table VALUES (1, NULL, 'abcd', NULL, NULL);
COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
```

```
  read_amount       INTEGER;
  read_offset       INTEGER;
  buffer            VARCHAR2(20);

BEGIN

  -- At time t1:
  SELECT c_lob INTO clob_selected
  FROM lob_table
  WHERE key_value = 1;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t2:
  UPDATE lob_table SET c_lob = empty_clob()
      WHERE key_value = 1;
  -- although the most current current LOB value is now empty,
  -- clob_selected still sees the LOB value as of the point
  -- in time of the SELECT

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
     buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  SELECT c_lob INTO clob_selected FROM lob_table WHERE
       key_value = 1;
  -- the SELECT allows clob_selected to see the most current
  -- LOB value

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  -- ERROR: ORA-01403: no data found
END;
/
```

WARNING: we advise that you avoid updating the same LOB with different locators. You will avoid many pitfalls if you use only one locator to update the same LOB value.

Using *lob_table as* defined above, two CLOBs are created as potential locators: clob_updated and clob_copied.

- At the time of the first SELECT INTO (at t1), the value in *c_lob* is associated with the locator *clob_updated.*
- The second operation (at t2) copies the value in *clob_updated* to *clob_copied.* At this juncture, both locators see the same value. The example demonstrates this with a series of dbms_lob.read calls.
- At this juncture (at t3), the program utilizes dbms_lob.write to alter the value in *clob_updated*, and a dbms_lob.read reveals a new value.
- However, a dbms_lob.read of the value through *clob_copied* (at t4) reveals that it still sees the value of the LOB as of the point in time of the assignment from *clob_updated* (at t2).
- It is not until *clob_updated* is assigned to *clob_copied* (t5) that *clob_copied* sees the modification made by *clob_updated.*

Example of an Updated LOB Locator

```
INSERT INTO lob_table
   VALUES (1, NULL, 'abcd', NULL, NULL);

COMMIT;

DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  clob_copied      CLOB;
  read_amount      INTEGER; ;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);
BEGIN

-- At time t1:
```

```
SELECT c_lob INTO clob_updated FROM lob_table
    WHERE key_value = 1
    FOR UPDATE;

-- At time t2:
clob_copied := clob_updated;
-- after the assign, clob_copied and clob_updated see the same
-- LOB value

read_amount := 10;
read_offset := 1;
dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
dbms_output.put_line('clob_updated value: ' || buffer);
-- Produces the output 'abcd'

read_amount := 10;
dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
dbms_output.put_line('clob_copied value: ' || buffer);
-- Produces the output 'abcd'


-- At time t3:
write_amount := 3;
write_offset := 5;
buffer := 'efg';
dbms_lob.write(clob_updated, write_amount, write_offset,
      buffer);

read_amount := 10;
dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
dbms_output.put_line('clob_updated value: ' || buffer);
-- Produces the output 'abcdefg'


-- At time t4:
read_amount := 10;
dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
dbms_output.put_line('clob_copied value: ' || buffer);
-- Produces the output 'abcd'


-- At time t5:
clob_copied := clob_updated;

read_amount := 10;
```

```
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcdefg'
END;
/
```

LOB bind variables

When a LOB locator is used as the source to update another internal LOB (as in a SQL INSERT or UPDATE statement, the DBMS_LOB.COPY routine, and so on), the snapshot environment in the source LOB locator determines the LOB value that is used as the source. If the source locator (for example L1) is a read consistent locator, then the LOB value as of the point in time of the SELECT of L1 is used. If the source locator (for example L2) is an updated locator, then the LOB value associated with L2's snapshot environment at the time of the operation is used.

Using *lob_table as* defined above, three CLOBs are created as potential locators: clob_selected, clob_updated and clob_copied.

- At the time of the first SELECT INTO (at t1), the value in *c_lob* is associated with the locator *clob_updated.*
- The second operation (at t2) copies the value in *clob_updated* to *clob_copied.* At this juncture, both locators see the same value.
- Then (at t3), the program utilizes dbms_lob.write to alter the value in *clob_updated,* and a dbms_lob.read reveals a new value.
- However, a dbms_lob.read of the value through *clob_copied* (at t4) reveals that *clob_copied* does not see the change made by *clob_updated.*
- Therefore (at t5), when *clob_copied* is used as the source for the value of the INSERT statement, we insert the value associated with *clob_copied* (i.e. without the new changes made by *clob_updated*). This is demonstrated by the subsequent dbms_lob.read of the value just inserted.

Example of Updating a LOB with a PL/SQL Variable

```
INSERT INTO lob_table
   VALUES (1, NULL, 'abcd', NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
```

```
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);
BEGIN

  -- At time t1:
  SELECT c_lob INTO clob_updated FROM lob_table
      WHERE key_value = 1
      FOR UPDATE;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'


  -- At time t2:
  clob_copied := clob_updated;


  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
       buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'
  -- note that clob_copied doesn't see the write made before
  -- clob_updated


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'
```

```
  -- At time t5:
  -- the insert uses clob_copied view of the LOB value which does
  -- not include clob_updated changes
  INSERT INTO lob_table values (2, NULL, clob_copied, NULL,
      NULL) RETURNING c_lob INTO clob_selected;

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset,
       buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

LOB locators cannot span transactions

Modifying an internal LOB's value through the LOB locator via DBMS_LOB, OCI, or SQL INSERT or UPDATE statements changes the locator from a read consistent locator to an updated locator. Further, the INSERT or UPDATE statement automatically starts a transaction and locks the row. Once this has occurred, the locator may *not* be used outside the current transaction. In other words, LOB locators cannot span transactions.

Using *lob_table as* defined above, a CLOB locator is created: clob_updated.

- At the time of the first SELECT INTO (at t1), the value in *c_lob* is associated with the locator *clob_updated*.
- The second operation (at t2), utilizes the dbms_lob.write command to alter the value in *clob_updated*, and a dbms_lob.read reveals a new value.
- The commit statement (at t3) ends the current transaction.
- Therefore (at t4), the subsequent dbms_lob.read operation fails because the *clob_updated* locator refers to a different (already committed) transaction. This is noted by the error returned. You must re-select the LOB locator before using it in further dbms_lob (and OCI) operations.

Example of Locator Not Spanning a Transaction

```
INSERT INTO lob_table
     VALUES (1, NULL, 'abcd', NULL, NULL);
COMMIT;

DECLARE
```

```
  num_var          INTEGER;
  clob_updated     CLOB;
  read_amount      INTEGER;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);

BEGIN

     -- At time t1:
     SELECT      c_lob
     INTO        clob_updated
     FROM        lob_table
     WHERE       key_value = 1
     FOR UPDATE;

     read_amount := 10;
     read_offset := 1;
     dbms_lob.read(clob_updated, read_amount, read_offset,
          buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcd'

     -- At time t2:
     write_amount := 3;
     write_offset := 5;
     buffer := 'efg';
     dbms_lob.write(clob_updated, write_amount, write_offset,
          buffer);

     read_amount := 10;
     dbms_lob.read(clob_updated, read_amount, read_offset,
         buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcdefg'


    -- At time t3:
    COMMIT;


    -- At time t4:
    read_amount := 10;
    dbms_lob.read(clob_updated , read_amount, read_offset,
```

```
         buffer);
    -- ERROR: ORA-22990: LOB locators cannot span transactions
END;
/
```

Examples of a Locator Not Spanning a Transaction

Assume the following tables:

```
CREATE TABLE tdrslob01 (
     a    CLOB,
     b    BLOB,
     c    NUMBER);

CREATE TABLE foo (
     key  NUMBER);

CONNECT to the database
EXECUTE "SELECT C,'I',A FROM tdrslob01 ORDER BY 1"
[ARRAY FETCH GET 16 rows INTO OCILobLocator ARRAY]
for (i=0; i<16; i++)
{
  EXECUTE "INSERT INTO foo VALUES(5)"
  OCITransCommit(...);
  FETCH TEXT in CLOB USING locators fetched
}
```

The sequence runs successfully because the SELECT of the locators occurs outside of a transaction. This means that the locators selected are not associated with a transaction. Even though the INSERT in the 'for' loop implicitly starts a transaction, the subsequent COMMIT in the 'for' loop ends the transaction. The FETCH of the LOB data via the locator returned from the SELECT outside a transaction succeeds. Both the SELECT of the locator and the FETCH of the locator data occur outside a transaction.

However, the addition of one statement produces an error:

```
CONNECT to the database
EXECUTE "INSERT INTO foo VALUES(5)"  <===
EXECUTE "SELECT C,'I',A FROM tdrslob01 ORDER BY 1"
[GET 16 rows]
for (i=0; i<16; i++)
{
  EXECUTE "INSERT INTO foo VALUES(5)"
  OCITransCommit(...);
```

```
  FETCH text in CLOB USING locators fetched <== get ORA-22990
}
```

In the second example, the SELECT of the locators occurs inside a transaction (the INSERT statement implicitly started a transaction). This means that the locators selected are associated with a transaction. The COMMIT in the 'for' loop commits the transaction in which the locators were selected. Therefore, the subsequent FETCH is trying to fetch locator values from the previous transaction which was already committed. Consequently, the 22990 error is returned.

Executing a COMMIT right after the first INSERT will succeed:

```
CONNECT to the database
EXECUTE "INSERT INTO foo VALUES(5)"  <===
OCITransCommit(...);  <===
EXECUTE "SELECT C,'I',A FROM tdrslob01 ORDER BY 1"
[get 16 rows]
for (i=0; i<16; i++)
{
  EXECUTE "INSERT INTO foo VALUES(5)"
  OCITransCommit(...);
  FETCH text in CLOB using locators fetched
}
```

In this example, the INSERT implicitly starts a transaction and the COMMIT ends the transaction. Therefore, the SELECT of the locators occurs outside of a transaction. This means that the locators selected are not associated with a transaction. Again, even though the INSERT in the 'for' loop implicitly starts a transaction, the subsequent COMMIT in the 'for' loop ends the transaction. Therefore, the FETCH of the LOB data via the locator returned from the SELECT which occurred outside a transaction succeeds. Both the SELECT of the locator and the FETCH of the locator data occur outside of a transaction.

Efficient Reads and Writes of Large Amounts of LOB Data

The most efficient way to read or write large amounts of LOB data is to use OCILobRead() or OCILobWrite() with the streaming mechanism enabled via polling or a callback.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about these APIs and a sample program of how to use them.

Reading LOB Values

When reading the LOB value, it is not an error to try to read beyond the end of the LOB. This means that you can always specify an input amount of 4 gigabytes regardless of the starting offset and the amount of data in the LOB. You do need to incur a round-trip to the server to call OCILobGetLength() to find out the length of the LOB value in order to determine the amount to read.

For example, assume that the length of a LOB is 5,000 bytes and you want to read the entire LOB value starting at offset 1,000. Also assume that you do not know the current length of the LOB value. Here's the OCI read call, excluding the initialization all parameters:

```
#define MAX_LOB_SIZE 4294967295
ub4  amount =  MAX_LOB_SIZE;
ub4  offset = 1000;
OCILobRead(svchp, errhp, locp, &amount, offset, bufp, bufl, 0, 0, 0, 0)
```

Writing LOB Values

As noted previously, the best way to populate the LOB with data, or write large amounts of data to the LOB, is to use the OCILobWrite() call with streaming. If you know how much data will be written to the LOB, specify that amount when calling OCILobWrite(). This will allow for the contiguity of the LOB data on disk. Apart from being spatially efficient, contiguous structure of the LOB data will make for faster reads and writes in subsequent operations.

Copying LOBs

Copying internal LOBs

The internal LOB types — BLOB, CLOB, and NCLOB — use *copy semantics*, as opposed to the *reference semantics* which apply to BFILEs. When a BLOB, CLOB, or NCLOB is copied from one row to another row in the same table or in a different table, the actual LOB value is copied, not just the LOB locator. For example, assuming lob_table1 and lob_table2 have schemas identical to lob_table described above, the statement

```
INSERT INTO lob_table1 (key_value, b_lob)
    (SELECT key_value, b_lob FROM lob_table2 T2
     WHERE T2.key_value = 101);
```

creates a new LOB locator in the table lob_table1, and copies the LOB data from lob_table2 to the location pointed to by a new LOB locator which is inserted into table lob_table1.

Copying external LOBs

BFILE types use *reference semantics* instead of copy semantics. This means that only the BFILE locator is copied from one row to another row. Put another way: it is not possible to make a copy of an external LOB value without issuing an operating system command to copy the operating system file.

Deleting LOBs

Deleting Internal LOBs

You delete a row that contains an internal LOB column / attribute by (a) using the explicit SQL DML command DELETE, or (b) using a SQL DDL command that effectively deletes it, such as DROP TABLE, TRUNCATE TABLE, or DROP TABLESPACE. In either case you delete the LOB locator *and the LOB value as well.*

But note that due to the consistent read mechanism, the old LOB value remains accessible with the value that it had at the time of execution of the statement (such as SELECT) that returned the LOB locator.

See Also: "Read consistent locators" on page 6-24.

Of course, two distinct rows of a table with a LOB column have their own distinct LOB locators and distinct copies of the LOB values irrespective of whether the LOB values are the same or different. This means that deleting one row has no effect on the data or LOB locator in another row even if one LOB was originally copied from another row.

Deleting External LOBs

The LOB value in a BFILE, however, does not get deleted by using SQL DDL or SQL DML commands. Only the BFILE locator is deleted. Deletion of a record containing a BFILE column amounts to de-linking that record from an existing file, *not* deleting the physical operating system file itself. An SQL DELETE statement on a particular row deletes the BFILE locator for the particular row, thereby removing the reference to the operating system file.

The following DELETE, DROP TABLE, or TRUNCATE TABLE statements delete the row, and hence the BFILE locator that refers to `image1.gif`, but leave the operating system file undeleted in the filesystem.

```
DELETE FROM lob_table
     WHERE key_value = 21;
```

```
DROP TABLE lob_table;

TRUNCATE TABLE lob_table;
```

Copying Data from LONGs to LOBs

One of the problems you may face is how to convert data from the LONG datatype into LOB format. The `loadlob.sql` PL/SQL program demonstrates how to convert a LONG to a LOB by using the DBMS_LOB.LOADFROMFILE method. The program requires that you perform a sequence of steps:

1. Leave the LONG column in the old table and add a new LOB column using the ALTER TABLE command.

   Note: The ALTER TABLE command is not able to change the type of a LONG column to a LOB column. LONG and LOB columns are two distinct datatypes so it is not possible to assign a LONG column to a LOB column.

2. Write the data in the LONG or LONG RAW to a flat file.
3. Use CREATE DIRECTORY to point to the directory where the BFILE (flat file) was written.
4. Using either OCI or PL/SQL, there are three different ways you can copy the data from the server-side flat file into the LOB:
   a. The OCI command `OCILobLoadFromFile` or the PL/SQL command `DBMS_LOB.LOADFROMFILE`(): This is the fastest ways to copy from a server-side operating system flat file to a LOB.
   b. The OCI command `OCILobWrite`() from a server-side external procedure: The flat file will be on the server-side even if the program which calls the server-side external procedure is run from the client. This is the second fastest way to transfer from a server side operating system flat file to a LOB.
   c. The OCI command `OCILobWrite`(): This method is used in the `bull_lob` program listed below. This may not be the fastest way to convert a LONG to a LOB, but it may be the only alternative in the circumstances. In such cases, when the program is run on a remote client machine, the LONG data on the server must be written to a client machine flat file, and then the client flat file written back to the server LOB column. Since

this will involve two trips across the network, the load on performance must be considered if this will be an operation that is frequently repeated.

Note: The user will need to do their own character set conversions for CLOBS and NCLOBS because the flat file or BFILE will store the data as binary or raw data.

WARNING: The export/import utility is currently not capable of converting from LONGS TO LOBs.

Example

The example that follows shows the PL/SQL version of method 4(a) listed above for loading a LONG which has been written to a flat file named /tmp/sound_clip into a LOB column.

Note: There is a separate bulletin which addresses how to convert a LONG to a LOB using the OCILobWrite command.

Complete the following steps to execute the loadlob.sql PL/SQL script:

1. Create the file sound_clip with the following contents and copy it to the /tmp directory:

   ```
   sound_clip: abcdefghijklmnopqrstuvwxyz
   ```

2. Run the following SQL script:

```
% sqlplus scott/tiger @loadlob
loadlob.sql
-----------
set echo on;
connect sys/change_on_install;
grant all on dbms_lob to scott;
grant create any directory to scott;
connect scott/tiger;
drop directory some_dir_alias;
```

```
create directory some_dir_alias as '/tmp';
drop table multimedia;

/* Create the table */

CREATE TABLE multimedia
(
   id           NUMBER,
   video_clip   CLOB DEFAULT empty_clob(),
   audio_clip   CLOB DEFAULT NULL,
   some_file    BFILE DEFAULT NULL
) ;

/* Load data into the table */
/* Insert 10 rows into the table which defaults to initializing */
/* the video_clip to empty and the audio_clip and some_file to null. */

/* The fastest way to do this is to use array inserts with OCI */
/* (see OCIBindArrayOfStruct) */
/* The less speedy method is to use a loop in PL/SQL as follows. */

declare
   loop_count integer;
begin
   loop_count := 1;
   while loop_count <= 10 loop
     insert into multimedia (id) values (loop_count);
     loop_count := loop_count + 1;
   end loop;
end;
/

/* Initialize the first audio clip to the actual value. */
/* Then copy this value to all rows in the table. */

declare
  ac      clob;
  amount  integer;
  a_file  bfile := BFILENAME('SOME_DIR_ALIAS', 'sound_clip');
begin
  update multimedia set audio_clip = empty_clob() where id = 1 returning
        audio_clip into ac;

/* Open the server side file that contains the audio clip, load it into */
/* the CLOB and then close the file.  Assume that the audio clip is */
```

```
/* only 32,000 bytes long and that it starts at position 1 in the file. */

  dbms_lob.fileopen(a_file, dbms_lob.file_readonly);
  amount := 26;

/* Note that the destination and source offsets default to 1 */

  dbms_lob.loadfromfile(ac, a_file, amount);
  dbms_lob.fileclose(a_file);
  commit;

/* Update all rows in the table to the audio clip you just loaded. */
  update multimedia set audio_clip =
        (select audio_clip from multimedia where id = 1)
        where audio_clip is null;
end;
/

select id, audio_clip from multimedia;
```

3. The output should resemble:

```
SQL> @loadlob
SQL> set echo on;
SQL> connect sys/change_on_install;
Connected.
SQL> GRANT ALL on dbms_lob to scott;
Grant succeeded.
SQL> GRANT CREATE ANY DIRECTORY to scott;
Grant succeeded.
SQL> CONNECT scott/tiger;
Connected.
SQL> DROP DIRECTORY some_dir_alias;
Directory dropped.
SQL> CREATE DIRECTORY some_dir_alias as '/tmp';
Directory created.
SQL> DROP TABLE multimedia;
Table dropped.
SQL>
SQL> /* CREATE THE TABLE */
SQL>
SQL> create table multimedia
  2  (
  3  id     number,
  4  video_clip   clob default empty_clob(),
```

```
  5  audio_clip   clob default null,
  6  some_file    bfile default null
  7  ) ;

Table created.

SQL>
SQL>
SQL> /* LOAD DATA INTO THE TABLE */
SQL> /* Insert 10 rows into the table which defaults to initializing */
DOC> /* the video_clip to empty and the audio_clip and some_file to null.*/
DOC> */
SQL>
SQL> /* The fast way to do this is to use array inserts with OCI */
DOC> /* (see OCIBindArrayOfStruct) */
DOC> /* The not so fast way is to use a loop in plsql as follows. */

SQL>
SQL> declare
  2  loop_count integer;
  3  begin
  4  loop_count := 1;
  5  while loop_count <= 10 loop
  6    insert into multimedia (id) values (loop_count);
  7    loop_count := loop_count + 1;
  8  end loop;
  9  end;
 10  /

PL/SQL procedure successfully completed.

SQL> /* Initialize the first audio clip to the actual value. */
DOC> /* Then copy this value to all rows in the table. */

SQL> DECLARE
  2    ac      CLOB;
  3    amount  INTEGER;
  4    a_file  BFILE := BFILENAME('SOME_DIR_ALIAS', 'sound_clip');
  5  BEGIN
  6    UPDATE multimedia SET audio_clip = empty_clob() WHERE id = 1
returning
  7      audio_clip into ac;
  8
  8  /* Open the server side file that contains the audio clip, load it  */
  9  /* into the clob and then close the file.  Note, assume that the */
```

```
 10  /* audio clip is only 32,000 bytes long and that it starts at */
 11  /* position 1 in the file.*/
 12    dbms_lob.fileopen(a_file, dbms_lob.file_readonly);
 13    amount := 26;
 14    /* note that the destination and source offsets default to 1 */
 15    dbms_lob.loadfromfile(ac, a_file, amount);
 16    dbms_lob.fileclose(a_file);
 17    COMMIT;
 18
 18  /* Update all rows in the table to the audio clip we just loaded. */
 19    UPDATE multimedia SET audio_clip =
 20       (SELECT audio_clip FROM multimedia WHERE id = 1)
 21       WHERE audio_clip is null;
 22  end;
 23  /

PL/SQL procedure successfully completed.

SQL>
SQL> select id, audio_clip from multimedia;

        ID
----------
AUDIO_CLIP
--------------------------------------------------------------------------------
          1
abcdefghijklmnopqrstuvwxyz
         2
abcdefghijklmnopqrstuvwxyz
         3
abcdefghijklmnopqrstuvwxyz

        ID
----------
AUDIO_CLIP
--------------------------------------------------------------------------------
         4
abcdefghijklmnopqrstuvwxyz
         5
abcdefghijklmnopqrstuvwxyz
         6
abcdefghijklmnopqrstuvwxyz

        ID
----------
```

```
AUDIO_CLIP
--------------------------------------------------------------------------------
         7
abcdefghijklmnopqrstuvwxyz
         8
abcdefghijklmnopqrstuvwxyz
         9
abcdefghijklmnopqrstuvwxyz

        ID
-----------
AUDIO_CLIP
--------------------------------------------------------------------------------
        10
abcdefghijklmnopqrstuvwxyz

10 rows selected.

SQL>
SQL> quit
```

LOBs in the Object Cache

When you create an object in the object cache that contains an internal LOB attribute, the LOB attribute is implicitly set to empty. You may not use this empty LOB locator to write data to the LOB. You must first *flush* the object, thereby inserting a row into the table and creating an empty LOB — that is, a LOB with 0 length. Once the object is refreshed in the object cache (use OCI_PIN_LATEST), the real LOB locator is read into the attribute, and you can then call the OCI LOB API to write data to the LOB.

When creating an object with a BFILE attribute, the BFILE is set to NULL. It must be updated with a valid directory alias and filename before reading from the file.

When you copy one object to another in the object cache with a LOB locator attribute, only the LOB locator is copied. This means that the LOB attribute in these two different objects contain exactly the same locator which refers to *one and the same* LOB *value*. Only when the target object is flushed is a separate, physical copy of the LOB value made, which is distinct from the source LOB value.

> **See Also:** "Example of a Read Consistent Locator" on page 6-25 for a description of what version of the LOB value will be seen by each object if a write is performed through one of the locators.

Therefore, in cases where you want to modify the LOB that was the target of the copy, *you must flush the target object, refresh the target object, and then* write to the LOB through the locator attribute.

LOB Buffering Subsystem

LOB Buffering

Oracle8 provides a LOB buffering subsystem (LBS) for advanced OCI based applications such as DataCartridges, Web servers, and other client-based applications that need to buffer the contents of one or more LOBs in the client's address space. The client-side memory requirement for the buffering subsystem during its maximum usage is 512K bytes. It is also the maximum amount that you can specify for a single read or write operation on a LOB that has been enabled for buffered access.

Advantages of LOB Buffering

The advantages of buffering, especially for applications that perform a series of small reads and writes (often repeatedly) to specific regions of the LOB, are two fold:

- Buffering enables deferred writes to the server. You can buffer up several writes in the LOB's buffer in the client's address space and eventually *flush* the buffer to the server. This reduces the number of network roundtrips from your client application to the server, and hence, makes for better overall performance for LOB updates.
- Buffering reduces the overall number of LOB updates on the server, thereby reducing the number of LOB versions and amount of logging. This results in better overall LOB performance and disk space usage.

Considerations in the Use of LOB Buffering

The following caveats hold for buffered LOB operations:

- Oracle8 provides a simple buffering subsystem, and *not* a cache. To be specific, Oracle8 does not guarantee that the contents of a LOB's buffer are always in synch with the LOB value in the server. Unless you *explicitly flush* the contents of a LOB's buffer, you will not see the results of your buffered writes reflected in the actual LOB on the server.
- Error recovery for buffered LOB operations is your responsibility. Owing to the deferred nature of the actual LOB update, error reporting for a particular buffered read or write operation is deferred until the next access to the server based LOB.

- Transactions involving buffered LOB operations cannot migrate across user sessions — LBS is a single user, single threaded system.
- Oracle8 does not guarantee transactional support for buffered LOB operations. To ensure transactional semantics for buffered LOB updates, you must maintain logical savepoints in your application to rollback all the changes made to the buffered LOB in the event of an error. You should always wrap your buffered LOB updates within a logical savepoint.
- In any given transaction, once you have begun updating a LOB using buffered writes, it is your responsibility to ensure that the same LOB is not updated through any other operation within the scope of the same transaction *that bypasses the buffering subsystem*.

  You could potentially do this by using an SQL statement to update the server-based LOB. Oracle8 cannot distinguish, and hence prevent, such an operation. This will seriously affect the correctness and integrity of your application.
- Buffered operations on a LOB are done through its locator, just as in the conventional case. A locator that is enabled for buffering will provide a consistent read version of the LOB, until you perform a write operation on the LOB through that locator.

  See Also: "Read consistent locators" on page 6-24.

  Once the locator becomes an updated locator by virtue of its being used for a buffered write, it will always provide access to the most up-to-date version of the LOB *as seen through the buffering subsystem*. Buffering also imposes an additional significance to this updated locator — all further buffered writes to the LOB can be done *only through this updated locator*. Oracle8 will return an error if you attempt to write to the LOB through other locators enabled for buffering.

  See Also: "Updated locators" on page 6-27.
- You can pass an updated locator that was enabled for buffering as an IN parameter to a PL/SQL procedure. However, passing an IN OUT or an OUT parameter will produce an error, as will an attempt to return an updated locator.
- You cannot assign an updated locator that was enabled for buffering to another locator. There are a number of different ways that assignment of locators may occur — through OCILobAssign(), through assignment of PL/SQL variables, through OCIObjectCopy() where the object contains the LOB attribute, and so on. Assigning a consistent read locator that was enabled for buffering to a locator that did not have buffering enabled, turns buffering on for the target locator. By the same token, assigning a locator that was not enabled for buffering to

a locator that did have buffering enabled, turns buffering off for the target locator.

Similarly, if you SELECT into a locator for which buffering was originally enabled, the locator becomes overwritten with the new locator value, thereby turning buffering off.

- Appending to the LOB value using buffered write(s) is allowed, but only if the starting offset of these write(s) is exactly one byte (or character) past the end of the BLOB (or CLOB / NCLOB). In other words, the buffering subsystem does not support appends that involve creation of zero-byte fillers or spaces in the server based LOB.
- For CLOBs, Oracle8 requires that the character set form for the locator bind variable on the client side be the same as that of the LOB in the server. This is usually the case in most OCI LOB programs. The exception is when the locator is SELECTed from a *remote* database, which may have a different character set form from the database which is currently being accessed by the OCI program. In such a case, an error is returned. If there is no character set form input by the user, then we assume it is SQLCS_IMPLICIT.

LOB Buffering Operations

The Physical Structure of the LOB Buffer For Oracle 8.0, each user *session* has a fixed page pool of 16 pages, which are to be shared by all LOBs accessed in buffering mode from that session. Each *page* has a fixed size of up to 32K *bytes* (not characters). A LOB's buffer consists of one or more of these pages, up to a maximum of 16 per session. The maximum amount that you ought to specify for any given buffered read or write operation is 512K bytes, remembering that under different circumstances the maximum amount you may read / write could be smaller.

Consider that a LOB is divided into fixed-size, logical regions. Each page is mapped to one of these fixed size regions, and is in essence, their in-memory copy. Depending on the input offset and amount specified for a read or write operation, Oracle8 allocates one or more of the free pages in the page pool to the LOB's buffer. A *free page* is one that has not been read or written by a buffered read or write operation.

Using the LOB Buffering System For example, assuming a page size of 32K, for an input offset of 1000 and a specified read / write amount of 30000, Oracle8 reads the first 32K byte region of the LOB into a page in the LOB's buffer. For an input offset of 33000 and a read / write amount of 30000, the second 32K region of the LOB is read into a page. For an input offset of 1000, and a read / write amount of 35000, the

LOB's buffer will contain *two* pages — the first mapped to the region 1 — 32K, and the second to the region 32K+1 — 64K of the LOB.

This mapping between a page and the LOB region is temporary until Oracle8 maps another region to the page. When you attempt to access a region of the LOB that is not already available in full in the LOB's buffer, Oracle8 allocates any available free page(s) from the page pool to the LOB's buffer. If there are no free pages available in the page pool, Oracle8 reallocates the pages as follows. It ages out the *least recently used* page among the *unmodified* pages in the LOB's buffer and reallocates it for the current operation.

If no such page is available in the LOB's buffer, it ages out the least recently used page among the *unmodified* pages of *other* buffered LOBs in the same session. Again, if no such page is available, then it implies that all the pages in the page pool are *dirty* (i.e. they have been modified), and either the currently accessed LOB, or one of the other LOBs, need to be flushed. Oracle8 notifies this condition to the user as an error. Oracle8 *never* flushes and reallocates a dirty page implicitly — you can either flush them explicitly, or discard them by disabling buffering on the LOB.

To illustrate the above discussion, consider two LOBs being accessed in buffered mode — L1 and L2, each with buffers of size 8 pages. Assume that 6 of the 8 pages in L1's buffer are dirty, with the remaining 2 contain unmodified data read in from the server. Assume similar conditions in L2's buffer. Now, for the next buffered operation on L1, Oracle8 will reallocate the least recently used page from the two unmodified pages in *L1's buffer*. Once all the 8 pages in L1's buffer are used up for LOB writes, Oracle8 can service two more operations on L1 by allocating the two unmodified pages from *L2's buffer* using the least recently used policy. But for any further buffered operations on L1 or L2, Oracle8 returns an error.

If all the buffers are dirty and you attempt another read from or write to a buffered LOB, you will raise the following error:

```
Error 22280: no more buffers available for operation
```

There are two possible causes:

1. All buffers in the buffer pool have been used up by previous operations.

   In this case, flush the LOB(s) through the locator that is being used to update the LOB.

2. You are trying to flush a LOB without any previous buffered update operations.

   In this case, write to the LOB through a locator enabled for buffering before attempting to flush buffers.

Flushing the LOB Buffer The term *flush* refers to a set of processes. Writing data to the LOB in the buffer through the locator transforms the locator into an *updated locator*. Once you have updated the LOB data in the buffer through the updated locator, a flush call will

- write the dirty pages in the LOB's buffer to the server-based LOB, thereby updating the LOB value,
- reset the updated locator to be a read consistent locator, and
- either free the flushed buffers or turn the status of the buffer pages back from dirty to unmodified.

After the flush, the locator becomes a read consistent locator and can be assigned to another locator (L2 := L1).

For instance, suppose you have two locators, L1 and L2. Let us say that they are both *read consistent locators* and consistent with the state of the LOB data in the server. If you then update the LOB by writing to the buffer, L1 becomes an updated locator. L1 and L2 now refer to different versions of the LOB value. If you wish to update the LOB in the server, you must use L1 to retain the read consistent state captured in L2. The flush operation writes a new snapshot environment into the locator used for the flush. The important point to remember is that you must use the updated locator (L1), when you flush the LOB buffer. Trying to flush a read consistent locator will generate an error.

This raises the question: What happens to the data in the LOB buffer? There are two possibilities. In the default mode, the flush operation retains the data in the pages that were modified. In this case, when you read or write to the same range of bytes no roundtrip to the server is necessary. Note that *flush* in this context does not clear the data in the buffer. It also does not return the memory occupied by the flushed buffer to the client address space.

Note: Unmodified pages may now be aged out if necessary.

In the second case, you set the flag parameter in `OCILobFlushBuffer()` to `OCI_LOB_BUFFER_FREE` to free the buffer pages, and so return the memory to the client address space. Note that *flush* in this context updates the LOB value on the server, returns a read consistent locator, and frees the buffer pages.

Flushing the Updated LOB It is very important to note that you must flush a LOB that has been updated through the LBS:

- before committing the transaction,
- before migrating from the current transaction to another,
- before disabling buffering operations on a LOB
- before returning from an external callout execution into the calling function / procedure / method in PL / SQL.

Note: When the external callout is called from a PL / SQL block and the locator is passed as a parameter, all buffering operations, including the enable call, should be made within the callout itself. In other words, we recommend that you adhere to the following sequence:

- call the external callout,
- enable the locator for buffering,
- read / write using the locator,
- flush the LOB,
- disable the locator for buffering, and
- return to the calling function / procedure / method in PL / SQL.

Remember that Oracle8 never implicitly flushes the LOB.

Using Locators Enabled for Buffering Note that there are several cases in which you can use buffer-enabled locators and others in which you cannot.

- A locator that is enabled for buffering can only be used with the following OCI APIs:

  `OCILobRead()`, `OCILobWrite()`, `OCILobAssign()`, `OCILobIsEqual()`, `OCILobLocatorIsInit()`, `OCILobLocatorSize()`, `OCILobCharSetId()`, `OCILobCharSetForm()`.

- The following OCI APIs will return errors if used with a locator enabled for buffering:

  `OCILobCopy()`, `OCILobAppend()`, `OCILobErase()`, `OCILobGetLength()`, `OCILobTrim()`.

These APIs will also return errors when used with a locator which has not been enabled for buffering, but the LOB that the locator represents is already being accessed in buffered mode through some other locator.

- An error is returned from DBMS_LOB APIs if the input lob locator has buffering enabled.
- As in the case of all other locators, locators enabled for LOB buffering cannot span transactions.

Saving Locator State so as to Avoid a Reselect Suppose you want to save the current state of the LOB before further writing to the LOB buffer. In performing updates while using LOB buffering, writing to an existing buffer does not make a roundtrip to the server, and so does not refresh the snapshot environment in the locator. This would not be the case if you were updating the LOB directly without using LOB buffering. In that case, every update would involve a roundtrip to the server, and so would refresh the snapshot in the locator. In order to save the state of a LOB that has been written through the LOB buffer, you therefore need to

1. Flush the LOB, thereby updating the LOB and the snapshot environment in the locator (L1). At this point, the state of the locator (L1) and the LOB are the same.
2. Assign the locator (L1) used for flushing and updating to another locator (L2). At this point, the states of the two locators (L1 and L2), as well as the LOB are all identical.

L2 now becomes a read consistent locator with which you are able to access the changes made through L1 up until the time of the flush, but not after! This assignment avoids incurring a roundtrip to the server to reselect the locator into L2.

Example of LOB Buffering

The following *pseudocode* for an OCI program based on the `lob_table` schema briefly explains the concepts listed above.

```
OCI_BLOB_buffering_program()
{
   int            amount;
   int            offset;
   OCILobLocator  lbs_loc1, lbs_loc2, lbs_loc3;
   void          *buffer;
   int            bufl;

   -- Standard OCI initialization operations - logging on to
   -- server, creating and initializing bind variables etc.

   init_OCI();

   -- Establish a savepoint before start of LBS operations
   exec_statement("savepoint lbs_savepoint");

   -- Initialize bind variable to BLOB columns from buffered
   -- access:
   exec_statement("select b_lob into lbs_loc1 from lob_table
       where key_value = 12");
   exec_statement("select b_lob into lbs_loc2 from lob_table
       where key_value = 12 for update");
   exec_statement("select b_lob into lbs_loc2 from lob_table
       where key_value = 12 for update");

   -- Enable locators for buffered mode access to LOB:
   OCILobEnableBuffering(lbs_loc1);
   OCILobEnableBuffering(lbs_loc2);
   OCILobEnableBuffering(lbs_loc3);

   -- Read 4K bytes through lbs_loc1 starting from offset 1:
   amount = 4096; offset = 1; bufl = 4096;
   OCILobFileRead(.., lbs_loc1, offset, &amount, buffer, bufl,
      ..);
      if (exception)
          goto exception_handler;
          -- This will read the first 32K bytes of the LOB from
          -- the server into a page (call it page_A) in the LOB's
          -- client-side buffer.
          -- lbs_loc1 is a read consistent locator.
```

```
    -- Write 4K of the LOB throgh lbs_loc2 starting from
    -- offset 1:
    amount = 4096; offset = 1; bufl = 4096;
    buffer = populate_buffer(4096);
    OCILobFileWrite(.., lbs_loc2, offset, amount, buffer,
        bufl, ..);

if (exception)
    goto exception_handler;
    -- This will read the first 32K bytes of the LOB from
    -- the server into a new page (call it page_B) in the
    -- LOB's buffer, and modify the contents of this page
    -- with input buffer contents.
    -- lbs_loc2 is an updated locator.

    -- Read 20K bytes through lbs_loc1 starting from
    -- offset 10K
    amount = 20480; offset = 10240;
    OCILobFileRead(.., lbs_loc1, offset, &amount, buffer,
        bufl, ..);

if (exception)
  goto exception_handler;
    -- Read directly from page_A into the user buffer.
    -- There is no round-trip to the server because the
    -- data is already in the client-side buffer.

    -- Write 20K bytes through lbs_loc2 starting from offset
    -- 10K
    amount = 20480; offset = 10240; bufl = 20480;
    buffer = populate_buffer(20480);
    OCILobFileWrite(.., lbs_loc2, offset, amount, buffer,
         bufl, ..);

if (exception)
    goto exception_handler;
    -- The contents of the user buffer will now be written
    -- into page_B without involving a round-trip to the
    -- server.  This avoids making a new LOB version on the
    -- server and writing redo to the log.

    -- The following write through lbs_loc3 will also
    -- result in an error:
    amount = 20000; offset = 1000; bufl = 20000;
    buffer = populate_buffer(20000);
```

```
        OCILobFileWrite(.., lbs_loc3, offset, amount, buffer,
             bufl, ..);

    if (exception)
        goto exception_handler;
        -- No two locators can be used to update a buffered LOB
        -- through the buffering subsystem

    -- The following update through lbs_loc3 will also
    -- result in an error
    OCILobFileCopy(.., lbs_loc3, lbs_loc2, ..);

    if (exception)
        goto exception_handler;
        -- Locators enabled for buffering cannot be used with
        -- operations like Append, Copy, Trim etc.

    -- When done, flush LOB's buffer to the server:
    OCILobFlushBuffer(.., lbs_loc2, OCI_LOB_BUFFER_NOFREE);

    if (exception)
       goto exception_handler;
       -- This flushes all the modified pages in the LOB's buffer,
       -- and resets lbs_loc2 from updated to read consistent
       -- locator. The modified pages remain in the buffer
       -- without freeing memory.  These pages can be aged
       -- out if necessary.

    -- Disable locators for buffered mode access to LOB */
    OCILobDisableBuffering(lbs_loc1);
    OCILobDisableBuffering(lbs_loc2);
    OCILobDisableBuffering(lbs_loc3);

    if (exception)
       goto exception_handler;
       -- This disables the three locators for buffered access,
       -- and frees up the LOB's buffer resources.

    exception_handler:
    handle_exception_reporting();
    exec_statement("rollback to savepoint lbs_savepoint");
}
```

User Guidelines for Best Performance Practices

- Since LOBs are big, you can obtain the best performance by reading and writing large chunks of a LOB value at a time. This helps in several respects:
  - **a.** If accessing the LOB from the client side and the client is at a different node than the server, large reads / writes reduce network overhead.
  - **b.** If using the 'NOCACHE' option, each small read / write incurs an I / O. Reading / writing large quantities of data reduces the I / O.
  - **c.** Writing to the LOB creates a new version of the LOB CHUNK. Therefore, writing small amounts at a time will incur the cost of a new version for each small write. If logging is on, the CHUNK is also stored in the redo log.
- If you need to read / write small pieces of LOB data on the client, use LOB buffering — see OCILobEnableBuffering(), OCILobDisableBuffering(), OCILobFlushBuffer(), OCILobWrite(), OCILobRead(). Basically, turn on LOB buffering before reading / writing small pieces of LOB data.

  See Also: "LOB Buffering Subsystem" on page 6-47 for more information on LOB buffering.

- Use OCILobWrite() and OCILobRead() with a callback so data is streamed to / from the LOB. Make sure that the length of the entire write is set in the 'amount' parameter on input.
- Use a checkout / checkin model for LOBs. LOBs are optimized for the following:
  - **a.** SQL UPDATE which replaces the entire LOB value
  - **b.** Copy the entire LOB data to the client, modify the LOB data on the client side, copy the entire LOB data back to the database. This can be done using OCILobRead() and OCILobWrite() with streaming.

Working with Varying-Width Character Data

Varying width character data is not supported for BLOBs, CLOBs and NCLOBs. However, BLOBs can contain any data. Since CLOBs / NCLOBs cannot store varying width character sets, you may be tempted to store varying width characters in a BLOB and do the character set conversion yourself. The drawback is that you need to do these conversions, and also that the offset and amount parameters are in terms of bytes instead of characters. So, the danger is that you could retrieve text information from the BLOB but cut a varying width character in half because the byte amount you specified was not correct. Consequently, we caution against taking this course of action.

BFILEs likewise can contain any data including text. But, once again, in storing the text, you will need to do your own character set conversions and offset and amount parameters will be in bytes.

As stated above, CLOBs store fixed width single byte data, and NCLOBs store fixed width multi byte data. Neither supports varying width data.

You might expect from this that if the database character set is varying width, and a user tries to create a table with a CLOB column, the create will fail. This is almost the case, but the reality is a little different.

- If a user other than the system user tries to create a table with a CLOB column, the create will fail.
- If the system user tries to create a table with a CLOB column, the create will succeed. However, subsequent inserts into the table will fail if the CLOB column has a value other than NULL.

The same holds true for NCLOBs and the database national character set.

The reason for allowing the SQL DDL to pass while making sure that the SQL DML fails if the user tries to insert a non-null value into the LOB that has a varying width character set is so that the same table can be created and exist in several different databases regardless of the underlying CHAR (NCHAR) character set. The user can write one application and modify it slightly for databases where the CHAR (NCHAR) character set is varying width such that the insert sets the varying width LOB to NULL.

LOB Reference

Reference Overview

Although not explicitly marked, this section is organized on the following basis.

- SQL DML functions EMPTY_BLOB(), EMPTY_CLOB() and BFILENAME() which are used for initialization (immediately following this overview).
- Summary of means provided by the OCI for manipulating LOBs (beginning with "Using the OCI to Manipulate LOBs" on page 6-63).
- The DBMS_LOB package, listing all functions and procedures (beginning with "DBMS_LOB Package" on page 6-66). This section contains the main body of technical specifications that underlie LOBs.
- A brief list of constraints that apply to using LOBs at the time of the first production release of Oracle8.

EMPTY_BLOB() and EMPTY_CLOB() Functions

You can use the special functions EMPTY_BLOB () and EMPTY_CLOB () in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NULL internal LOB to empty. These are available as special functions in Oracle8 SQL DML, and are not part of the DBMS_LOB package.

Before you can start writing data to an internal LOB using OCI or the DBMS_LOB package, the LOB column must be made non-null, that is, it must contain a locator that points to an empty or populated LOB value. You can initialize a BLOB column's value to empty by using the function EMPTY_BLOB() in the VALUES clause of an INSERT statement. Similarly, a CLOB or NCLOB column's value can be initialized by using the function EMPTY_CLOB().

Syntax

```
FUNCTION EMPTY_BLOB( ) RETURN BLOB;
FUNCTION EMPTY_CLOB( ) RETURN CLOB;
```

Parameters

None.

Return Values

EMPTY_BLOB() returns an empty locator of type BLOB and EMPTY_CLOB() returns an empty locator of type CLOB, which can also be used for NCLOBs.

Pragmas

None.

Exceptions

An exception is raised if you use these functions anywhere but in the VALUES clause of a SQL INSERT statement or as the source of the SET clause in a SQL UPDATE statement.

Examples

The following example shows EMPTY_BLOB() usage with SQL DML:

```
INSERT INTO lob_table VALUES (1001, EMPTY_BLOB(), 'abcde', NULL, NULL);
UPDATE lob_table SET c_lob = EMPTY_CLOB() WHERE key_value = 1001;
INSERT INTO lob_table VALUES (1002, NULL, NULL, NULL, NULL);
```

The following example shows the correct and erroneous usage of EMPTY_BLOB() and EMPTY_CLOB () in PL/SQL programs:

```
DECLARE
  loba         BLOB;
  lobb         CLOB;
  read_offset  INTEGER;
  read_amount  INTEGER;
  rawbuf       RAW(20);
  charbuf      VARCHAR2(20);
BEGIN
  loba := EMPTY_BLOB();
  read_amount := 10; read_offset := 1;
  -- the following read will fail
  dbms_lob.read(loba, read_amount, read_offset, rawbuf);

  -- the following read will succeed;
  UPDATE lob_table SET c_lob = EMPTY_CLOB() WHERE key_value =
         1002 RETURNING c_lob INTO lobb;
dbms_lob.read(lobb, read_amount, read_offset, charbuf);
  dbms_output.put_line('lobb value: ' || charbuf);
```

BFILENAME() Function

The BFILENAME() function should be called as part of SQL INSERT to initialize a BFILE column or attribute for a particular row by associating it with a physical file in the server's filesystem.

The DIRECTORY object represented by the directory_alias parameter to this function must *already be defined* using SQL DDL before this function is called in SQL DML or a PL/SQL program. You can call the CREATE DIRECTORY() command after BFILENAME(). However, the target object must exist by the time you actually use the BFILE locator (for example, as having been used as a parameter to an operation such as OCILobFileOpen() or DBMS_LOB.FILEOPEN()).

Note that BFILENAME() does not validate privileges on this DIRECTORY object, or check if the physical directory that the DIRECTORY object represents actually exists. These checks are performed only during file access using the BFILE locator that was initialized by the BFILENAME() function.

You can use BFILENAME() as part of a SQL INSERT and UPDATE statement to initialize a BFILE column. You can also use it to initialize a BFILE locator variable in a PL/SQL program, and use that locator for file operations. However, if the corresponding directory alias and/or filename does not exist, then PL/SQL DBMS_LOB routines that use this variable will generate errors.

The 'directory_alias' parameter in the BFILENAME() function must be specified taking case-sensitivity of the directory name into consideration. This is described in the examples.

See Also: "DIRECTORY Name Specification" on page 6-17.

Syntax

```
FUNCTION BFILENAME(directory_alias IN VARCHAR2,
                   filename IN VARCHAR2)
RETURN BFILE;
```

See Also: "DIRECTORY Name Specification" on page 6-17 for information about the use of uppercase letters in the directory name, and OCILobFileSetName() in *Oracle Call Interface Programmer's Guide* for an equivalent OCI based routine.

Parameters

Table 6–1 FILENAME Parameters

Parameter Name	Meaning
directory_alias	The name of the DIRECTORY object that was created using the CREATE DIRECTORY command.
filename	The name of the operating system file on the server.

Return Values

BFILE locator upon success.

NULL if directory_alias has not been defined previously.

Pragmas

None.

Exceptions

None.

Example

To access a file 'scott.dat' located in SCOTT_DIR, and file 'mary.dat' located in Mary_Dir, the BFILE locators must be initialized as shown below.

```
DECLARE
    fil_1, fil_2 BFILE;
  result INTEGER;
BEGIN
    fil_1 := BFILENAME('SCOTT_DIR', 'scott.dat');
    fil_2 := BFILENAME('Mary_Dir', 'mary.dat');
    DBMS_LOB.FILEOPEN(fil_1);
    DBMS_LOB.FILEOPEN(fil_2);
    result := DBMS_LOB.COMPARE(fil_1, fil_2);
    IF (result != 0)
    THEN
      DBMS_OUTPUT.PUT_LINE('The two files are different');
    END IF;
    DBMS_LOB.FILECLOSE(fil_1);
    DBMS_LOB.FILECLOSE(fil_2);
    -- FILEOPEN will fail with the following initialization (in
       lowercase)
    fil_1 := BFILENAME('scott_dir', 'scott.dat');
    DBMS_LOB.FILEOPEN(fil_1);

-- this is an error
END;

INSERT INTO lob_table VALUES (21, NULL, NULL, NULL,
    BFILENAME('SCOTT_DIR','scott.dat'));
INSERT INTO lob_table VALUES (12, NULL, NULL, NULL,
    BFILENAME('Mary_Dir','mary.dat'));

DECLARE
```

```
    fil_1, fil_2 BFILE;
    result       INTEGER;

BEGIN
    SELECT f_lob INTO fil_1 FROM lob_table WHERE key_value = 21;
    SELECT f_lob INTO fil_2 FROM lob_table WHERE key_value = 12;
    DBMS_LOB.FILEOPEN(fil_1);
    DBMS_LOB.FILEOPEN(fil_2);
    result := DBMS_LOB.COMPARE(fil_1, fil_2);
    IF (result != 0)
    THEN
      DBMS_OUTPUT.PUT_LINE('The two files are different');
    END IF;
    DBMS_LOB.FILECLOSE(fil_1);
    DBMS_LOB.FILECLOSE(fil_2);
END;
```

See Also: DMBS_LOB.FILEGETNAME().

Using the OCI to Manipulate LOBs

The OCI includes functions that you can use to access data stored in BLOBs, CLOBs, NCLOBs, and BFILEs. These functions are mentioned briefly below.

See Also: *Oracle Call Interface Programmer's Guide* for detailed documentation, including parameters, parameter types, return values, and example code.

Table 6–2 OCI Functions for LOB Operations

OCI Function	Description
OCILobAppend()	Appends LOB value to another LOB.
OCILobAssign()	Assigns one LOB locator to another.
OCILobCharSetForm()	Returns the character set form of a LOB.
OCILobCharSetId()	Returns the character set ID of a LOB.
OCILobCopy()	Copies a portion of a LOB into another LOB.
OCILobDisableBuffer-ing()	Disable the buffering subsystem use.

Table 6–2 (Cont.) OCI Functions for LOB Operations

OCI Function	Description
`OCILobEnableBuffering()`	Use the LOB buffering subsystem for subsequent reads and writes of LOB data.
`OCILobErase()`	Erases part of a LOB, starting at a specified offset.
`OCILobFileClose()`	Closes an open BFILE.
`OCILobFileCloseAll()`	Closes all open BFILEs.
`OCILobFileExists()`	Checks whether a BFILE exists.
`OCILobFileGetName()`	Returns the name of a BFILE.
`OCILobFileIsOpen()`	Checks whether a BFILE is open.
`OCILobFileOpen()`	Opens a BFILE.
`OCILobFileSetName()`	Sets the name of a BFILE in a locator.
`OCILobFlushBuffer()`	Flush changes made to the LOB buffering subsystem to the database (sever)
`OCILobGetLength()`	Returns the length of a LOB or a BFILE.
`OCILobIsEqual()`	Checks whether two LOB locators refer to the same LOB.
`OCILobLoadFromFile()`	Loads BFILE data into an internal LOB.
`OCILobLocatorIsInit()`	Checks whether a LOB locator is initialized.
`OCILobLocatorSize()`	Returns the size of a LOB locator.
`OCILobRead()`	Reads a specified portion of a non-null LOB or a BFILE into a buffer.
`OCILobTrim()`	Truncates a LOB.
`OCILobWrite()`	Writes data from a buffer into a LOB, overwriting existing data.

The following chart compares the two interfaces in terms of LOB access:

Table 6–3 Comparison of DBMS_LOB and OCI Interfaces regarding LOB access

OCI (ociap.h)	DBMS_LOB (dbmslob.sql)
N/A	DBMS_LOB.COMPARE()
N/A	DBMS_LOB.INSTR()
N/A	DBMS_LOB.SUBSTR()
OCILobAppend	DBMS_LOB.APPEND()
OCILobAssign	N/A [use Pl/SQL assign operator]
OCILobCharSetForm	N/A
OCILobCharSetId	N/A
OCILobCopy	DBMS_LOB.COPY()
OCILobDisableBuffering	N/A
OCILobEnableBuffering	N/A
OCILobErase	DBMS_LOB.ERASE()
OCILobFileClose	DBMS_LOB.FILECLOSE()
OCILobFileCloseAll	DBMS_LOB.FILECLOSEALL()
OCILobFileExists	DBMS_LOB.FILEEXISTS()
OCILobFileGetName	DBMS_LOB.FILEGETNAME()
OCILobFileIsOpen	DBMS_LOB.FILEISOPEN()
OCILobFileOpen	DBMS_LOB.FILEOPEN()
OCILobFileSetName	N/A (use BFILENAME operator)
OCILobFlushBuffer	N/A
OCILobGetLength	DBMS_LOB.GETLENGTH()
OCILobIsEqual	N/A [use Pl/SQL equal operator]
OCILobLoadFromFile	DBMS_LOB.LOADFROMFILE()
OCILobLocatorIsInit	N/A [always initialize]
OCILobRead	DBMS_LOB.READ()
OCILobTrim	DBMS_LOB.TRIM()
OCILobWrite	DBMS_LOB.WRITE()

DBMS_LOB Package

The DBMS_LOB package provides routines to access BLOBs, CLOBs, NCLOBs, and BFILEs. You can use DBMS_LOB for access and manipulation of specific parts of a LOB, as well as complete LOBs. DBMS_LOB can read as well as modify BLOBs, CLOBs, and NCLOBs, and provides read-only operations on BFILEs.

All DBMS_LOB routines work based on LOB locators. For the successful completion of DBMS_LOB routines, you must provide an input locator that represents a LOB that *already exists* in the database tablespaces or external filesystem.

For internal LOBs, you must first use SQL DDL to define tables that contain LOB columns, and subsequently SQL DML to initialize or populate the locators in these LOB columns.

See Also: "LOB Locator Operations" on page 6-22

For external LOBs, you must ensure that a DIRECTORY object that represents a valid, existing physical directory has been defined, and physical files exist with read permission for Oracle. If your operating system uses case-sensitive pathnames, be sure you specify the directory in the correct format.

See Also: "BFILE Security" on page 6-17

Once the LOBs are defined and created, you may then SELECT a LOB locator into a local PL/SQL LOB variable and use this variable as an input parameter to DBMS_LOB for access to the LOB value. Examples provided with each DBMS_LOB routine will illustrate this in the following sections.

Package Routines

The routines that can modify BLOB, CLOB, and NCLOB values are:

- APPEND() — append the contents of the source LOB to the destination LOB
- COPY() — copy all or part of the source LOB to the destination LOB
- ERASE() — erase all or part of a LOB
- LOADFROMFILE() — load BFILE data into an internal LOB
- TRIM() — trim the LOB value to the specified shorter length
- WRITE()— write data to the LOB from a specified offset

The routines that read or examine LOB values are:

- GETLENGTH() — get the length of the LOB value
- INSTR() — return the matching position of the nth occurrence of the pattern in the LOB
- READ() — read data from the LOB starting at the specified offset
- SUBSTR() — return part of the LOB value starting at the specified offset

The read-only routines specific to BFILEs are:

- FILECLOSE() — close the file
- FILECLOSEALL()— close all previously opened files
- FILEEXISTS() — check if the file exists on the server
- FILEGETNAME() — get the directory alias and file name
- FILEISOPEN() — check if the file was opened using the input BFILE
- locators
- FILEOPEN() — open a file

Datatypes

Parameters for the DBMS_LOB routines use the datatypes:

- BLOB, for a source or destination binary LOB
- RAW, for a source or destination raw buffer (used with BLOB)
- CLOB, for a source or destination character LOB (including NCLOB)
- VARCHAR2, for a source or destination character buffer (used with CLOB and NCLOB)
- INTEGER, to specify the size of a buffer or LOB, the offset into a LOB, or the amount to access

Type Definitions

The DBMS_LOB package defines no special types. NCLOB is a special case of CLOBs for *fixed-width*, multi-byte national character sets. The clause 'ANY_CS' in the specification of DBMS_LOB routines for CLOBs allows them to accept a CLOB or NCLOB locator variable as input.

See Also: "LOB Datatypes" in the *Oracle8 SQL Reference*

Constants

The DBMS_LOB package defines the following constants.

```
LOBMAXSIZE                4294967295
FILE_READONLY              0
```

The maximum LOB size supported in Oracle 8.0 is 4 Gigabytes (2^{32}). However, the amount and offset parameters of the package can have values in the range 1 through 4294967295 (2^{32}-1).

The PL/SQL 3.0 language specifies the maximum size of a RAW or VARCHAR2 variable to be 32767 *bytes*.

Note: The value 32767 bytes is represented by MAXBUFSIZE in the following sections.

DBMS_LOB Exceptions

A DBMS_LOB function or procedure can raise any of the named exceptions shown in Table 6–4 .

Table 6–4 DBMS_LOB Exceptions

Exception	Code in error.msg	Meaning
INVALID_ARGVAL	21560	"argument %s is null, invalid, or out of range"
ACCESS_ERROR	22925	Attempt to read / write beyond maximum LOB size on <n>.
NO_DATA_FOUND	1403	EndofLOB indicator for looping read operations
VALUE_ERROR	6502	Invalid value in parameter.

access_error 22925 "operation would exceed maximum size allowed for a LOB"

noexist_directory 22285 "%s failed - directory does not exist"

nopriv_directory 22286 "%s failed - insufficient privileges on directory"

invalid_directory 22287 "%s failed - invalid or modified directory"

invalid_operation 22288 "%s operation failed"

unopened_file 22289 "cannot perform %s operation on an unopened file"

open_toomany 22290 "%s failed - max limit reached on number of open files"

DBMS_LOB *functions* return a NULL value if any of the input parameters to these routines are NULL or invalid, whereas DBMS_LOB *procedures* will raise exceptions. This behavior is consistent with Oracle8 SQL functions, and procedures in other built-in PL/SQL packages in Oracle8.

DBMS_LOB Security

This section describes the security domain for DBMS_LOB routines operating on internal LOBs (i.e. BLOB, CLOB and NCLOB) when you are using the Oracle server.

> **Note:** Any DBMS_LOB routine called from an anonymous PL/SQL block is executed using the privileges of the current user. Any DBMS_LOB routine called from a stored procedure is executed using the privileges of the owner of the stored procedure.

You can provide secure access to BFILEs using the DIRECTORY feature discussed in "BFILENAME() Function" on page 6-60.

DBMS_LOB General Usage Notes

1. Length, amount and offset parameters are specified in terms of *bytes* for BLOBs and BFILES, and *characters* for CLOBs and NCLOBs.
2. Note that PL/SQL 3.0 language specifies that constraints for both RAW and VARCHAR2 buffers are specified in terms of bytes. For example, if you declare a variable to be

   ```
   charbuf VARCHAR2(3000)
   ```

   charbuf can hold 3000 single byte characters or a 1500 2-byte fixed width characters. This has an important consequence for DBMS_LOB routines for CLOBs and NCLOBs.
3. You must ensure that the character set of the VARCHAR2 buffer in a DBMS_LOB routine for CLOBs exactly matches that of the CLOB. The package specification partially ensures this with the %CHARSET clause, but in certain cases where the

fixed-width character set is actually a subset of a varying width character set, it may not be possible to enforce this.

Hence, it is your responsibility to provide a buffer with the correct character set and enough buffer size for holding all the characters. No translation on the basis of session initialization parameters is performed.

4. Only positive, non-zero values (i.e. a value greater than or equal to 1) are allowed for the AMOUNT and OFFSET parameters. This implies that: negative offsets and ranges observed in Oracle SQL string functions and operators are not allowed.
5. Unless otherwise stated, the default value for an offset parameter is 1, which indicates the first byte in the BLOB or BFILE data, and the first character in the CLOB or NCLOB value. No default values are specified for the AMOUNT parameter — you have to input the values explicitly.
6. You are responsible for locking the row containing the destination internal LOB before calling any routines that modify the LOB such as APPEND, COPY, ERASE, TRIM, or WRITE. These routines do not implicitly lock the row containing the LOB.

BFILE-Specific Usage Notes

1. Recalling that COMPARE(), INSTR() and SUBSTR() are DBMS_LOB specific, the operations COMPARE(), INSTR(), READ(), SUBSTR(), FILECLOSE(), FILECLOSEALL() and LOADFROMFILE() operate only on an *opened* BFILE locator, that is, a successful FILEOPEN() call must precede a call to any of these routines.
2. For the functions FILEEXISTS(), FILEGETNAME() and GETLENGTH(), a file's open/close status is unimportant, however the file must exist physically and you must have adequate privileges on the DIRECTORY object and the file.
3. The DBMS_LOB package does not support any concurrency control mechanism for BFILE operations.
4. In the event of several open files in the session whose closure has not been handled properly, you can use the FILECLOSEALL() routine to close all files opened in the session, and resume file operations from the beginning.
5. If you are the creator of a DIRECTORY or have system privileges, use the CREATE OR REPLACE, DROP and REVOKE statements in SQL with extreme caution.

See Also: "Guidelines for DIRECTORY Usage" on page 6-19.

If you or other grantees of a particular directory object have several open files in a session, any of the above commands can adversely affect file operations. In the event of such abnormal termination, your only choice is to invoke a program or anonymous block that calls FILECLOSEALL(), reopen your files, and restart your file operations.

6. All files opened during a user session are implicitly closed at the end of the session. However, Oracle strongly recommends that you close the files after *both normal and abnormal* termination of operations on the BFILE.

   See Also: "Maximum Number of Open BFILEs" on page 6-20.

   In the event of normal program termination, proper file closure ensures that the number of files that are open simultaneously in the session remains less than SESSION_MAX_OPEN_FILES.

   In the event of abnormal program termination from a PL/SQL program, it is imperative that you provide an exception handler that ensures closure of all files opened in that PL/SQL program. This is necessary because, once an exception occurs, only the exception handler will have access to the BFILE variable in its most current state

   See Also: "Closing BFILEs after Program Termination" on page 6-21.

   Once the exception transfers program control outside the PL/SQL program block, all references to the open BFILEs are lost. The result is a larger open file count which may or may not exceed the SESSION_MAX_OPEN_FILES value.

   For example, consider a READ operation past the end of the BFILE value, which generates a NO_DATA_FOUND exception.

```
DECLARE
       fil bfile;
       pos INTEGER;
       amt binary_INTEGER;
       buf RAW(40);
BEGIN
       SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
       dbms_lob.FILEOPEN(fil, dbms_lob.file_readonly);
       amt := 40; pos := 1 + dbms_lob.getlength(fil); buf := '';
       dbms_lob.read(fil, amt, pos, buf);
       dbms_output.put_line('Read F1 past EOF: '||
           utl_raw.cast_to_varchar2(buf));
       dbms_lob.fileclose(fil);
```

```
END;

ORA-01403: no data found
ORA-06512: at "SYS.DBMS_LOB", line 373
ORA-06512: at line 10
```

Once the exception has occurred, the BFILE locator variable file goes out of scope, and no further operations on the file can be done using that variable. So the solution is to use an exception handler as shown below:

```
DECLARE
     fil bfile;
     pos INTEGER;
     amt binary_INTEGER;
     buf RAW(40);
BEGIN
     SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
     dbms_lob.FILEOPEN(fil, dbms_lob.file_readonly);
     amt := 40; pos := 1 + dbms_lob.getlength(fil); buf := '';
     dbms_lob.read(fil, amt, pos, buf);
     dbms_output.put_line('Read F1 past EOF: '||
          utl_raw.cast_to_varchar2(buf));
     dbms_lob.fileclose(fil);
     exception
     WHEN no_data_found
     then
       BEGIN
         dbms_output.put_line('End of File reached. Closing file');
         dbms_lob.fileclose(fil);
         -- or dbms_lob.filecloseall if appropriate
       END;
END;
     /

Statement processed.
End of File reached. Closing file
```

In general, it is good coding practice to ensure that files opened in a PL/SQL block using DBMS_LOB are closed before normal/abnormal termination of the block.

DBMS_LOB.APPEND() Procedure

You can call the internal APPEND() procedure to append the contents of a source internal LOB to a destination LOB. The procedure appends the complete source

LOB. There are two overloaded APPEND() procedures, as shown in the syntax section below.

Syntax

```
PROCEDURE APPEND (dest_lob IN OUT  BLOB,
                  src_lob  IN      BLOB);
PROCEDURE APPEND (dest_lob IN OUT  CLOB  CHARACTER SET ANY_CS,
                  src_lob  IN      CLOB  CHARACTER SET dest_lob%CHARSET);
```

Parameters

Table 6–5 APPEND Parameters

Parameter Name	Meaning
dest_lob	The locator for the internal LOB to which the data is to be appended.
src_lob	The locator for the internal LOB from which the data is to be read.

Exceptions

VALUE_ERROR, if either the source or the destination LOB is null.

Example

```
PROCEDURE Example_1a IS
    dest_lob, src_lob  BLOB;
BEGIN
    -- get the LOB locators
    -- note that the FOR UPDATE clause locks the row
    SELECT b_lob INTO dest_lob
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_lob INTO src_lob
        FROM lob_table
        WHERE key_value = 21;
    DBMS_LOB.APPEND(dest_lob, src_lob);
    COMMIT;
EXCEPTION
    WHEN some_exception
    THEN handle_exception;
END;

PROCEDURE Example_1b IS
    dest_lob, src_lob  BLOB;
```

```
BEGIN
    -- get the LOB locators
    -- note that the FOR UPDATE clause locks the row
    SELECT b_lob INTO dest_lob
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_lob INTO src_lob
        FROM lob_table
        WHERE key_value = 12;
    DBMS_LOB.APPEND(dest_lob, src_lob);
    COMMIT;
EXCEPTION
    WHEN some_exception
    THEN handle_exception;
END;
```

DBMS_LOB.COMPARE() Function

You can call the COMPARE() function to compare two entire LOBs, or parts of two LOBs. You can only compare LOBs of the same datatype. That is, you compare LOBs of BLOB type with other BLOBs, and CLOBs with CLOBs, and BFILEs with BFILEs. For BFILEs, the file has to be already opened using a successful FILEOPEN() operation for this operation to succeed.

COMPARE() returns zero if the data exactly matches over the range specified by the *offset* and *amount* parameters. Otherwise, a non-zero INTEGER is returned.

For fixed-width *n*-byte CLOBs, if the input amount for COMPARE is specified to be greater than (4294967295 / *n*), then COMPARE matches characters in a range of size (4294967295 / *n*), or Max(length(clob1), length(clob2)), whichever is lesser.

Syntax

```
FUNCTION COMPARE (
    lob_1            IN BLOB,
    lob_2            IN BLOB,
    amount           IN INTEGER := 4294967295,
    offset_1         IN INTEGER := 1,
    offset_2         IN INTEGER := 1)
RETURN INTEGER;

FUNCTION COMPARE (
    lob_1            IN CLOB  CHARACTER SET ANY_CS,
    lob_2            IN CLOB  CHARACTER SET lob_1%CHARSET,
    amount           IN INTEGER := 4294967295,
    offset_1         IN INTEGER := 1,
```

```
    offset_2         IN INTEGER := 1)
RETURN INTEGER;
FUNCTION COMPARE (
    lob_1            IN BFILE,
    lob_2            IN BFILE,
    amount           IN INTEGER,
    offset_1         IN INTEGER := 1,
    offset_2         IN INTEGER := 1)
RETURN INTEGER;
```

Parameters

Table 6–6 COMPARE Parameters

Parameter Name	Meaning
lob_1	LOB locator of first target for comparison.
lob_2	LOB locator of second target for comparison
amount	Number of bytes or characters to compare over.
offset_1	Offset in bytes or characters on the first LOB (origin: 1) for the comparison.
offset_2	Offset in bytes or characters on the first LOB (origin: 1) for the comparison.

Return Values

- INTEGER -- zero if the comparison succeeds, non-zero if not.
- NULL, if
  - amount < 1
  - amount > LOBMAXSIZE
  - offset_1 or offset_2 < 1
  - offset_1 or offset_2 > LOBMAXSIZE

Pragmas

```
PRAGMA RESTRICT_REFERENCES(compare, WNDS, WNPS, RNDS, RNPS);
```

Exceptions

For BFILE operations, UNOPENED_FILE if the file was not opened using the input locator, NOEXIST_DIRECTORY if the directory does not exist,

NOPRIV_DIRECTORY if you do not have privileges for the directory, INVALID_DIRECTORY if the directory has been invalidated after the file was opened, INVALID_OPERATION if the file does not exist, or if you do not have access privileges on the file.

Examples

```
PROCEDURE Example2a IS
    lob_1, lob_2      BLOB;
    retval            INTEGER;
BEGIN
    SELECT b_col INTO lob_1 FROM lob_table
        WHERE key_value = 45;
    SELECT b_col INTO lob_2 FROM lob_table
        WHERE key_value = 54;
    retval := DBMS_LOB.COMPARE(lob_1, lob_2, 5600, 33482,
         128);
    IF retval = 0 THEN
      ;    /* process compared code */
    ELSE
      ;    /* process not compared code */
    END IF;
END;

PROCEDURE Example_2b IS
    fil_1, fil_2       BFILE;
    retval             INTEGER;
BEGIN
    SELECT f_lob INTO fil_1 FROM lob_table WHERE key_value = 45;
    SELECT f_lob INTO fil_2 FROM lob_table WHERE key_value = 54;
    DBMS_LOB.FILEOPEN(fil_1, DBMS_LOB.FILE_READONLY);
    DBMS_LOB.FILEOPEN(fil_2, DBMS_LOB.FILE_READONLY);
    retval := DBMS_LOB.COMPARE(fil_1, fil_2, 5600,
                                3348276, 2765612);
    IF (retval = 0)
    THEN
        ; /* process compared code */
    ELSE
        ; /* process not compared code */
    END IF;
    DBMS_LOB.FILECLOSE(fil_1);
    DBMS_LOB.FILECLOSE(fil_2);
END;
```

DBMS_LOB.COPY() Procedure

You can call the COPY() procedure to copy all, or a part of, a source internal LOB to a destination internal LOB. You can specify the offsets for both the source and destination LOBs, and the number of bytes or characters to copy.

If the offset you specify in the destination LOB is beyond the end of the data currently in this LOB, zero-byte fillers or spaces are inserted in the destination BLOB or CLOB respectively. If the offset is less than the current length of the destination LOB, existing data is overwritten.

It is not an error to specify an amount that exceeds the length of the data in the source LOB. Thus, you can specify a large amount to copy from the source LOB which will copy data from the src_offset to the end of the source LOB.

Syntax

```
PROCEDURE COPY (
  dest_lob    IN OUT BLOB,
  src_lob     IN     BLOB,
  amount      IN     INTEGER,
  dest_offset IN     INTEGER := 1,
  src_offset  IN     INTEGER := 1);

PROCEDURE COPY (
  dest_lob    IN OUT CLOB  CHARACTER SET ANY_CS,
  src_lob     IN     CLOB  CHARACTER SET dest_lob%CHARSET,
  amount      IN     INTEGER,
  dest_offset IN     INTEGER := 1,
  src_offset  IN     INTEGER := 1);
```

Parameters

Table 6–7 COPY Parameters

Parameter Name	Meaning
dest_lob	LOB locator of the copy target.
src_lob	LOB locator of source for the copy.
amount	Number of bytes or characters to copy.
dest_offset	Offset in bytes or characters in the destination LOB (origin: 1) for the start of the copy.
src_offset	Offset in bytes or characters in the source LOB (origin: 1) for the start of the copy.

Return Value

None.

Pragmas

None.

Exceptions

VALUE_ERROR, if any of the input parameters are NULL or invalid.
INVALID_ARGVAL, if

- src_offset or dest_offset < 1
- src_offset or dest_offset > LOBMAXSIZE
- amount < 1
- amount > LOBMAXSIZE

Example

```
PROCEDURE Example_3a IS
    lobd, lobs     BLOB;
    amt            INTEGER := 3000;
BEGIN
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_col INTO lobs
        FROM lob_table
        WHERE key_value = 21;
    DBMS_LOB.COPY(lobd, lobs, amt);
    COMMIT;
   EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;

PROCEDURE Example_3b IS
    lobd, lobs     BLOB;
    amt            INTEGER := 3000;
BEGIN
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    SELECT b_col INTO lobs
        FROM lob_table
```

```
        WHERE key_value = 12;
    DBMS_LOB.COPY(lobd, lobs, amt);
    COMMIT;
   EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

DBMS_LOB.ERASE() Procedure

You can call the ERASE() procedure to erase an entire internal LOB, or part of an internal LOB. The *offset* parameter specifies the starting offset for the erasure, and the *amount* parameter specifies the number of bytes or characters to erase.

When data is erased from the middle of a LOB, zero-byte fillers or spaces are written for BLOBs or CLOBs respectively.

The actual number of bytes or characters erased can differ from the number you specified in the *amount* parameter if the end of the LOB value is reached before erasing the specified number. The actual number of characters or bytes erased is returned in the *amount* parameter.

Syntax

```
PROCEDURE ERASE (
    lob_loc          IN OUT       BLOB,
    amount           IN OUT       INTEGER,
    offset           IN           INTEGER := 1);

PROCEDURE ERASE (
    lob_loc          IN OUT       CLOB,
    amount           IN OUT       INTEGER,
    offset           IN           INTEGER := 1);
```

Parameters

Table 6–8 ERASE Parameters

Parameter Name	Meaning
lob_loc	Locator for the LOB to be erased.
amount	Number of bytes (for BLOBs) or characters (for CLOBs) to be erased.
offset	Absolute offset from the beginning of the LOB in bytes (for BLOBs) or characters (CLOBs).

Return Values

None.

Pragmas

None.

Exceptions

VALUE_ERROR, if any input parameter is NULL.

INVALID_ARGVAL, if

- AMOUNT < 1 or AMOUNT > LOBMAXSIZE
- OFFSET < 1 or OFFSET > LOBMAXSIZE

Example

```
PROCEDURE Example_4 IS
    lobd       BLOB;
    amt        INTEGER := 3000;
BEGIN
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 12 FOR UPDATE;
    DBMS_LOB.ERASE(dest_lob, amt, 2000);
    COMMIT;
END;
```

See Also: DBMS_LOB.TRIM()

DBMS_LOB.FILECLOSE() Procedure

You can call the FILECLOSE() procedure to close a BFILE that has already been opened via the input locator. Note that Oracle has only read-only access to BFILEs. This means that BFILEs cannot be written through Oracle.

Syntax

```
PROCEDURE FILECLOSE (
    file_loc IN OUT BFILE);
```

Parameter

Table 6–9 FILECLOSE Parameter

Parameter Name	Meaning
file_loc	Locator for the BFILE to be closed.

Return Values

None.

Pragmas

None.

Exceptions

VALUE_ERROR, if NULL input value for file_loc. UNOPENED_FILE if the file was not opened with the input locator, NOEXIST_DIRECTORY if the directory does not exist, NOPRIV_DIRECTORY if you do not have privileges for the directory, INVALID_DIRECTORY if the directory has been invalidated after the file was opened, INVALID_OPERATION if the file does not exist, or you do not have access privileges on the file.

Example

```
PROCEDURE Example_5 IS
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    DBMS_LOB.FILEOPEN(fil);
    -- file operations
    DBMS_LOB.FILECLOSE(fil);
    EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

See Also: DBMS_LOB.FILEOPEN(), DBMS_LOB.FILECLOSEALL()

DBMS_LOB.FILECLOSEALL() Procedure

You can call the FILECLOSEALL() procedure to close all BFILEs opened in the session.

Syntax

```
PROCEDURE FILECLOSEALL;
```

Return Values

None.

Pragmas

None.

Exceptions

UNOPENED_FILE, if no file has been opened in the session.

Example

```
PROCEDURE Example_6 IS
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    DBMS_LOB.FILEOPEN(fil);
    -- file operations
    DBMS_LOB.FILECLOSEALL;
    EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

See Also: DBMS_LOB.FILEOPEN(), DBMS_LOB.FILECLOSE()

DBMS_LOB.FILEEXISTS() Function

You can call the FILEEXISTS() function to find out if a given BFILE locator points to a file that actually exists on the server's filesystem.

Syntax

```
FUNCTION FILEEXISTS (
      file_loc     IN    BFILE)
RETURN INTEGER;
```

Parameter

Table 6–10 FILEEXISTS Parameter

Parameter Name	Meaning
file_loc	Locator for the BFILE.

Return Values

INTEGER: 1 if the physical file exists, 0 if it does not exist.

NULL, if:

- file_loc is NULL
- file_loc does not have the necessary directory and OS privileges
- file_loc cannot be read because of an OS error.

Pragmas

```
PRAGMA RESTRICT_REFERENCES(fileexists, WNDS, RNDS, WNPS, RNPS);
```

Exceptions

NOEXIST_DIRECTORY if the directory does not exist, NOPRIV_DIRECTORY if you do not have privileges for the directory, INVALID_DIRECTORY if the directory has been invalidated after the file was opened.

Example

```
PROCEDURE Exsmple_7 IS
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 12;
    IF (DBMS_LOB.FILEEXISTS(fil))
    THEN
        ; -- file exists code
    ELSE
        ; -- file does not exist code
    END IF;
    EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

See Also: DBMS_LOB.FILEISOPEN

DBMS_LOB.FILEGETNAME() Procedure

You can call the FILEGETNAME() procedure to determine the *dir_alias* and filename, given a BFILE locator. This function only indicates the directory alias name and filename assigned to the locator, not if the physical file or directory actually exists. Maximum constraint values for the *dir_alias* buffer is 30, and for the entire pathname is 2000.

Syntax

```
PROCEDURE FILEGETNAME (
      file_loc   IN    BFILE,
      dir_alias  OUT   VARCHAR2
      filename   OUT   VARCHAR2);
```

Parameters

Table 6–11 FILEGETNAME Parameters

Parameter Name	Meaning
file_loc	Locator for the BFILE.
dir_alias	Directory alias
filename	Name of the BFILE

Return Values

None.

Pragmas

None.

Exceptions

VALUE_ERROR, if any of the input parameters are NULL or invalid.
INVALID_ARGVAL, if dir_alias or filename are NULL.

Example

```
PROCEDURE Example_8 IS
    fil BFILE;
    dir_alias VARCHAR2(30);
    name VARCHAR2(2000);
BEGIN
```

```
    IF (DBMS_LOB.FILEEXISTS(fil))
    THEN
        DBMS_LOB.FILEGETNAME(fil, dir_alias, name);
        DBMS_OUTPUT.PUT_LINE ("Opening " || dir_alias || name);
        DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
        -- file operations
        DBMS_OUTPUT.FILECLOSE(fil);
    END IF;
END;
```

See Also: BFILENAME() function

DBMS_LOB.FILEISOPEN() Function

You can call the FILEISOPEN() function to find out whether a BFILE was opened with the give FILE locator. If the input FILE locator was never passed to the DBMS_LOB.FILEOPEN procedure, the file is considered not to be opened by this locator. However, a different locator may have this file open. In other words, openness is associated with a specific locator.

Syntax

```
FUNCTION FILEISOPEN (
      file_loc   IN    BFILE)
RETURN INTEGER;
```

Parameter

Table 6–12 FILEISOPEN Parameter

Parameter Name	Meaning
file_loc	Locator for the BFILE.

Return Values

Integer.

Pragmas

```
PRAGMA RESTRICT_REFERENCES(fileisopen, WNDS, RNDS, WNPS, RNPS);
```

Exceptions

NOEXIST_DIRECTORY if the directory does not exist, NOPRIV_DIRECTORY if you do not have privileges for the directory, INVALID_DIRECTORY if the direc-

tory has been invalidated after the file was opened. INVALID_OPERATION if the file does not exist, or you do not have access privileges on the file.

Example

```
PROCEDURE Example_9 IS
DECLARE
    fil      BFILE;
    pos      INTEGER;
    pattern  VARCHAR2(20);
BEGIN
    SELECT f_lob INTO fil FROM lob_table
        WHERE key_value = 12;
    -- open the file
    IF (FILEISOPEN(fil))
    THEN
        pos := DBMS_LOB.INSTR(fil, pattern, 1025, 6);
         -- more file operations
         DBMS_LOB.FILECLOSE(fil);
   ELSE
        ; -- return error
    END IF;
END;
```

See Also: DBMS_LOB.FILEEXISTS.

DBMS_LOB.FILEOPEN() Procedure

You can call the FILEOPEN procedure to open a BFILE for read-only access. BFILEs may not be written through Oracle.

Syntax

```
PROCEDURE FILEOPEN (
      file_loc   IN OUT  BFILE,
      open_mode  IN      BINARY_INTEGER := file_readonly);
```

Parameters

Table 6–13 FILEOPEN Parameters

Parameter Name	Meaning
file_loc	Locator for the BFILE.
open_mode	Open mode.

Return Values

None.

Pragmas

None.

Exceptions

VALUE_ERROR exception is raised if file_loc or open_mode is NULL. INVALID_ARGVAL exception is raised if open_mode is not equal to FILE_READONLY. OPEN_TOOMANY if the number of open files in the session exceeds SESSION_MAX_OPEN_FILES, NOEXIST_DIRECTORY if the directory does not exist, INVALID_DIRECTORY if the directory has been invalidated after the file was opened, INVALID_OPERATION if the file does not exist, or you do not have access privileges on the file.

Example

```
PROCEDURE Example_10 IS
    fil BFILE;
BEGIN
    -- open BFILE
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    IF (DBMS_LOB.FILEEXISTS(fil))
    THEN
        DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
        -- file operation
        DBMS_LOB.FILECLOSE(fil);
    END IF;
    EXCEPTION
        WHEN some_exception
        THEN handle_exception;
END;
```

See Also: DBMS_LOB.FILECLOSE(), DBMS_LOB.FILECLOSEALL().

DBMS_LOB.GETLENGTH() Function

You can call the GETLENGTH() function to get the length of the specified LOB. The length in bytes or characters is returned. The length returned for a BFILE includes the EOF if it exists. Note that any 0-byte or space filler in the LOB caused by previous ERASE() or WRITE() operations is also included in the length count. The length of an empty internal LOB is 0.

Syntax

```
FUNCTION GETLENGTH (
  lob_loc    IN  BLOB)
RETURN INTEGER;

FUNCTION GETLENGTH (
  lob_loc    IN  CLOB   CHARACTER SET ANY_CS)
RETURN INTEGER;

FUNCTION GETLENGTH (
  lob_loc    IN  BFILE)
RETURN INTEGER;
```

Parameter

Table 6–14 GETLENGTH Parameter

Parameter Name	Meaning
lob_loc	The locator for the LOB whose length is to be returned.

Return Values

The length of the LOB in bytes or characters as an INTEGER. NULL is returned if the input LOB is null. NULL is returned in the following cases for BFILEs:

- lob_loc is NULL
- lob_loc does not have the necessary directory and OS privileges
- lob_loc cannot be read because of an OS read error

Pragmas

```
PRAGMA RESTRICT_REFERENCES(getlength, WNDS, WNPS, RNDS, RNPS);
```

Exceptions

None.

Examples

```
PROCEDURE Example_11a IS
    lobd        BLOB;
    length      INTEGER;
BEGIN
    -- get the LOB locator
    SELECT b_lob INTO lobd FROM lob_table
        WHERE key_value = 42;
    length := DBMS_LOB.GETLENGTH(lob_loc);
```

```
    IF length IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('LOB is null.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The length is '
            || length);
    END IF;
END;
PROCEDURE Example_11b IS
DECLARE
    len INTEGER;
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 12;
    len := DBMS_LOB.LENGTH(fil);
END;
```

DBMS_LOB.INSTR() Function

You can call the INSTR function to return the matching position of the Nth occurrence of the pattern in the LOB, starting from the offset you specify. For CLOBs, the VARCHAR2 buffer (the PATTERN parameter) and the LOB value must be from the same character set (single byte or fixed-width multibyte). For BFILEs, the file has to be already opened using a successful FILEOPEN() operation for this operation to succeed.

Operations that accept RAW or VARCHAR2 parameters for pattern matching, such as INSTR, do not support regular expressions or special matching characters (as in the case of SQL LIKE) in the pattern parameter or substrings.

Syntax

```
FUNCTION INSTR (
  lob_loc    IN   BLOB,
  pattern    IN   RAW,
  offset     IN   INTEGER := 1,
  nth        IN   INTEGER := 1)
RETURN INTEGER;

FUNCTION INSTR (
  lob_loc    IN   CLOB      CHARACTER SET ANY_CS,
  pattern    IN   VARCHAR2  CHARACTER SET lob_loc%CHARSET,
  offset     IN   INTEGER := 1,
  nth        IN   INTEGER := 1)
RETURN INTEGER;
```

```
FUNCTION INSTR (
  lob_loc    IN   BFILE,
  pattern    IN   RAW,
  offset     IN   INTEGER := 1,
  nth        IN   INTEGER := 1)
RETURN INTEGER;
```

Parameters

Table 6–15 INSTR Parameters

Parameter Name	Meaning
lob_loc	The locator for the LOB to be examined.
pattern	The pattern to be tested for. The pattern is a group of RAW bytes for BLOBS, and a character string (VARCHAR2) for CLOBs.
offset	The absolute offset in bytes (BLOBs) or characters (CLOBs) at which the pattern matching is to start.
nth	The occurrence number, starting at 1.

Return Values

INTEGER, offset of the start of the matched pattern, in bytes or characters. It returns 0 if the pattern is not found.

A NULL is returned if:

- any one or more of the IN parameters was null or invalid.
- OFFSET < 1 or OFFSET > LOBMAXSIZE
- nth < 1
- nth > LOBMAXSIZE

Pragmas

```
PRAGMA RESTRICT_REFERENCES(instr, WNDS, WNPS, RNDS, RNPS);
```

Exceptions

For BFILEs, UNOPENED_FILE if the file was not opened using the input locator, NOEXIST_DIRECTORY if the directory does not exist, NOPRIV_DIRECTORY if you do not have privileges for the directory, INVALID_DIRECTORY if the directory has been invalidated after the file was opened,

INVALID_OPERATION if the file does not exist, or if you do not have access privileges on the file.

Examples

```
PROCEDURE Example_12a IS
    lobd        CLOB;
    pattern     VARCHAR2 := 'abcde';
    position    INTEGER := 10000;
BEGIN
-- get the LOB locator
    SELECT b_col INTO lobd
        FROM lob_table
        WHERE key_value = 21;
    position := DBMS_LOB.INSTR(lobd,
                        pattern, 1025, 6);
    IF position = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Pattern not found');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The pattern occurs at '
                || position);
    END IF;
END;

PROCEDURE Example_12b IS
DECLAR    E
    fil BFILE;
    pattern VARCHAR2;
    pos INTEGER;
BEGIN
    -- initialize pattern
    -- check for the 6th occurrence starting from 1025th byte
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 12;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    pos := DBMS_LOB.INSTR(fil, pattern, 1025, 6);
    DBMS_LOB.FILECLOSE(fil);
END;
```

See Also: DBMS_LOB.SUBSTR()

DBMS_LOB.LOADFROMFILE() Procedure

You can call the LOADFROMFILE() procedure to copy all, or a part of, a source external LOB (BFILE) to a destination internal LOB. You can specify the offsets for both the source and destination LOBs, and the number of bytes to copy from the source BFILE. Note that the amount and src_offset, since they refer to the BFILE, are in

terms of bytes and the destination offset is either in bytes or characters for BLOBs and CLOBs respectively.

> **Note:** The input BFILE must have already been opened prior to using this procedure. Also, no character set conversions are performed implicitly when binary BFILE data is loaded into a CLOB. The BFILE data must already be in the same character set as the CLOB in the database. No error checking is performed to verify this.

If the offset you specify in the destination LOB is beyond the end of the data currently in this LOB, zero-byte fillers or spaces are inserted in the destination BLOB or CLOB respectively. If the offset is less than the current length of the destination LOB, existing data is overwritten.

It is not an error to specify an amount that exceeds the length of the data in the source BFILE. Thus, you can specify a large amount to copy from the BFILE which will copy data from the src_offset to the end of the BFILE.

Syntax

```
PROCEDURE loadfromfile (
   dest_lob    IN OUT BLOB,
   src_file    IN     BFILE,
   amount      IN     INTEGER,
   dest_offset IN     INTEGER := 1,
   src_offset  IN     INTEGER  := 1);

PROCEDURE LOADFROMFILE(
   dest_lob    IN OUT CLOB CHARACTER SET ANY_CS,
   src_file    IN     BFILE,
   amount      IN     INTEGER,
   dest_offset IN     INTEGER := 1,
   src_offset  IN     INTEGER := 1);
```

Parameters

Table 6–16 INSTR Parameters

Parameter Name	Meaning
dest_lob	LOB locator of the target for the load.
src_file	BFILE locator of the source for the load.
amount	Number of bytes to load from the BFILE.

Table 6–16 INSTR Parameters

Parameter Name	Meaning
dest_offset	Offset in bytes or characters in the destination LOB (origin: 1) for the start of the load.
src_offset	Offset in bytes in the source BFILE (origin: 1) for the start of the load.

Return Values

None

Pragmas

None.

Exceptions

VALUE_ERROR, if any of the input parameters are NULL or invalid.

INVALID_ARGVAL, if

- src_offset or dest_offset < 1
- src_offset or dest_offset > LOBMAXSIZE
- amount < 1
- amount > LOBMAXSIZE

Examples

```
PROCEDURE Example_l2f IS
  lobd       BLOB;
  fils       BFILE := BFILENAME('SOME_DIR_OBJ','some_file');
  amt        INTEGER := 4000;
BEGIN
  DBMS_LOB.FILEOPEN(fils, dbms_lob.file_readonly);
  DBMS_LOB.LOADFROMFILE(lobd, fils, amt);
  COMMIT;
  DBMS_LOB.FILECLOSE(fils);
END;
```

DBMS_LOB.READ() Procedure

You can call the READ() procedure to read a piece of a LOB, and return the specified amount into the *buffer* parameter, starting from an absolute *offset* from the beginning of the LOB.

The number of bytes or characters actually read is returned in the *amount* parameter. If the end of LOB value is reached during a READ(), *amount* will be set to 0, and a NO_DATA_FOUND exception will be raised.

Syntax

```
PROCEDURE READ (
  lob_loc   IN      BLOB,
  amount    IN OUT  BINARY_INTEGER,
  offset    IN      INTEGER,
  buffer    OUT     RAW);

PROCEDURE READ (
  lob_loc   IN      CLOB     CHARACTER SET ANY_CS,
  amount    IN OUT  BINARY_INTEGER,
  offset    IN      INTEGER,
  buffer    OUT     VARCHAR2 CHARACTER SET lob_loc%CHARSET);

PROCEDURE READ (
  lob_loc   IN       BFILE,
  amount    IN OUT   BINARY_INTEGER,
  offset    IN       INTEGER,
  buffer    OUT      RAW);
```

Parameters

Table 6–17 READ Parameters

Parameter Name	Meaning
lob_loc	The locator for the LOB to be read.
amount	The number of bytes or characters to be read.
offset	The offset in bytes (for BLOBs) or characters (for CLOBs) from the start of the LOB (origin: 1).
buffer	The output buffer for the read operation.

Return Values

None.

Pragmas

None.

Exceptions

READ can raise any of the following exceptions:

- VALUE_ERROR
  - any of *lob_loc*, *amount*, or *offset* parameters are null
- INVALID_ARGVAL
  - AMOUNT < 1
  - AMOUNT > MAXBUFSIZE
  - OFFSET < 1
  - OFFSET > LOBMAXSIZE
  - AMOUNT is greater, in bytes or characters, than the capacity of BUFFER
- NO_DATA_FOUND
  - the end of the LOB is reached and there are no more bytes or characters to read from the LOB. AMOUNT has a value of 0.
- For BFILEs operations, UNOPENED_FILE if the file is not opened using the input locator, NOEXIST_DIRECTORY if the directory does not exist, NOPRIV_DIRECTORY if you do not have privileges for the directory, INVALID_DIRECTORY if the directory has been invalidated after the file was opened, INVALID_OPERATION if the file does not exist, or if you do not have access privileges on the file

Examples

```
PROCEDURE Example_13a IS
    src_lob        BLOB;
    buffer         RAW;
    amt            BINARY_INTEGER := 32767;
    pos            INTEGER := 2147483647;
BEGIN
    SELECT b_col INTO src_lob
        FROM lob_table
        WHERE key_value = 21;
    LOOP
        DBMS_LOB.READ (src_lob, amt, pos, buffer);
        /* process the buffer */
```

```
        pos := pos + amt;
    END LOOP;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('End of data');
END;

PROCEDURE Example_13b IS
    fil BFILE;
    buf RAW(32767);
    amt BINARY_INTEGER := 32767;
    pos INTEGER := 2147483647;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    LOOP
        DBMS_LOB.READ(fil, amt, pos, buf);
        -- process contents of buf
        pos := pos + amt;
    END LOOP;
    EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
      BEGIN
        DBMS_OUTPUT.PUTLINE ('End of LOB value reached');
        DBMS_LOB.FILECLOSE(fil);
      END;
END;

/* Example for efficient I/O on OS that performs */
/* better with block I/O rather than stream I/O  */
PROCEDURE Example_13c IS
    fil BFILE;
    amt BINARY_INTEGER := 1024; -- or n x 1024 for reading n
    buf RAW(1024); -- blocks at a time
    tmpamt BINARY_INTEGER;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 99;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    LOOP
        DBMS_LOB.READ(fil, amt, pos, buf);
        -- process contents of buf
        pos := pos + amt;
    END LOOP;
    EXCEPTION
```

```
        WHEN NO_DATA_FOUND
        THEN
            BEGIN
                DBMS_OUTPUT.PUTLINE ('End of data reached');
                DBMS_LOB.FILECLOSE(fil);
            END;
END;
```

DBMS_LOB.SUBSTR() Function

You can call the SUBSTR() function to return *amount* bytes or characters of a LOB, starting from an absolute *offset* from the beginning of the LOB.

For fixed-width *n*-byte CLOBs, if the input amount for SUBSTR() is specified to be greater than (32767 / *n*), then SUBSTR() returns a character buffer of length (32767 / *n*), or the length of the CLOB, whichever is lesser.

Syntax

```
FUNCTION SUBSTR(
  lob_loc     IN    BLOB,
  amount      IN    INTEGER := 32767,
  offset      IN    INTEGER := 1)
RETURN RAW;

FUNCTION SUBSTR(
  lob_loc     IN    CLOB   CHARACTER SET ANY_CS,
  amount      IN    INTEGER := 32767,
  offset      IN    INTEGER := 1)
RETURN VARCHAR2 CHARACTER SET lob_loc%CHARSET;

FUNCTION SUBSTR(
  lob_loc     IN    BFILE,
  amount      IN    INTEGER := 32767,
  offset      IN    INTEGER := 1)
RETURN RAW;
```

Parameters

Table 6–18 SUBSTR Parameters

Parameter Name	Meaning
lob_loc	The locator for the LOB to be read.
amount	The number of bytes or characters to be read.

Table 6–18 SUBSTR Parameters

Parameter Name	Meaning
offset	The offset in bytes (for BLOBs) or characters (for CLOBs) from the start of the LOB (origin: 1).

Return Values

RAW, for the function overloading that has a BLOB or BFILE in parameter.

VARCHAR2, for the CLOB version.

NULL, if:

- any input parameter is null
- AMOUNT < 1
- AMOUNT > 32767
- OFFSET < 1
- OFFSET > LOBMAXSIZE

Pragmas

```
PRAGMA RESTRICT_REFERENCES(substr, WNDS, WNPS, RNDS, RNPS);
```

Exceptions

For BFILE operations, UNOPENED_FILE if the file is not opened using the input locator, NOEXIST_DIRECTORY if the directory does not exist, NOPRIV_DIRECTORY if you do not have privileges for the directory, INVALID_DIRECTORY if the directory has been invalidated after the file was opened, INVALID_OPERATION if the file does not exist, or if you do not have access privileges on the file

Example

```
PROCEDURE Example_14a IS
    src_lob         CLOB;
    pos             INTEGER := 2147483647;
    buf             VARCHAR2(32000);
BEGIN
    SELECT c_lob INTO src_lob FROM lob_table
        WHERE key_value = 21;
    buf := DBMS_LOB.SUBSTR(src_lob, 32767, pos);
    /* process the data */
END;
```

```
PROCEDURE Example_14b IS
    fil BFILE;
    pos INTEGER := 2147483647;
    pattern RAW;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE key_value = 21;
    DBMS_LOB.FILEOPEN(fil, DBMS_LOB.FILE_READONLY);
    pattern := DBMS_LOB.SUBSTR(fil, 255, pos);
    DBMS_LOB.FILECLOSE(fil);
END;
```

See Also: DBMS_LOB.INSTR(), DBMS_LOB.READ()

DBMS_LOB.TRIM() Procedure

You can call the TRIM() procedure to trim the value of the internal LOB to the length you specify in the *newlen* parameter. Specify the length in bytes for BLOBs, and in characters for CLOBs.

If you attempt to TRIM() an empty LOB, nothing occurs, and TRIM() returns no error. If the new length that you specify in *newlen* is greater than the size of the LOB, an exception is raised.

Syntax

```
FUNCTION TRIM (
    lob_loc         IN    BLOB,
    newlen          IN    INTEGER);

FUNCTION TRIM (
    lob_loc         IN    CLOB,
    newlen          IN    INTEGER):
```

Parameters

Table 6–19 TRIM Parameters

Parameter Name	Meaning
lob_loc	The locator for the internal LOB whose length is to be trimmed.
newlen	The new, trimmed length of the LOB value in bytes for BLOBs or characters for CLOBs.

Return Values

None.

Pragmas

None.

Exceptions

VALUE_ERROR, if lob_loc is null.

INVALID_ARGVAL, if

- NEW_LEN < 0
- NEW_LEN > LOBMAXSIZE

Example

```
PROCEDURE Example_15 IS
    lob_loc        BLOB;
BEGIN
-- get the LOB locator
    SELECT b_col INTO lob_loc
        FROM lob_table
        WHERE key_value = 42 FOR UPDATE;
    DBMS_LOB.TRIM(lob_loc, 4000);
    COMMIT;
END;
```

See Also: DBMS_LOB.ERASE()

\DBMS_LOB.WRITE() Procedure

You can call the WRITE() procedure to write a specified *amount* of data into an internal LOB, starting from an absolute *offset* from the beginning of the LOB. The data is written from the *buffer* parameter.

WRITE() replaces (overwrites) any data that already exists in the LOB at the offset, for the length you specify.

It is an error if the input amount is more than the data in the buffer. If the input amount is less than the data in the buffer, only amount bytes/characters from the buffer is written to the LOB. If the offset you specify is beyond the end of the data currently in the LOB, zero-byte fillers or spaces are inserted in the BLOB or CLOB respectively.

Syntax

```
PROCEDURE WRITE (
  lob_loc  IN OUT  BLOB,
  amount   IN      BINARY_INTEGER,
  offset   IN      INTEGER,
  buffer   IN      RAW);

PROCEDURE WRITE (
  lob_loc  IN OUT  CLOB   CHARACTER SET ANY_CS,
  amount   IN      BINARY_INTEGER,
  offset   IN      INTEGER,
  buffer   IN      VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

Parameters

Table 6–20 WRITE Parameters

Parameter Name	Meaning
lob_loc	The locator for the internal LOB to be written to.
amount	The number of bytes or characters to write, or that were written.
offset	The offset in bytes (for BLOBs) or characters (for CLOBs) from the start of the LOB (origin: 1) for the write operation.
buffer	The input buffer for the write.

Return Values

None.

Pragmas

None.

Exceptions

- VALUE_ERROR
  - if any of LOB_LOC, AMOUNT, or OFFSET parameters are null, out of range, or invalid
- INVALID_ARGVAL
  - AMOUNT < 1
  - AMOUNT > MAXBUFSIZE
  - OFFSET < 1
  - OFFSET > LOBMAXSIZE

Example

```
PROCEDURE Example_16 IS
    lob_loc        BLOB;
    buffer         RAW;
    amt            BINARY_INTEGER := 32767;
    pos            INTEGER := 2147483647;
    i              INTEGER;
BEGIN
    SELECT b_col INTO lob_loc
        FROM lob_table
        WHERE key_value = 12;
    FOR i IN 1..3 LOOP
        DBMS_LOB.WRITE (lob_loc, amt, pos, buffer);
        /* fill in more data */
        pos := pos + amt;
    END LOOP;
    EXCEPTION4
        WHEN some_exception
        THEN handle_exception;
END;
```

See Also: DBMS_LOB.APPEND(), DBMS_LOB.COPY().

LOB Restrictions

The use of LOBs are subject to some restrictions:

- LOBs must be stored in tables -- they cannot be transient / temporary.
- A LONG datatype may not be converted nor migrated to a LOB datatype and vice versa.

  A workaround is to do the following:

  1. Write the data in the long RAW to a server side file.
  2. Use the Oracle8 command CREATE DIRECTORY to point to the directory where the file was written.
  3. Use the Oracle8 command OCILobLoadFromFile() or DBMS_LOB.LOADFROMFILE() to populate the LOB with the data in the file.

  If the LONG isn't too big, another way is to read the LONG into a buffer and call OCILobWrite or DBMS_LOB.WRITE() to write the LONG data to the LOB.

  In either case, you'll need to either add a LOB column to the original table or create a new table that contains the LOB column. Oracle8 does not allow changing the datatype of a column to a LOB type.

- Distributed LOBs are not supported. Specifically, this means that the user cannot use a remote locator in the SELECT and WHERE clauses. This includes using DBMS_LOB package functions. In addition, references to objects in remote tables with or without LOB attributes is not allowed.

  For example, the following operations are invalid:

  – SELECT lobcol from table1@remote_site;
  – INSERT INTO lobtable select type1.lobattr from table1@remote_site;
  – SELECT dbms_lob.length(lobcol) from table1@remote_site;

  Valid operations on LOB columns in remote tables include:

  – CREATE TABLE as select * from table1@remote_site;
  – INSERT INTO t select * from table1@remote_site;
  – UPDATE t set lobcol = (select lobcol from table1@remote_site);
  – INSERT INTO table1@remote...
  – UPDATE table1@remote...
  – DELETE table1@remote...

- There is no loader (direct or conventional path) support for LOBs. Instead, use OCILobLoadFromFile(), DBMS_LOB.LOADFROMFILE(), or OCILobWrite() with streaming.
- When binding an internal LOB in order to use piece-wise INSERT/UPDATE, the bind variable may be of type SQLT_CHR or SQLT_LBI but is limited to 4k. You cannot bind a SQLT_LNG to a LOB or a SQLT_LBI that is longer than 4k.

Also, LOBs are not allowed in the following places:

- LOBs are not allowed in partitioned tables nor are they allowed in clustered tables and thus cannot be a cluster key.
- LOBs are not allowed in GROUP BY, ORDER BY, SELECT DISTINCT, aggregates and JOINS. However, UNION ALL is allowed on tables with LOBs. UNION, MINUS, and SELECT DISTINCT are allowed on LOB attributes if the object type has a MAP or ORDER function.
- LOBS are not analyzed in ANALYZE... COMPUTE/ESTIMATE STATISTICS statements.
- LOBs are not allowed in index only tables.
- LOBs are not allowed in VARRAYs.
- NCLOBs are not allowed as attributes in object types but NCLOB parameters are allowed in methods.
- Triggers are not supported on LOBs. However, you can use a LOB in the body of a trigger as follows:
  - you cannot write to a LOB (:old or :new value) in any kind of trigger.
  - in regular triggers you can read the :old value but you cannot read the :newvalue.
  - in INSTEAD OF triggers, you can read the :old and the :new values, which is to say that the :old and :new values can be read but not written.
  - you cannot specify LOB type columns in an OF clause, because BFILE types can be updated without updating the underlying table on which the trigger is defined.
  - using OCI functions or the DBMS_LOB package to update LOB values or LOB attributes of object columns will not fire triggers defined on the table containing the columns or the attributes.

- Client-side PL/SQL procedures may not call the DBMS_LOB package routines. However, you can use server-side PL/SQL procedures or anonymous blocks in PRO*C to call the DBMS_LOB package routines.

7

User-Defined Datatypes — An Extended Example

This chapter contains an extended example of how to use user-defined types. The chapter has the following major sections:

- Introduction
- A Purchase Order Example

Introduction

User-defined types are schema objects in which users formalize the data structures and operations that appear in their applications.

> **See Also:** *Oracle8 Concepts* for a discussion of user-defined types and how to use them.

The example in this chapter illustrates the most important aspects of defining and using user-defined types. The definitions of object type methods use the PL/SQL language. The remainder of the example uses Oracle SQL.

> **See Also:** *Oracle8 SQL Reference* for a complete description of SQL syntax and usage.

PL/SQL provides additional capabilities beyond those illustrated here, especially in the area of accessing and manipulating the elements of collections.

> **See Also:** *PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities.

Client applications that use the Oracle call interface (OCI) can take advantage of its extensive facilities for accessing objects and collections and manipulating them on the client side.

> **See Also:** *Programmer's Guide to the Oracle Call Interface* for a complete discussion of those facilities.

A Purchase Order Example

This example is based on a simple business activity: managing the data in customer orders. The example is presented in three parts. The first two are in this chapter. The third is in Chapter 8, "Object Views—An Extended Example".

Each part implements a schema to support the basic activity. The first part implements the schema using only Oracle's built-in datatypes. This is called the *relational approach*. Using this approach, you create tables to hold the application's data and use well-known techniques to implement the application's entity relationships.

The second and third parts use user-defined types (UDTs) to translate the entities and relationships directly into schema objects that can be manipulated by a DBMS. This is called the *object-relational* approach. The second and third parts UDTs. They differ only in the way they implement the underlying data storage:

- The second part of the example creates object tables to hold the underlying data instead of the relational tables created in the first part.
- The third part uses the relational tables created in the first part. Rather than building object tables, it uses object views to represent virtual object tables.

Entities and Relationships

The basic entities in this example are:

- Customers
- The stock of products for sale
- Purchase orders

Customers have a one-to-many relationship with purchase orders because a customer can place many orders, but a given purchase order is placed by a single customer.

Purchase orders have many-to-many relationship with stock items because purchase order can contain many stock items, and a stock item can appear on many purchase orders.

The usual way to manage the many-to-many relationship between purchase orders and stock is to introduce another entity called a line item list. A purchase order can have an arbitrary number of line items, but each line item belongs to a single purchase order. A stock item can appear on many line items, but each line item refers to a single stock item.

Table 7–1 lists the required information about each of these entities for an application that manages customer orders needs.

Table 7–1 Information Required about Entities in the Purchase Order Example

Entity	Required Information
`Customer`	Contact information
`Stock`	Item identification, cost, and taxability code
`Purchase Order`	Customer, order and ship dates, shipping address
`Line Item List`	Stock item, quantity, price, discount for each line item

The problem is that the real-world attributes entities are complex, and so they each require a complex set of attributes to map their data structure. An address contains

attributes such as street, city, state, and zipcode. A customer may have several phone numbers. The line item list is an entity in its own right and also an attribute of a purchase order. Standard built-in types cannot represent them directly. The object-relational approach makes it possible to handle this rich structure in different ways.

Part 1: Relational Approach

The relational approach normalizes entities and their attributes, and structures the customer, purchase order, and stock entities into tables. It breaks addresses into their standard components. It sets an arbitrary limit on the number of telephone numbers a customer can have and assigns a column to each.

The relational approach separates line items from their purchase orders and puts them into a table of their own. The table has columns for foreign keys to the stock and purchase order tables.

Tables

The relational approach results in the following tables:

```
CREATE TABLE customer_info (
  custno       NUMBER,
  custname     VARCHAR2(200),
  street       VARCHAR2(200),
  city         VARCHAR2(200),
  state        CHAR(2),
  zip          VARCHAR2(20),
  phone1       VARCHAR2(20),
  phone2       VARCHAR2(20),
  phone3       VARCHAR2(20),
  PRIMARY KEY (custno)
  ) ;

CREATE TABLE purchase_order (
  pono         NUMBER,
  custno       NUMBER REFERENCES customer_info,
  orderdate    DATE,
  shiptodate   DATE,
  shiptostreet VARCHAR2(200),
  shiptocity   VARCHAR2(200),
  shiptostate  CHAR(2),
  shiptozip    VARCHAR2(20),
  PRIMARY KEY (pono)
  ) ;
```

```
CREATE TABLE stock_info (
  stockno      NUMBER PRIMARY KEY,
  cost         NUMBER,
  tax_code     NUMBER
  ) ;

CREATE TABLE line_items (
  lineitemno   NUMBER,
  pono         NUMBER REFERENCES purchase_order,
  stockno      NUMBER REFERENCES stock_info,
  quantity     NUMBER,
  discount     NUMBER,
  PRIMARY KEY (pono, lineitemno)
  ) ;
```

The first table, CUSTOMER_INFO, stores information about customers. It does not refer to the other tables, but the PURCHASE_ORDER table contains a CUSTNO column, which contains a foreign key to the CUSTOMER_INFO table.

The foreign key implements the many-to-one relationship of purchase orders to customers. Many purchase orders might come from a single customer, but only one customer issues a given purchase order.

The LINE_ITEMS table contains foreign keys PONO to the PURCHASE_ORDER table and STOCKNO to the STOCK_INFO table.

Inserting Values

In an application based on the tables defined in the previous section, statements such as the following insert data into the tables:

```
INSERT INTO customer_info
  VALUES (1, 'Jean Nance', '2 Avocet Drive',
         'Redwood Shores', 'CA', '95054',
         '415-555-1212', NULL, NULL) ;

INSERT INTO customer_info
  VALUES (2, 'John Nike', '323 College Drive',
         'Edison', 'NJ', '08820',
         '609-555-1212', '201-555-1212', NULL) ;

INSERT INTO purchase_order
  VALUES (1001, 1, SYSDATE, '10-MAY-1997',
          NULL, NULL, NULL, NULL) ;
```

```
INSERT INTO purchase_order
  VALUES (2001, 2, SYSDATE, '20-MAY-1997',
         '55 Madison Ave', 'Madison', 'WI', '53715') ;

INSERT INTO stock_info VALUES(1004, 6750.00, 2) ;
INSERT INTO stock_info VALUES(1011, 4500.23, 2) ;
INSERT INTO stock_info VALUES(1534, 2234.00, 2) ;
INSERT INTO stock_info VALUES(1535, 3456.23, 2) ;

INSERT INTO line_items VALUES(01, 1001, 1534, 12,  0) ;
INSERT INTO line_items VALUES(02, 1001, 1535, 10, 10) ;
INSERT INTO line_items VALUES(10, 2001, 1004,  1,  0) ;
INSERT INTO line_items VALUES(11, 2001, 1011,  2,  1) ;
```

Selecting

Selecting

Assuming that values have been inserted into these tables in the usual way, your application would execute queries of the following kind to retrieve the necessary information from the stored data.

Customer and Line Item Data for Purchase Order 1001

```
SELECT   C.custno, C.custname, C.street, C.city, C.state,
         C.zip, C.phone1, C.phone2, C.phone3,

         P.pono, P.orderdate,

         L.stockno, L.lineitemno, L.quantity, L.discount

 FROM    customer_info  C,
         purchase_order P,
         line_items     L

 WHERE   C.custno = P.custno
  AND    P.pono = L.pono
  AND    P.pono = 1001;
```

Total Value of Each Purchase Order

```
SELECT   P.pono, SUM(S.cost * L.quantity)

 FROM    purchase_order P,
         line_items     L,
         stock_info     S
```

```
WHERE   P.pono = L.pono
 AND    L.stockno = S.stockno

GROUP BY P.pono;
```

Purchase Order and Line Item Data Involving Stock Item 1004

```
SELECT  P.pono, P.custno,
        L.stockno, L.lineitemno, L.quantity, L.discount

FROM    purchase_order P,
        line_items     L

WHERE     P.pono = L.pono
  AND     L.stockno = 1004;
```

Updating

Given the schema objects described above, you would execute statements such as the following to update the stored data:

Update the Quantity for Purchase Order 01 and Stock Item 1001

```
UPDATE  line_items

 SET    quantity = 20

 WHERE  pono     = 1
  AND   stockno  = 1001 ;
```

Deleting

In an application based on the tables defined earlier, statements such as the following delete stored data:

Delete Purchase Order 1001

```
DELETE
 FROM   line_items
 WHERE  pono = 1001 ;

DELETE
 FROM   purchase_order
 WHERE  pono = 1001 ;
```

Part 2: Object-Relational Approach with Object Tables

Why a Different Approach May Be Needed

Applications written in third generation languages (3GL) such as C++, are able to implement highly complex user-defined types that encapsulate data with methods. By contrast, SQL provides only basic, scalar types and no way of encapsulating these with relevant operations.

So why not create applications using a 3GL? First, DBMSs provide a functionality that would take millions of person-hours to replicate. Second, one of the problems of information management using 3GLs is that they are not persistent — or, if they are persistent, that they sacrifice security to obtain the necessary performance by way of locating the application logic and the data logic in the same address space. Neither trade-off is acceptable to users of DBMSs for whom both persistence and security are basic requirements.

This leaves the application developer with the problem of simulating complex types by some form of mapping into SQL. Apart from the many person-hours required, this involves serious problems of implementation. You must

- translate from application logic into data logic on ‘write’, and then
- perform the reverse process on ‘read’ (and vica versa).

Obviously, there is heavy traffic back and forth between the client address space and that of the server, with the accompanying decrement in performance. And if client and server are on different machines, the toll may on performance from network roundtrips may be considerable.

O-R technology resolves these problems. In the course of this and the following chapter we will consider examples that implement this new functionality.

The Object-Relational (O-R) Way

The O-R approach to the example we have been considering begins with the same entity relationships outlined in “Entities and Relationships” on page 7-3. But user-defined types make it possible to carry more of that structure into the database schema.

Rather than breaking up addresses or the customer’s contact phones into unrelated columns in relational tables, the O-R approach defines types to represent them; rather than breaking line items out into a separate table, the O-R approach allows them to stay with their respective purchase orders as nested tables.

In the O-R approach, the main entities — *customers*, *stock*, and *purchase orders* — become objects. Object references express the *n: 1* relationships between them. Collection types model their multi-valued attributes.

Given an O-R strategy, there are two approaches to implementation:

- create and populate object tables
- use object views to represent virtual object tables from existing relational data.

The remainder of this chapter develops the O-R schema and shows how to implement it with object tables. Chapter 8, "Object Views—An Extended Example" implements the same schema with object views.

Defining Types

The following statements set the stage:

```
CREATE TYPE line_item_t ;
CREATE TYPE purchase_order_t ;
CREATE TYPE stock_info_t ;
```

The preceding three statements define incomplete object types. The incomplete definitions notify Oracle that full definitions are coming later. Oracle allows types that refer to these types to compile successfully. Incomplete type declarations are like forward declarations in C and other programming languages.

The next statement defines an array type.

```
CREATE TYPE phone_list_t AS VARRAY(10) OF VARCHAR2(20) ;
```

The preceding statement defines the type PHONE_LIST_T. Any data unit of type PHONE_LIST_T is a VARRAY of up to 10 telephone numbers, each represented by a data item of type VARCHAR2.

A list of phone numbers could occupy a VARRAY or a nested table. In this case, the list is the set of contact phone numbers for a single customer. A VARRAY is a better choice than a nested table for the following reasons:

- The order of the numbers might be important. VARRAYs are ordered. Nested tables are unordered.
- The number of phone numbers for a specific customer is small. VARRAYs force you to specify a maximum number of elements (10 in this case) in advance. They use storage more efficiently than nested tables which have no special size limitations.

- There is no reason to query the phone number list, so the nested table format offers no benefit.

In general, if ordering and bounds are not important design considerations, designers can use the following rule of thumb for deciding between VARRAYs and nested tables: If you need to query the collection, use nested tables; if you intend to retrieve the collection as a whole, use VARRAYs.

```
CREATE TYPE address_t AS OBJECT (
  street  VARCHAR2(200),
  city    VARCHAR2(200),
  state   CHAR(2),
  zip     VARCHAR2(20)
  ) ;
```

The preceding statement defines the object type ADDRESS_T. Data units of this type represent addresses. All of their attributes are character strings, representing the usual parts of a slightly simplified mailing address.

The next statement defines an object type that uses other user-defined types as building blocks. The object type also has a comparison method.

```
CREATE TYPE customer_info_t AS OBJECT (
  custno     NUMBER,
  custname   VARCHAR2(200),
  address    address_t,
  phone_list phone_list_t,

  ORDER MEMBER FUNCTION
    cust_order(x IN customer_info_t) RETURN INTEGER,

  PRAGMA RESTRICT_REFERENCES (
    cust_order,  WNDS, WNPS, RNPS, RNDS)
  ) ;
```

The preceding statement defines the object type CUSTOMER_INFO_T. Data units of this type are objects that represent blocks of information about specific customers. The attributes of a CUSTOMER_INFO_T object are a number, a character string, an ADDRESS_T object, and a VARRAY of type PHONE_LIST_T.

Every CUSTOMER_INFO_T object also has an associated order method, one of the two types of comparison methods. Whenever Oracle needs to compare two CUSTOMER_INFO_T objects, it invokes the CUST_ORDER method to do so.

The two types of comparison methods are map methods and order methods. This application uses one of each for purposes of illustration.

Note: An ORDER method must be called for every two objects being compared, whereas a MAP method is called once per object. In general, when sorting a set of objects, the number of times an ORDER method would be called is more than the number of times a MAP method would be called. Given that the system can perform scalar value comparisons very efficiently, coupled with the fact that calling a user-defined function is slower compared to calling a kernel implemented function, sorting objects using the ORDER method is relatively slow compared to sorting the mapped scalar values (returned by the MAP function).

See Also:

- *Oracle8 Concepts* for a discussion of ORDER and MAP methods.
- *PL/SQL User's Guide and Reference* for details of how to use pragma declarations.

The statement does not include the actual PL/SQL program implementing the method CUST_ORDER. That appears in a later section.

The next statement completes the definition of the incomplete object type LINE_ITEM_T declared at the beginning of this section.

```
CREATE TYPE line_item_t AS OBJECT (
  lineitemno NUMBER,
  STOCKREF   REF stock_info_t,
  quantity   NUMBER,
  discount   NUMBER
  ) ;
```

Data units of type LINE_ITEM_T are objects that represent line items. They have three numeric attributes and one REF attribute. The LINE_ITEM_T models the line item entity and includes an object reference to the corresponding stock object.

```
CREATE TYPE line_item_list_t AS TABLE OF line_item_t ;
```

The preceding statement defines the table type LINE_ITEM_LIST_T. A data unit of this type is a nested table, each row of which contains a LINE_ITEM_T object. A nested table of line items is better choice to represent the multivalued line item list of a purchase order than a VARRAY of LINE_ITEM_T objects would be, for the following reasons:

- Querying the contents of line items is likely to be a requirement for most applications. This is an inefficient operation for VARRAYs because it involves casting the VARRAY to a nested table first.
- Indexing on line item data may be a requirement in some applications. Nested tables allow this, but it is not possible with VARRAYs.
- The order of line items is usually unimportant, and the line item number can be used to specify an order when necessary.
- There is no practical upper bound on the number of line items on a purchase order. Using a VARRAY requires specifying an upper bound on the number of elements.

The following statement completes the definition of the incomplete object type PURCHASE_ORDER_T declared at the beginning of this section.

```
CREATE TYPE purchase_order_t AS OBJECT (
  pono           NUMBER,
  custref        REF customer_info_t,
  orderdate      DATE,
  shipdate       DATE,
  line_item_list line_item_list_t,
  shiptoaddr     address_t,

  MAP MEMBER FUNCTION
    ret_value RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (
      ret_value, WNDS, WNPS, RNPS, RNDS),

  MEMBER FUNCTION
    total_value RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (total_value, WNDS, WNPS)
  ) ;
```

The preceding statement defines the object type PURCHASE_ORDER_T. Data units of this type are objects representing purchase orders. They have six attributes, including a REF, a nested table of type LINE_ITEM_LIST_T, and an ADDRESS_T object.

Objects of type PURCHASE_ORDER_T have two methods: RET_VALUE and TOTAL_VALUE. One is a MAP method, one of the two kinds of comparison methods. A MAP method returns the relative position of a given record within the order of records within the object. So, whenever Oracle needs to compare two PURCHASE_ORDER_T objects, it implicitly calls the RET_VALUE method to do so.

The two pragma declarations provide information to PL/SQL about what sort of access the two methods need to the database.

> **See Also:** *PL/SQL User's Guide and Reference* for complete details of how to use pragma declarations.

The statement does not include the actual PL/SQL programs implementing the methods RET_VALUE and TOTAL_VALUE. That appears in a later section.

The next statement completes the definition of STOCK_INFO_T, the last of the three incomplete object types declared at the beginning of this section.

```
CREATE TYPE stock_info_t AS OBJECT (
  stockno    NUMBER,
  cost       NUMBER,
  tax_code   NUMBER
  ) ;
```

Data units of type STOCK_INFO_T are objects representing the stock items that customers order. They have three numeric attributes.

Method definitions

This section shows how to specify the methods of the CUSTOMER_INFO_T and PURCHASE_ORDER_T object types.

```
CREATE OR REPLACE TYPE BODY purchase_order_t AS
  MEMBER FUNCTION total_value RETURN NUMBER IS
    i          INTEGER;
    stock      stock_info_t;
    line_item  line_item_t;
    total      NUMBER := 0;
    cost       NUMBER;

  BEGIN
    FOR i IN 1..SELF.line_item_list.COUNT  LOOP

      line_item := SELF.line_item_list(i);
      SELECT DEREF(line_item.stockref) INTO stock FROM DUAL ;

      total := total + line_item.quantity * stock.cost ;

      END LOOP;
    RETURN total;
  END;
```

```
  MAP MEMBER FUNCTION ret_value RETURN NUMBER IS
  BEGIN
    RETURN pono;
  END;
END;
```

The preceding statement defines the body of the PURCHASE_ORDER_T object type, that is, the PL/SQL programs that implement its methods.

The RET_VALUE Method

The RET_VALUE method is simple: you use it to return the number of its associated PURCHASE_ORDER_T object.

The TOTAL_VALUE Method

The TOTAL_VALUE method uses a number of O-R means to return the sum of the values of the line items of its associated PURCHASE_ORDER_T object:

- As already noted, the basic function of the TOTAL_VALUE method is to return the sum of the values of the line items of its associated PURCHASE_ORDER_T object. The keyword SELF, which is implicitly created as a parameter to every function, lets you refer to that object.
- The keyword COUNT gives the count of the number of elements in a PL/SQL table or array. Here, in combination with LOOP, the application iterates through all the elements in the collection — in this case, the items of the purchase order. In this way SELF.LINE_ITEM_LIST.COUNT counts the number of elements in the nested table that match the LINE_ITEM_LIST attribute of the PURCHASE_ORDER_T object, here represented by SELF.
- The DEREF operator takes a reference value as an argument, and returns a row object. In this case, DEREF (LINE_ITEM.STOCKREF) takes the STOCKREF attribute as an argument, and returns STOCK_INFO_T object. Looking back to our data definition, you will see that STOCKREF is an attribute of the LINE_ITEM_T object which is itself an element of the LINE_ITEM_LIST. This list object, which we have structured as a nested table, is in turn an attribute of the PURCHASE_ORDER_T object represented by SELF. This may seem rather complicated until you take it up again from a real-world perspective in which a purchase order (PURCHASE_ORDER_T) contains a list (LINE_ITEM_LIST) of items (LINE_ITEM_T), each of which contains a reference (STOCKREF) to information about the item (STOCK_INFO_T). The operation which we have been considering simply fetches the required data by O-R means.

- All these entities are abstract datatypes, and as such can be viewed as templates for objects — as PURCHASE_ORDER_T is a template for all purchase order objects. How then are to we retrieve the values of actual stock objects? The SQL SELECT statement with the explicit DEREF call is required, because Oracle does not support implicit dereferencing of REFs within PL/SQL programs. The PL/SQL variable STOCK is of type STOCK_INFO_T. The select statement sets it to the object represented by DEREF (LINE_ITEM.STOCKREF). And this object is the actual stock item referred to in the i-th line item
- Having retrieved the stock item in question, the next step is to compute its cost. The program refers to the stock item's cost as STOCK.COST, the COST attribute of the STOCK object. But to compute the cost of the item we also need to know the quantity of items ordered. In our application, the term LINE_ITEM.QUANTITY represents the QUANTITY attribute of each LINE_ITEM_T object.

The remainder of the method program is straightforward. The loop sums the extended values of the line items, and the method returns the total as its value.

The CUST_ORDER Method

The following statement defines the CUST_ORDER method of the CUSTOMER_INFO_T object type.

```
CREATE OR REPLACE TYPE BODY customer_info_t AS
  ORDER MEMBER FUNCTION
  cust_order (x IN customer_info_t) RETURN INTEGER IS
  BEGIN
    RETURN custno - x.custno;
  END;
END;
```

As mentioned earlier, the function of the CUST_ORDER operation is to compare information about two customer orders. The mechanics of the operation are quite simple. The order method CUST_ORDER takes another CUSTOMER_INFO_T object as an input argument and returns the difference of the two CUSTNO numbers. Since it subtracts the CUSTNO of the other CUSTOMER_INFO_T object from its own object's CUSTNO, the method returns (a) a negative number if its own object has a smaller value of CUSTNO, or (b) a positive number if its own has a larger value of CUSTNO, or (c) zero if the two objects have the same value of CUSTNO — in which case it is referring to itself! If CUSTNO has some meaning in the real world (e.g., lower numbers are created earlier in time than higher numbers), the actual value returned by this function could be useful.

This completes the definition of the user-defined types used in the purchase order application. Note that none of the declarations create tables or reserve data storage space.

Creating Object Tables

To this point the example is the same, whether you plan to create and populate object tables or implement the application with object views on top of the relational tables that appear in the first part of the example. The remainder of this chapter continues the example using object tables. Chapter 8, “Object Views—An Extended Example” picks up from this point and continues the example with object views.

Generally, you can think of the relationship between the "objects" and "tables" in the following way:

- classes, which represent entities, map to tables,
- attributes map to columns, and
- objects map to rows.

Viewed in way, each table is an implicit type whose objects (specific rows) each have the same attributes (the column values). The creation of explicit abstract datatypes and of object tables introduce a new level of functionality.

The Object Table CUSTOMER_TAB

Creating object tables: the basic syntax. The following statement defines an object table CUSTOMER_TAB to hold objects of type CUSTOMER_INFO_T.

```
CREATE TABLE customer_tab OF customer_info_t
 (custno PRIMARY KEY);
```

As you can see, there is a syntactic difference in the definition of object tables, namely the use of the term "OF". You may recall that we earlier defined the attributes of CUSTOMER_INFO_T objects as:

```
custno      NUMBER
custname    VARCHAR2(200)
address     address_t
phone_list  phone_list_t
po_list     po_reflist_t
```

This means that the table CUSTOMER_TAB has columns of CUSTNO, CUSTNAME, ADDRESS, PHONE_LIST and PO_LIST, and that each row is an object of type CUSTOMER_INFO_T. And, as you will see, this notion of *row object* offers a significant advance in functionality.

Abstract datatypes as a template for object tables. Note first that the fact that there is a type CUSTOMER_INFO_T means that you could create numerous tables of type CUSTOMER_INFO_T. For instance, you could create a table CUSTOMER_TAB2 also of type CUSTOMER_INFO_T. By contrast, without this ability, you would have to define each table individually.

Being able to create tables of the same type does not mean that you cannot introduce variations. Note that the statement by which we created CUSTOMER_TAB defined a primary key constraint on the CUSTNO column. This constraint applies only to this table. Another object table of CUSTOMER_INFO_T objects (e.g., CUSTOMER_TAB2) need not satisfy this constraint. This illustrates an important point: constraints apply to tables, not to type definitions.

Object tables with embedded objects. Examining the definition of CUSTOMER_TAB, you will see that the ADDRESS column contains ADDRESS_T objects. Put another way: an abstract datatype may have attributes that are themselves abstract datatypes. When these types are instantiated as objects, the included objects are instantiated at the same time (unless they allow for NULL values, in which case place-holders for their values are created). ADDRESS_T objects have attributes of built-in types which means that they are leaf-level scalar attributes of

CUSTOMER_INFO_T. Oracle creates columns for ADDRESS_T objects and their attributes in the object table CUSTOMER_TAB. You can refer to these columns using the dot notation. For example, if you wish to build an index on the ZIP column, you can refer to it as ADDRESS.ZIP.

The PHONE_LIST column contains VARRAYs of type PHONE_LIST_T. You may recall that we defined each object of type PHONE_LIST_T as a VARRAY of up to 10 telephone numbers, each represented by a data item of type VARCHAR2.

```
CREATE TYPE phone_list_t AS VARRAY(10) OF VARCHAR2(20) ;
```

Since each VARRAYs of type PHONE_LIST_T can contain no more than 200 characters (10 x 20), plus a small amount of overhead. Oracle stores the VARRAY as a single data unit in the PHONE_LIST column. Oracle stores VARRAYs that exceed 4000 bytes in BLOBs which means that they are stored outside the table. This raises an interesting question to which we will return: How does the DBMS reference these external objects?

The Object Table STOCK_TAB

The next statement creates an object table for STOCK_INFO_T objects:

```
CREATE TABLE stock_tab OF stock_info_t
 (stockno PRIMARY KEY) ;
```

This does not introduce anything new. The statement creates the STOCK_TAB object table. Since Each row of the table is a STOCK_INFO_T object having three numeric attributes:

```
  stockno    NUMBER,
  cost       NUMBER,
  tax_code   NUMBER
```

Oracle assigns a column for each attribute, and the CREATE TABLE statement places a primary key constraint on the STOCKNO column.

The Object Table PURCHASE_TAB

The next statement defines an object table for PURCHASE_ORDER_T objects:

```
CREATE TABLE purchase_tab OF purchase_order_t (
  PRIMARY KEY (pono),
  SCOPE FOR (custref) IS customer_tab
  )
  NESTED TABLE line_item_list STORE AS po_line_tab ;
```

The preceding statement creates the PURCHASE_TAB object table. Each row of the table is a PURCHASE_ORDER_T object. Attributes of PURCHASE_ORDER_T objects are:

```
pono           NUMBER
custref        REF customer_info_t
orderdate      DATE
shipdate       DATE
line_item_list line_item_list_t
shiptoaddr     address_t
```

The REF operator. The first new element introduced here has to do with the way the statement places a scope on the REFs in the CUSTREF column. When there is no restriction on scope (the default case), the REF operator allows you to reference any row object. However, these CUSTREF REFs can refer only to row objects in the CUSTOMER_TAB object table. The scope limitation applies only to CUSTREF columns of the CUSTOMER_TAB object table. It does not apply to the CUSTREF attributes of PURCHASE_ORDER_T objects that might be stored in any other object table.

Nested tables. A second new element has to do with the fact that each row has a nested table column LINE_ITEM_LIST. The last line of the statement creates the table PO_LINE_TAB to hold the LINE_ITEM_LIST columns of all of the rows of the PURCHASE_TAB table. Nested tables are particularly well-suited to coding master-detail relationships, such as we are considering in this purchase order example. As we will discuss, nested tables can also do much to remove the complexity of relational joins from applications.

All the rows of a nested table are stored in a separate storage table. A hidden column in the storage table, called the NESTED_TABLE_ID matches the rows with their corresponding parent row. All the elements in the nested table belonging to a particular parent have the same NESTED_TABLE_ID value.

For example, all the elements of the nested table of a given row of PURCHASE_TAB have the same value of NESTED_TABLE_ID. The nested table elements that belong to a different row of PURCHASE_TAB have a different value of NESTED_TABLE_ID.

The top level attributes of the nested table type map to columns in the storage table. A nested table whose elements are not of an object type has a single unnamed column. Oracle recognizes the keyword COLUMN_VALUE as representing the name of that column. For example, to place a scope on the REF column in a nested table of REFs, we can use the COLUMN_VALUE column name to refer to it.

Oracle creates columns in CUSTOMER_TAB for the remaining leaf level scalar attributes of PURCHASE_ORDER_T objects, namely, ORDERDATE, SHIPDATE, and the attributes of the ADDRESS_T object in SHIPTOADDR.

At this point all of the tables for the purchase order application are in place. The next section shows how to add additional specifications to these tables.

Altering the Tables

The next statement alters the PO_LINE_TAB storage table, which holds the LINE_ITEM_LIST nested table columns of the object table PURCHASE_TAB, to place a scope on the REFs it contains.

```
ALTER TABLE po_line_tab
  ADD (SCOPE FOR (stockref) IS stock_tab) ;
```

The PO_LINE_TAB storage table holds nested table columns of type LINE_ITEM_LIST_T. The definition of that type (from earlier in the chapter) is:

```
CREATE TYPE line_item_list_t AS TABLE OF line_item_t ;
```

An attribute of a LINE_ITEM_T object, and hence one column of the PO_LINE_TAB storage table, is STOCKREF, which is of type REF STOCK_INFO_T. The object table STOCK_TAB holds row objects of type STOCK_INFO_T. The alter statement restricts the scope of the REFs in the STOCKREF column to the object table STOCK_TAB.

The next statement further alters the PO_LINE_TAB storage table to specify its index storage.

```
ALTER TABLE po_line_tab
  STORAGE (NEXT 5K PCTINCREASE 5 MINEXTENTS 1 MAXEXTENTS 20) ;
```

The next statement creates an index on the PO_LINE_TAB storage table:

```
CREATE INDEX po_nested_in
  ON         po_line_tab (NESTED_TABLE_ID) ;
```

A storage table for a nested table column of an object table has a hidden column called NESTED_TABLE_ID. The preceding statement creates an index on that column, making access to the contents of LINE_ITEM_LIST columns of the PURCHASE_TAB object table more efficient.

The next statement shows how to use NESTED_TABLE_ID to enforce uniqueness of a column of a nested table within each row of the enclosing table. It creates a

unique index on the PO_LINE_TAB storage table. That table holds the LINE_ITEM_LIST columns of all of the rows of the PURCHASE_TAB table.

```
CREATE UNIQUE INDEX po_nested
  ON                po_line_tab (NESTED_TABLE_ID, lineitemno) ;
```

By including the LINEITEMNO column in the index key and specifying a unique index, the statement ensures that the LINEITEMNO column contains distinct values within each purchase order.

Inserting Values

The statements in this section show how to insert the same data into the object tables just created as the statements on page 7-5 insert into the relational tables of the first part of the example.

stock_tab

```
INSERT INTO stock_tab VALUES(1004, 6750.00, 2);
INSERT INTO stock_tab VALUES(1011, 4500.23, 2);
INSERT INTO stock_tab VALUES(1534, 2234.00, 2);
INSERT INTO stock_tab VALUES(1535, 3456.23, 2);
```

customer_tab

```
INSERT INTO customer_tab
  VALUES (
    1, 'Jean Nance',
    address_t('2 Avocet Drive', 'Redwood Shores', 'CA', '95054'),
    phone_list_t('415-555-1212')
    ) ;

INSERT INTO customer_tab
  VALUES (
    2, 'John Nike',
    address_t('323 College Drive', 'Edison', 'NJ', '08820'),
    phone_list_t('609-555-1212','201-555-1212')
    ) ;
```

purchase_tab

```
INSERT INTO purchase_tab
  SELECT  1001, REF(C),
          SYSDATE,'10-MAY-1997',
          line_item_list_t(),
          NULL
```

```
   FROM   customer_tab C
   WHERE  C.custno = 1 ;
```

The preceding statement constructs a PURCHASE_ORDER_T object with the following attributes:

```
  pono             1001
  custref          REF to customer number 1
  orderdate        SYSDATE
  shipdate         10-MAY-1997
  line_item_list   an empty line_item_list_t
  shiptoaddr       NULL
```

The statement uses a query to construct a REF to the row object in the CUSTOMER_TAB object table that has a CUSTNO value of 1.

The next statement uses a flattened subquery, signaled by the keyword THE, to identify the target of the insertion, namely the nested table in the LINE_ITEM_LIST column of the row object in the PURCHASE_TAB object table that has a PONO value of 1001.

```
INSERT INTO THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 1001
  )
  SELECT  01, REF(S), 12, 0
   FROM   stock_tab S
   WHERE  S.stockno = 1534;
```

The preceding statement inserts a line item into the nested table identified by the flattened subquery. The line item that it inserts contains a REF to the row object in the object table STOCK_TAB that has a STOCKNO value of 1534.

The following statements are similar to the preceding two.

```
INSERT INTO purchase_tab
  SELECT  2001, REF(C),
          SYSDATE,'20-MAY-1997',
          line_item_list_t(),
          address_t('55 Madison Ave','Madison','WI','53715')
   FROM   customer_tab C
   WHERE  C.custno = 2;

INSERT INTO THE (
  SELECT  P.line_item_list
```

```
   FROM   purchase_tab P
   WHERE  P.pono = 1001
  )
  SELECT  02, REF(S), 10, 10
   FROM   stock_tab S
   WHERE  S.stockno = 1535;

INSERT INTO THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 2001
  )
  SELECT  10, REF(S), 1, 0
   FROM   stock_tab S
   WHERE  S.stockno = 1004;

INSERT INTO THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 2001
  )
  VALUES( line_item_t(11, NULL, 2, 1) ) ;
```

The next statement uses a table alias to refer to the result of the flattened subquery

```
UPDATE THE (
  SELECT  P.line_item_list
   FROM   purchase_tab P
   WHERE  P.pono = 2001
  ) plist

  SET plist.stockref =
   (SELECT REF(S)
     FROM  stock_tab S
     WHERE S.stockno = 1011
     )

  WHERE plist.lineitemno = 11 ;
```

Selecting

The following query statement implicitly invokes a comparison method. It shows how Oracle uses the ordering of PURCHASE_ORDER_T object types that the comparison method defines:

```
SELECT  p.pono
 FROM   purchase_tab p
 ORDER BY VALUE(p);
```

The preceding instruction causes Oracle to invoke the map method RET_VALUE for each PURCHASE_ORDER_T object in the selection. Since that method simply returns the value of the object's PONO attribute, the result of the selection is a list of purchase order numbers in ascending numerical order.

The following queries correspond to the queries in "Selecting" on page 7-6.

Customer and Line Item Data for Purchase Order 1001

```
SELECT  DEREF(p.custref), p.shiptoaddr, p.pono,
        p.orderdate, line_item_list

 FROM   purchase_tab p

 WHERE  p.pono = 1001 ;
```

Total Value of Each Purchase Order

```
SELECT   p.pono, p.total_value()

 FROM    purchase_tab p ;
```

Purchase Order and Line Item Data Involving Stock Item 1004

```
SELECT   po.pono, po.custref.custno,

         CURSOR (
           SELECT  *
            FROM   TABLE (po.line_item_list) L
            WHERE  L.stockref.stockno = 1004
           )

 FROM    purchase_tab po ;
```

Deleting

The following example has the same effect as the two deletions needed in the relational case (see "Deleting" on page 7-7). In this case Oracle automatically deletes all line items belonging to the deleted purchase order. The relational case requires a separate step.

Delete Purchase Order 1001

```
DELETE
 FROM   purchase_order
 WHERE  pono = 1001 ;
```

This concludes the object table version of the purchase order example. The next chapter develops an alternative version of the example using relational tables and object views.

8

Object Views—An Extended Example

This chapter contains an extended example of how to use object views. The chapter has the following major sections:

- Introduction
- Purchase Order Example

Introduction

Object views are virtual object tables, materialized out of data from tables or views.

> **See Also:** For a discussion of object views and how to use them, see *Oracle8 Concepts*.

The example in this chapter illustrates the most important aspects of defining and using object views. The definitions of triggers use the PL/SQL language. The remainder of the example uses Oracle SQL.

> **See Also:** *Oracle8 SQL Reference* for a complete description of SQL syntax and usage.

PL/SQL provides additional capabilities beyond those illustrated here, especially in the area of accessing and manipulating the elements of collections.

> **See Also:** *PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities.

Client applications that use the Oracle call interface (OCI) can take advantage of its extensive facilities for accessing the objects and collections defined by object views and manipulating them on the client side.

> **See Also:** *Programmer's Guide to the Oracle Call Interface* for a complete discussion of those facilities.

Purchase Order Example

Chapter 7, "User-Defined Datatypes — An Extended Example" develops a purchase order example by following these steps:

1. Establish the entities and relationships.
2. Implement the entity-relationship structure by creating and populating relational tables.
3. Define an object-relational schema of user-defined types to model the entity-relationship structure.
4. Implement the entity-relationship structure using the object-relational schema to create and populate object tables.

The approach in this chapter uses the same initial steps but a different final step. Rather than creating and populating object tables, this approach uses object views to materialize virtual object tables out of data in the relational tables.

Defining Object Views

The example developed in Chapter 7 contains three object tables: CUSTOMER_TAB, STOCK_TAB, and PURCHASE_TAB. this chapter contains three corresponding object views: CUSTOMER_VIEW, STOCK_VIEW, and PURCHASE_VIEW.

The statement that creates an object view has four parts:

- The name of the view.
- The name of the object type it is based on.
- The source of the primary-key-based object identifier.
- A selection that populates the virtual object table corresponding to the object type.

The customer_view View

The definition of the CUSTOMER_INFO_T object type appears on page 10. This object view is based on that object type.

```
CREATE OR REPLACE VIEW
customer_view OF customer_info_t WITH OBJECT OID(custno) AS
  SELECT  C.custno, C.custname,
          address_t(C.street, C.city, C.state, C.zip),
          phone_list_t (C.phone1, C.phone2, C.phone3)
   FROM   customer_info C ;
```

This object view selects its data from the CUSTOMER_INFO table. The definition of this table appears on page 4.

The CUSTOMER_INFO_T object type has the following attributes:

```
custno      NUMBER
custname    VARCHAR2(200)
address     address_t
phone_list  phone_list_t
```

The object view definition takes the CUSTNO and CUSTNAME attributes from correspondingly named columns of the CUSTOMER_INFO table. It uses the STREET, CITY, STATE, and ZIP columns of the CUSTOMER_INFO table as arguments to the constructor function for the ADDRESS_T object type, which is defined on page 10.

The stock_view View

The definition of the STOCK_INFO_T object type appears on page 13. This object view is based on that object type.

```
CREATE OR REPLACE VIEW
stock_view OF stock_info_t WITH OBJECT OID(stockno) AS
  SELECT  *
   FROM   stock_info ;
```

This object view selects its data from the STOCK_INFO table. The definition of this table appears on page 5.

The selection used to materialize the object view is extremely simple, because the object type definition and the table definition correspond exactly.

The purchase_view View

The definition of the PURCHASE_ORDER_T object type appears on page 12. This object view is based on that object type.

```
CREATE OR REPLACE VIEW
purchase_view OF purchase_order_t WITH OBJECT OID (pono) AS
  SELECT  P.pono,
          MAKE_REF (customer_view, P.custno),
          P.orderdate, P.shiptodate,
          CAST (
            MULTISET (
              SELECT  line_item_t (
                        L.lineitemno,
                        MAKE_REF(stock_view, L.stockno),
                        L.quantity, L.discount
                        )
               FROM   line_items L
               WHERE  L.pono= P.pono
              )
             AS line_item_list_t
             ),
           address_t (P.shiptostreet, P.shiptocity,
                      P.shiptostate,  P.shiptozip)
   FROM   purchase_order P ;
```

This object view is based on the LINE_ITEMS table, which is defined on page 5, the PURCHASE_ORDER table, which is defined on page 4, and the CUSTOMER_VIEW and STOCK_VIEW object views defined in the two previous sections.

The PURCHASE_ORDER_T object type has the following attributes:

```
pono            NUMBER
custref         REF customer_info_t
orderdate       DATE
shipdate        DATE
```

```
line_item_list line_item_list_t
shiptoaddr     address_t
```

The object view definition takes its PONO column from the PONO column of the PURCHASE_ORDER table. It uses the expression MAKE_REF (CUSTOMER_VIEW, CUSTNO) to create a REF to the row object in the customer_view object view identified by CUSTNO. That REF becomes the CUSTREF column.

The object view definition takes its ORDERDATE and SHIPDATE columns from the ORDERDATE and SHIPTODATE columns of the PURCHASE_ORDER table.

The object view definition uses the term

```
CAST (
  MULTISET (
    SELECT  line_item_t (
              L.lineitemno,
              MAKE_REF(stock_view, L.stockno),
              L.quantity, L.discount
              )
     FROM   line_items L
     WHERE  L.pono= P.pono
    )
  AS line_item_list_t
  ),
```

to materialize the LINE_ITEM_LIST column of the object view. At the innermost level of this expression, the operator MAKE_REF(STOCK_VIEW, STOCKNO) builds a REF to the row object in the STOCK_VIEW object view identified by STOCKNO. That REF becomes one of the input arguments to the constructor function for the LINE_ITEM_T object type. The other arguments come from the LINEITEMNO, QUANTITY, and DISCOUNT columns of the LINE_ITEMS table.

The selection results in a set of LINE_ITEM_T objects, one for each row of the LINE_ITEMS table whose PONO column matches the PONO column of the row of the PURCHASE_ORDER table that is currently being examined in the outer selection. The MULTISET operator tells Oracle to regard the set of LINE_ITEM_T objects as a multiset, making it an appropriate argument for the CAST operator, which turns it into a nested table of type LINE_ITEM_LIST_T, as specified by the AS clause.

The resulting nested table becomes the LINE_ITEM_LIST column of the object view.

Finally, the definition uses the SHIPTOSTREET, SHIPTOCITY, SHIPTOSTATE, and SHIPTOZIP columns of the PURCHASE_ORDER table as arguments to the construc-

tor function for the ADDRESS_T object type to materialize the SHIPTOADDR column of the object view.

Updating the Object Views

Oracle provides INSTEAD OF triggers as a way to update complex object views. This section presents the INSTEAD OF triggers necessary to update the object views just defined.

Oracle invokes an object view's INSTEAD OF trigger whenever a command directs it to change the value of any attribute of a row object in the view. Oracle makes both the current value and the requested new value of the row object available to the trigger program. It recognizes the keywords :OLD and :NEW as representing the current and new values.

INSTEAD OF Trigger for purchase_view

```
CREATE OR REPLACE TRIGGER
poview_insert_tr INSTEAD OF INSERT ON purchase_view

DECLARE
 line_itms     line_item_list_t ;
 i             INTEGER ;
 custvar       customer_info_t ;
 stockvar      stock_info_t ;
 stockvartemp  REF stock_info_t ;

BEGIN
 line_itms := :NEW.line_item_list ;

 SELECT DEREF(:NEW.custref) INTO custvar FROM DUAL ;

 INSERT INTO purchase_order VALUES (
   :NEW.pono, custvar.custno, :NEW.orderdate, :NEW.shipdate,
   :NEW.shiptoaddr.street,    :NEW.shiptoaddr.city,
   :NEW.shiptoaddr.state,     :NEW.shiptoaddr.zip ) ;

 FOR i IN 1..line_itms.COUNT LOOP
   stockvartemp := line_itms(i).stockref ;
   SELECT DEREF(stockvartemp) INTO stockvar FROM DUAL ;

   INSERT INTO line_items VALUES (
     line_itms(i).lineitemno, :NEW.pono, stockvar.stockno,
     line_itms(i).quantity,    line_itms(i).discount ) ;
```

```
 END LOOP ;

END ;
```

This trigger program inserts new values into the PURCHASE_ORDER table. Then, in a loop, it inserts new values into the LINE_ITEMS table for each LINE_ITEM_T object in the nested table in the new LINE_ITEM_LIST column.

The use of the STOCKVARTEMP variable is an alternative to implicitly dereferencing the REF represented by LINE_ITMS(i).STOCKREF.

INSTEAD OF Trigger for customer_view

```
CREATE OR REPLACE TRIGGER
custview_insert_tr INSTEAD OF INSERT ON customer_view

DECLARE
  phones  phone_list_t;
  tphone1 customer_info.phone1%TYPE := NULL;
  tphone2 customer_info.phone2%TYPE := NULL;
  tphone3 customer_info.phone3%TYPE := NULL;
BEGIN
  phones := :NEW.phone_list;

  IF phones.COUNT > 2 THEN
    tphone3 := phones(3);
  END IF;

  IF phones.COUNT > 1 THEN
    tphone2 := phones(2);
  END IF;

  IF phones.COUNT > 0 THEN
    tphone1 := phones(1);
  END IF;

  INSERT INTO customer_info VALUES (
   :NEW.custno,       :NEW.custname,      :NEW.address.street,
   :NEW.address.city, :NEW.address.state, :NEW.address.zip,
    tphone1,           tphone2,            tphone3);
END ;
```

This trigger function updates the CUSTOMER_INFO table with the new information. Most of the program deals with updating the three phone number columns of the customer table from the :NEW.PHONE_LIST VARRAY of phone numbers. The IF

statements assure that the program does not attempt to access :NEW.PHONE_LIST elements with indexes greater than :NEW.PHONE_LIST.COUNT.

There is a slight mismatch between these two representations, because the VARRAY is defined hold up to 10 numbers, while the customer table has only three phone number columns. The trigger program discards :NEW.PHONE_LIST elements with indexes greater than 3.

INSTEAD OF Trigger for stock_view

```
CREATE OR REPLACE TRIGGER
stockview_insert_tr INSTEAD OF INSERT ON stock_view

BEGIN
  INSERT INTO stock_info VALUES (
    :NEW.stockno, :NEW.cost, :NEW.tax_code );
END ;
```

This trigger function updates the STOCK_INFO table with the new information.

Sample Updates

The following statement fires the CUSTOMER_VIEW trigger.

```
INSERT INTO customer_view VALUES (
   13, 'Ellan White',
   address_t('25 I Street', 'Memphis', 'TN', '05456'),
              phone_list_t('615-555-1212') );
```

The preceding statement inserts a new customer into the database via the CUSTOMER_VIEW object view.

The following statement fires the PURCHASE_VIEW trigger.

```
INSERT INTO purchase_view
  SELECT  3001, REF(c), SYSDATE, SYSDATE,
          CAST(
            MULTISET(
               SELECT line_item_t(41, REF(S), 20, 1)
                FROM  stock_view S
                WHERE S.stockno = 1535
                )
            AS line_item_list_t
            ),
          address_t('22 Nothingame Ave','Cockstown','AZ','44045')
```

```
FROM   customer_view c
WHERE  c.custno = 1
```

The preceding statement inserts a new purchase order into the database via the PURCHASE_VIEW object view. Customer number 1 has ordered 20 of stock item 1535. The statement assigns number 3001 to the purchase order and number 41 to the line item.

Selecting

The three queries in “Selecting” on page 7-23 work exactly as written, but with the object table name PURCHASE_TAB replaced by the object view name PURCHASE_VIEW. Queries involving other object tables work with the analogous name replacement.

9

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
- Referential Integrity in a Distributed Database
- Using CHECK Integrity Constraints
- Defining Integrity Constraints
- Enabling and Disabling Integrity Constraints
- Altering Integrity Constraints
- Dropping Integrity Constraints
- Managing FOREIGN KEY Integrity Constraints
- Listing Integrity Constraint Definitions

> **See Also:** *Trusted Oracle* documentation 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

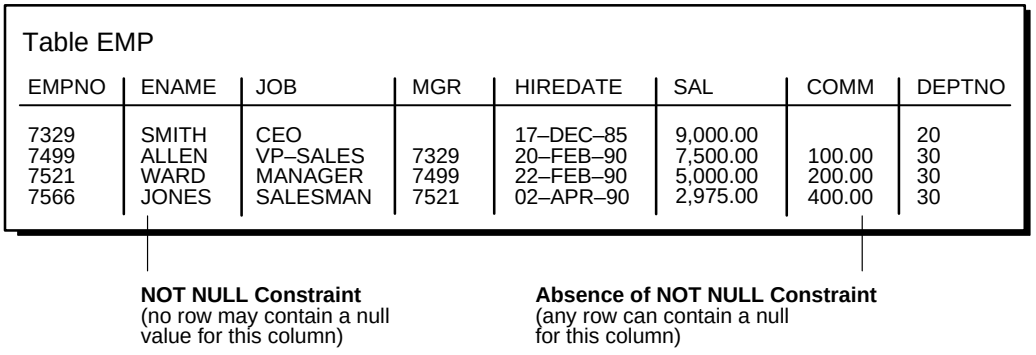
By default, all columns can contain nulls. Only define `NOT NULL` constraints for columns of a table that absolutely require values at all times.

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 Also: "Relationships Between Parent and Child Tables" on page 9-9.

Figure 9–1 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

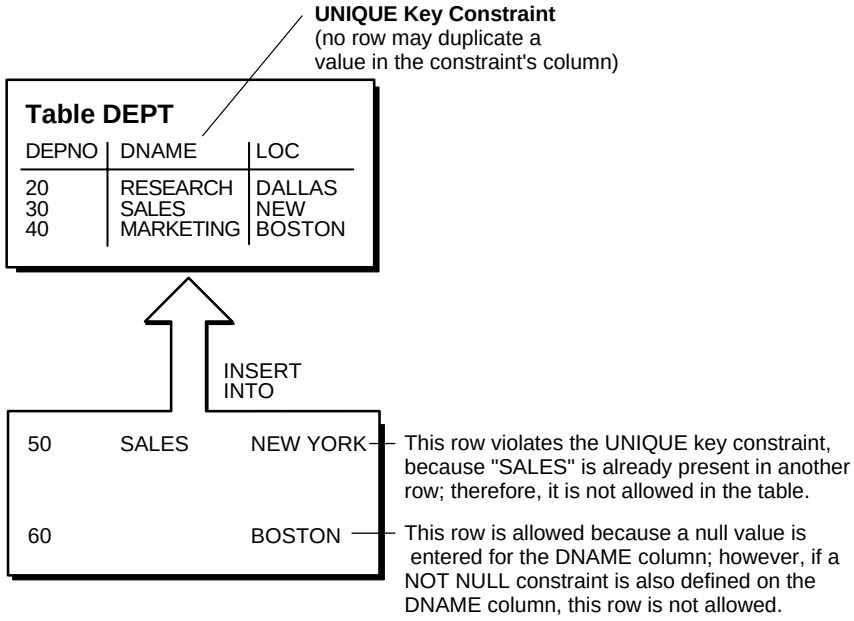
Only assign default values to columns that contain a typical value. For example, in the DEPT table, if most departments are located at one site, the default value for the LOC column can be set to this value (such as NEW YORK).

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 9–2 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 9-9.

Figure 9–3 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

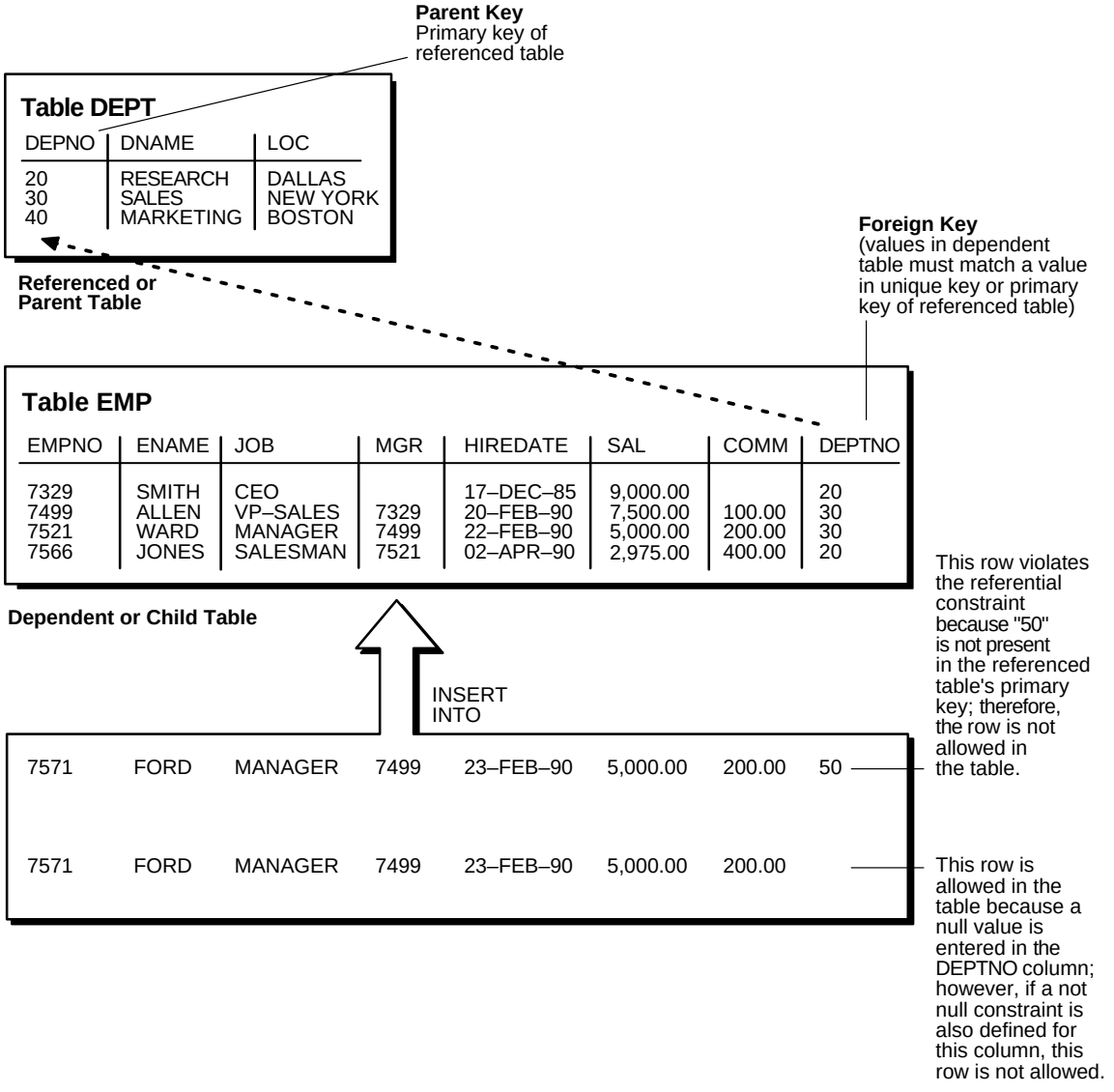
By default (that is, without any NOT NULL or CHECK clauses), and in accordance with the ANSI/ISO standard, the FOREIGN KEY constraint enforces the "match none" rule for composite foreign keys. The "full" and "partial" rules can also be enforced by using CHECK and NOT NULL constraints, as follows:

- 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 13, "Using Database Triggers".

Figure 9–3 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 9–3 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.

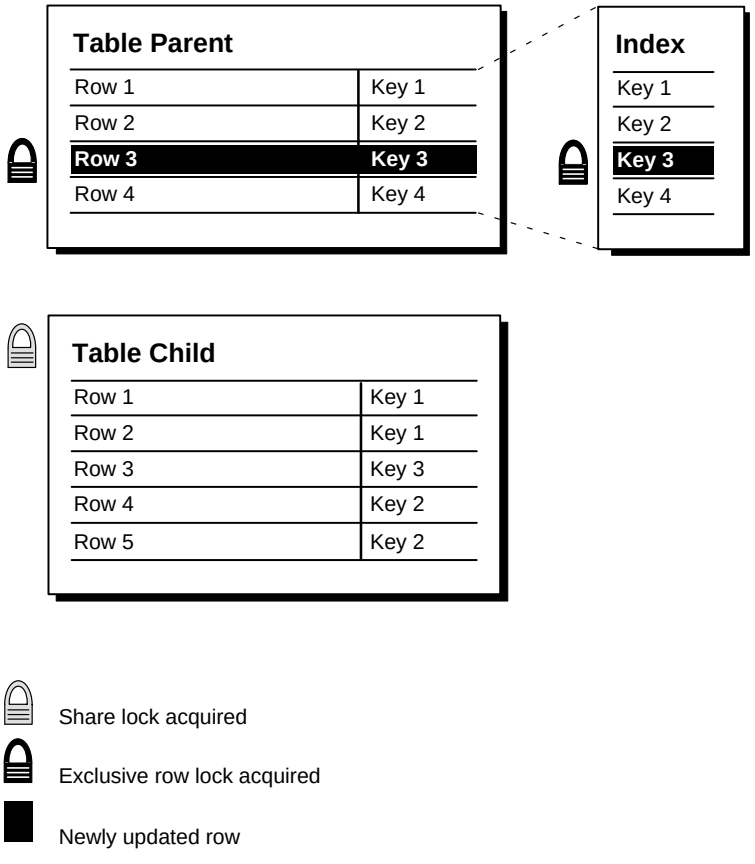
No Index on the Foreign Key Figure 9–4 illustrates the locking mechanisms used by Oracle when no index is defined on the foreign key and when rows are being updated or deleted in the parent table. Inserts into the parent table do not require any locks on the child table.

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 9–4 Locking Mechanisms When No Index Is Defined on the Foreign Key

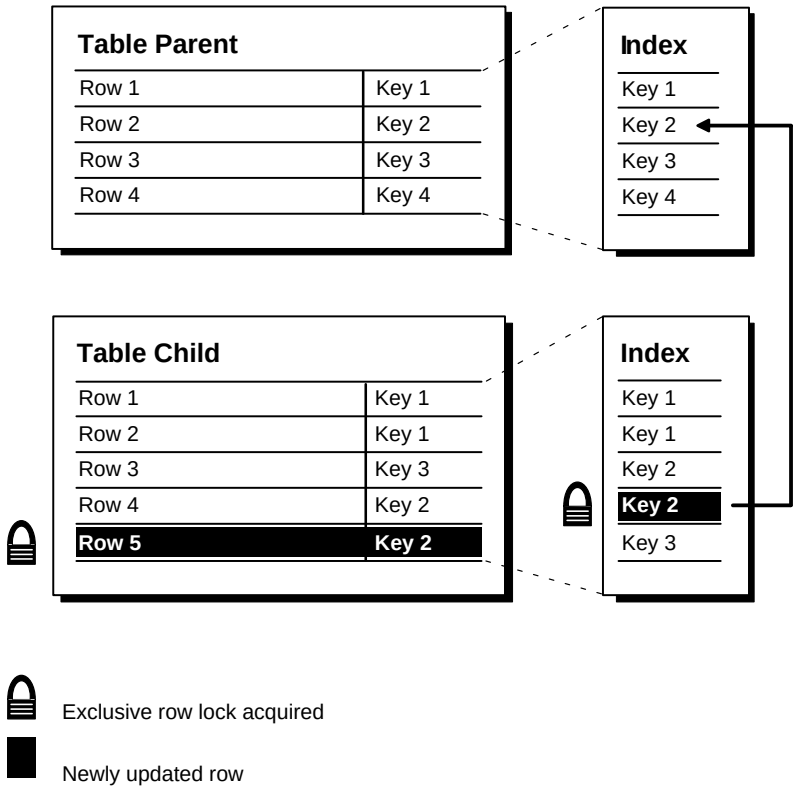


Index on the Foreign Key Figure 9–5 illustrates the locking mechanisms used by Oracle when an index is defined on the foreign key, and new rows are inserted, updated or deleted in the child table.

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 9–5 Locking Mechanisms 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 13, "Using Database Triggers". Using triggers to maintain referential integrity requires the distributed option; for more information refer to *Oracle8 Distributed Database Systems*

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 Also:** "CHECK and NOT NULL Integrity Constraints" on page 9-15.

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 Also: *Oracle8 SQL Reference* 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 *Oracle8 SQL Reference*.

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.

Note: There are additional considerations if you are using Trusted Oracle; see the *Trusted Oracle* 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 9–1 lists each type of constraint and the associated restrictions with the ALTER TABLE command.

Table 9–1 Restrictions for Defining Integrity Constraints 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

** 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 sys-

tem privilege. FOREIGN KEY integrity constraints also require some additional privileges.

> **See Also:** "Privileges Required for FOREIGN KEY Integrity Constraints" on page 9-26 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.

> **See Also:** "Listing Integrity Constraint Definitions" on page 9-27 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.

> **See Also:** "Enabling and Disabling Key Integrity Constraints" on page 9-22 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 information about defining and managing FOREIGN KEY constraints

> **See Also:** "Managing FOREIGN KEY Integrity Constraints" on page 9-25. UNIQUE key and PRIMARY KEY constraints are usually enabled by the database administrator, and the *Oracle8 Administrator's Guide* .

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 9-23.

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 9-23 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 9-23 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 9-27 and *Oracle8 Reference*.

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. UNIQUE key and PRIMARY KEY constraints are usually managed by the database administrator.

See Also: "Managing FOREIGN KEY Integrity Constraints" on page 9-25., and the *Oracle8 Administrator's Guide*.

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 Also:** *Oracle8 Tuning* 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 tabled 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
-----------------      ------ ------------   -----------
AAAA5bAADAAAAEQAAA     SCOTT  DEPT           SYS_C00301
AAAA5bAADAAAAEQAAB     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. UNIQUE key and PRIMARY KEY constraints are usually managed by the database administrator.

> **See Also:** "Managing FOREIGN KEY Integrity Constraints" on page 9-25., and the *Oracle8 Administrator's Guide*.

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 *Oracle8 Reference* 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.

Constraint Type	Character
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, "Managing Schema Objects" 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 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
```

10

Using Procedures and Packages

This chapter discusses the procedural capabilities of Oracle, including:

- PL/SQL Procedures and Packages
- PL/SQL Packages
- Remote Dependencies
- Cursor Variables
- Hiding PL/SQL Code
- Error Handling
- Invoking Stored Procedures
- Calling Stored Functions from SQL Expressions
- Supplied Packages
- Describing Stored Procedures
- Listing Information about Procedures and Packages
- The DBMS_ROWID Package
- The UTL_HTTP Package

Note: If you are using Trusted Oracle, also see the *Trusted Oracle* documentation for additional information.

PL/SQL Procedures and Packages

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
- stand-alone stored procedures and functions
- packages, that can contain stored procedures and functions

See Also: 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 12-22):

```
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 Also:** *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 Also:** A description of cursor variables on page 10-25.

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 13, "Using Database Triggers" in this Guide for more information.

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 *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 10-3:

```
PROCEDURE get_emp_names (dept_num IN NUMBER) IS
    emp_name       VARCHAR2(10);
    CURSOR         c1 (depno NUMBER) IS
                      SELECT ename FROM emp
                        WHERE deptno = depno;

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 10–1 summarizes the information about parameter modes. Parameter modes are explained in detail in the *PL/SQL User's Guide and Reference*.

Table 10–1 Parameter Modes

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

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

> **Note:** 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 in "Procedure Parameters" on page 10-5 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, which 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 10-5 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 10-8, 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 from the *get_emp.sql* file (*.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.

> **WARNING: 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, but note that this is done without 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 *Oracle8 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.

> **Note:** 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.

WARNING: The procedure or function is dropped without any warning.

External Procedures

A PL/SQL procedure executing on an Oracle Server can call an external procedure, written in a 3GL. The 3GL procedure executes in a separate address space from that of the Oracle Server.

See Also: For information about external procedures, see the *PL/SQL User's Guide and Reference*.

PL/SQL 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 10-14.

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 10-10.

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 stand-alone procedure, a stand-alone 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.

Remote Dependencies

Dependencies among PL/SQL *library units* (packages, stored procedures, and stored functions) can be handled in two ways:

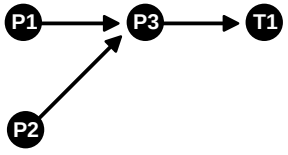
- timestamps
- signatures

Timestamps

If timestamps are used to handle dependencies among PL/SQL library units, whenever you alter a library unit or a relevant schema object all of its dependent units are marked as invalid and must be recompiled before they can be executed.

Each library unit carries a timestamp that is set by the server when the unit is created or recompiled. Figure 10–1 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 10–1 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, Oracle provides 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 "Controlling Remote Dependencies" on page 10-23 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 DBlink 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 10–2 shows the classes of types.

Table 10–2 Datatype Classes

Varchar Types:	**Number Types:**
VARCHAR2	NUMBER
VARCHAR	INTEGER
STRING	INT
LONG	SMALLINT
ROWID	DECIMAL
	DEC
Character Types:	REAL
CHARACTER	FLOAT
CHAR	NUMERIC
	DOUBLE PRECISION
Raw Types:	NUMERIC
RAW	
LONG RAW	
Integer Types:	**Date Type:**
BINARY_INTEGER	DATE
PLS_INTEGER	
BOOLEAN	**MLS Label Type:**
NATURAL	MLSLABEL
POSITIVE	
POSITIVEN	
NATURALN	

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 of Signatures

In the `GET_EMP_NAME` procedure defined on page 10-5, 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 10-22:

```
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.

- 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 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 by comparing timestamps at runtime. If 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 10-19, 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 the timestamp dependency mode.
- 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.

You can also 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 of Cursor Variables

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:

- *Pro*C/C++ Precompiler Programmer's Guide*
- *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=ouput_file]
```

> **See Also:** 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.

> **Note:** 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 "Handling Errors in Remote Procedures" on page 10-33.

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:

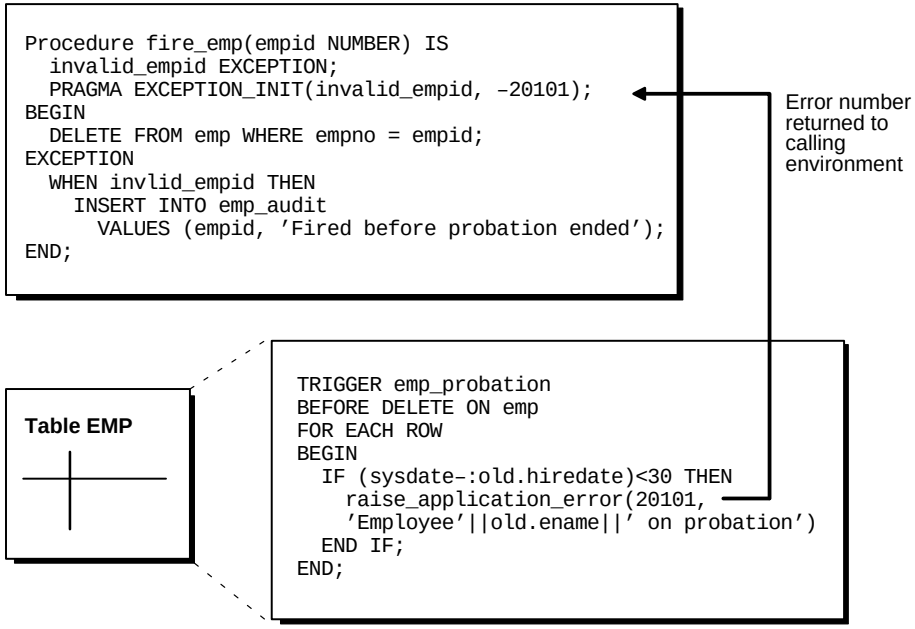
- Issue a RAISE statement that names the appropriate exception. A RAISE statement stops the execution of the procedure and control passes to an exception handler (if any).
- Call the RAISE_APPLICATION_ERROR procedure to return a user-specified error number and message.

You can also define an exception handler to handle user-specified error messages. For example, Figure 10–2 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 (stand-alone or package).

Figure 10–2 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" on page 10-33.

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 stand-alone (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.

Note: Before issuing the SHOW ERRORS command, use the SET CHARWIDTH command to get long lines on output. The value 132 is usually a good choice:

```
SET CHARWIDTH 132
```

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 WHER 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
-------------- -----------------------------------------
3/27           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 Also:** *Oracle8 Reference* 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 "Output from Stored Procedures and Triggers" on page 12-22 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. For more information, see "Calling Stored Functions from SQL Expressions" on page 10-42.

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:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide,*
- *SQL*Module for Ada Programmer's Guide*

Name Resolution When Invoking Procedures

References to procedures and packages are resolved according to the algorithm described in "Name Resolution in SQL Statements" on page 4-45.

Privileges Required to Execute a Procedure

If you are the owner of a stand-alone procedure or package, you can execute the stand-alone 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 stand-alone or packaged procedure owned by another user, the following conditions apply:

- You must have the EXECUTE privilege for the stand-alone 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);
```

> **Note:** 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 1subprogram, 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);
```

> **See Also:** For information on exception handling when calling remote procedures, see page 10-33.

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 (stand-alone 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:** U**nlike stored procedures, which use compile-time binding, runtime binding is used when referencing remote procedures. 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 stand-alone procedures and packages to

- hide the identity of the name and owner of a procedure or package

- provide location transparency for remotely stored procedures (stand-alone 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

You cannot call stored PL/SQL functions from a CHECK constraint clause of a CREATE or ALTER TABLE command or use them 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;
```

These sample calls to PL/SQL functions 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 stand-alone 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)
RNDS	means "reads no database state" (does not query database tables)
WNPS	means "writes no package state" (does not change the values of packaged variables)
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 example, 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 that 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 stand-alone) 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);
    ...
```

Serially Reusable PL/SQL Packages

PL/SQL packages normally consume user global area (UGA) memory corresponding to the number of package variables and cursors in the package. This limits scalability because the memory increases linearly with the number of users. The solution is to allow some packages to be marked as SERIALLY_REUSABLE (using pragma syntax).

For serially reusable packages, the package global memory is not kept in the UGA per user, but instead it is kept in a small pool and reused for different users. This means that the global memory for such a package is only used within a unit of work. At the end of that unit of work, the memory can therefore be released to the pool to be reused by another user (after running the initialization code for all the global variables).

The unit of work for serially reusable packages is implicitly a CALL to the server, for example, an OCI call to the server, or a PL/SQL client-to-server RPC call or server-to-server RPC call.

Package States

The state of a nonreusable package (one not marked SERIALLY_REUSABLE) persists for the lifetime of a session. A package's *state* includes global variables, cursors, and so on.

The state of a serially reusable package persists only for the lifetime of a CALL to the server. On a subsequent call to the server, if a reference is made to the serially reusable package, Oracle creates a new *instantiation* (described below) of the serially reusable package and initializes all the global variables to NULL or to the default values provided. Any changes made to the serially reusable package state in the previous CALLs to the server are not visible.

Note: Creating a new instantiation of a serially reusable package on a CALL to the server does not necessarily imply that Oracle allocates memory or configures the instantiation object. Oracle simply looks for an available instantiation work area (which is allocated and configured) for this package in a least-recently used (LRU) pool in SGA. At the end of the CALL to the server this work area is returned back to the LRU pool. The reason for keeping the pool in the SGA is that the work area can be reused across users who have requests for the same package.

Why Serially Reusable Packages?

Since the state of a non-reusable package persists for the lifetime of the session, this locks up UGA memory for the whole session. In applications such as Oracle Office a log-on session can typically exist for days together. Applications often need to use certain packages only for certain localized periods in the session and would ideally like to de-instantiate the package state in the middle of the session once they are done using the package.

With SERIALLY_REUSABLE packages the application developers have a way of modelling their applications to manage their memory better for scalability. Package state that they care about only for the duration of a CALL to the server should be captured in SERIALLY_REUSABLE packages.

Syntax

A package can be marked serially reusable by a pragma. The syntax of the pragma is:

```
PRAGMA SERIALLY_REUSABLE;
```

A package specification can be marked serially reusable whether or not it has a corresponding package body. If the package has a body, the body must have the serially reusable pragma if its corresponding specification has the pragma; and it cannot have the serially reusable pragma unless the specification also has the pragma.

Semantics

A package that is marked SERIALLY_REUSABLE has the following properties:

- Its package variables are meant for use only within the WORK boundaries, which correspond to CALLs to the server (either OCI call boundaries or PL/SQL RPC calls to the server).

  Note: If the application programmer makes a mistake and depends on a package variable that is set in a previous unit of work, the application program can fail. PL/SQL cannot check for such cases.

- A pool of package instantiations is kept and whenever a "unit of work" needs this package, one of the instantiations is "reused" as follows:
  - The package variables are reinitialized (for example, if the package variables have default values, then those values are reinitialized).
  - The initialization code in the package body is executed again.
- At the "end work" boundary, cleanup is done.
  - If any cursors were left open, they are silently closed.
  - Some non-reusable secondary memory is freed (such as memory for collection variables or long VARCHAR2s).
  - This package instantiation is returned back to the pool of reusable instantiations kept for this package.
- Serially reusable packages cannot be accessed from within triggers. If you attempt to access a serially reusable package from a trigger, Oracle issues the

error message "cannot access Serially Reusable package <*string*> in the context of a trigger."

Example 1

This example has a serially reusable package specification (there is no body). It demonstrates how package variables behave across CALL boundaries.

```
connect scott/tiger;

create or replace package SR_PKG is
  pragma SERIALLY_REUSABLE;
  n number := 5;                -- default initialization
end SR_PKG;
/
```

Suppose your Enterprise Manager (or SQL*Plus) application issues the following:

```
connect scott/tiger

# first CALL to server
begin
 SR_PKG.n := 10;
end;
/

# second CALL to server
begin
 dbms_output.put_line(SR_PKG.n);
end;
/
```

The above program will print:

```
5
```

> **Note:** If the package had not had the pragma SERIALLY_REUSABLE, the program would have printed '10'.

Example 2

This example has both a package specification and body which are serially reusable. Like Example 1, this example demonstrates how the package variables behave across CALL boundaries.

```
SQL> connect scott/tiger;
```

```
Connected.
SQL>
SQL> drop package SR_PKG;
Statement processed.
SQL>
SQL> create or replace package SR_PKG is
     2>
     3>  pragma SERIALLY_REUSABLE;
     4>
     5>  type str_table_type is table of varchar2(200) index by binary_integer;
     6>
     7>  num     number        := 10;
     8>  str     varchar2(200) := 'default-init-str';
     9>  str_tab str_table_type;
    10>
    11>  procedure print_pkg;
    12>  procedure init_and_print_pkg(n number, v varchar2);
    13>
    14> end SR_PKG;
    15> /
Statement processed.
SQL>
SQL>
SQL> create or replace package body SR_PKG is
     2>
     3>  -- the body is required to have the pragma since the
     4>  -- specification of this package has the pragma
     5>  pragma SERIALLY_REUSABLE;
     6>
     7>  procedure print_pkg is
     8>  begin
     9>    dbms_output.put_line('num: ' || SR_PKG.num);
    10>    dbms_output.put_line('str: ' || SR_PKG.str);
    11>
    12>    dbms_output.put_line('number of table elems: ' ||
SR_PKG.str_tab.count);
    13>    for i in 1..SR_PKG.str_tab.count loop
    14>      dbms_output.put_line(SR_PKG.str_tab(i));
    15>    end loop;
    16>  end;
    17>
    18>  procedure init_and_print_pkg(n number, v varchar2) is
    19>  begin
    20>
    21>    -- init the package globals
```

```
    22>    SR_PKG.num := n;
    23>    SR_PKG.str := v;
    24>    for i in 1..n loop
    25>      SR_PKG.str_tab(i) := v || ' ' || i;
    26>    end loop;
    27>
    28>    -- now print the package
    29>    print_pkg;
    30>  end;
    31>
    32> end SR_PKG;
    33> /
Statement processed.
SQL> show errors;
No errors for PACKAGE BODY SR_PKG
SQL>
SQL> set serveroutput on;
Server Output                   ON
SQL>
SQL> Rem SR package access in a CALL
SQL> begin
     2>
     3>   -- initialize and print the package
     4>   dbms_output.put_line('Initing and printing pkg state..');
     5>   SR_PKG.init_and_print_pkg(4, 'abracadabra');
     6>
     7>   -- print it in the same call to the server.
     8>   -- we should see the initialized values.
     9>   dbms_output.put_line('Printing package state in the same CALL...');
    10>   SR_PKG.print_pkg;
    11>
    12> end;
    13> /
Statement processed.
Initing and printing pkg state..
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
Printing package state in the same CALL...
num: 4
str: abracadabra
```

```
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
SQL>
SQL> Rem SR package access in subsequent CALL
SQL> begin
     2>
     3>  -- print the package in the next call to the server.
     4>  -- We should that the package state is reset to the initial (default)
values.
     5>  dbms_output.put_line('Printing package state in the next CALL...');
     6>  SR_PKG.print_pkg;
     7>
     8> end;
     9> /
Statement processed.
Printing package state in the next CALL...
num: 10
str: default-init-str
number of table elems: 0
SQL>
```

Example 3

This example demonstrates that any open cursors in serially reusable packages get closed automatically at the end of a WORK boundary (which is a CALL), and that in a new CALL these cursors need to be opened again.

```
Rem  For serially reusable pkg: At the end work boundaries
Rem  (which is currently the OCI call boundary) all open
Rem  cursors will be closed.
Rem
Rem  Since the cursor is closed - every time we fetch we
Rem  will start at the first row again.

SQL> connect scott/tiger;
Connected.
SQL>
SQL> drop package  SR_PKG;
Statement processed.
SQL> drop table people;
Statement processed.
SQL>
```

```
SQL>
SQL> create table people (name varchar2(20));
Statement processed.
SQL>
SQL> insert into  people  values ('ET');
1 row processed.
SQL> insert into  people  values ('RAMBO');
1 row processed.
SQL>
SQL> create or replace package SR_PKG is
     2>
     3>  pragma SERIALLY_REUSABLE;
     4>  cursor c is select name from people;
     5>
     6> end SR_PKG;
     7> /
Statement processed.
SQL> show errors;
No errors for PACKAGE SR_PKG
SQL>
SQL> set serveroutput on;
Server Output                   ON
SQL>
SQL> create or replace procedure fetch_from_cursor is
     2>   name varchar2(200);
     3> begin
     4>
     5>  if (SR_PKG.c%ISOPEN) then
     6>   dbms_output.put_line('cursor is already open.');
     7>  else
     8>   dbms_output.put_line('cursor is closed; opening now.');
     9>   open SR_PKG.c;
    10>  end if;
    11>
    12>  -- fetching from cursor.
    13>  fetch SR_PKG.c into name;
    14>  dbms_output.put_line('fetched: ' || name);
    15>
    16>  fetch SR_PKG.c into name;
    17>  dbms_output.put_line('fetched: ' || name);
    18>
    19>  -- Oops forgot to close the cursor (SR_PKG.c).
    20>  -- But, since it is a Serially Reusable pkg's cursor,
    21>  -- it will be closed at the end of this CALL to the server.
    22>
```

```
    23> end;
    24> /
Statement processed.
SQL> show errors;
No errors for PROCEDURE FETCH_FROM_CURSOR
SQL>
SQL> set serveroutput on;
Server Output                   ON
SQL>
SQL> execute fetch_from_cursor;
Statement processed.
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
SQL>
SQL> execute fetch_from_cursor;
Statement processed.
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
SQL>
SQL> execute fetch_from_cursor;
Statement processed.
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
SQL>
```

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 the Oracle Server, either to extend the functionality of the database or to give PL/SQL access to some SQL features. You may 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.

Packages Supporting SQL Features

Oracle supplies the following packaged procedures to give PL/SQL access to some features of SQL:

- DBMS_DDL
- DBMS_SESSION
- DBMS_TRANSACTION
- DBMS_UTILITY

Table 10–6 describes each of these packages. The footnotes at the end of Table 10–6 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.

Table 10–3 DBMS_DDL

Package	Procedure(Arguments)	SQL Command Equivalent
DBMS_DDL	alter_compile(type varchar2, schema varchar2, name varchar2) (notes 1, 2, 3, 4)	ALTER PROCEDURE Pro*C 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

Note 1: not allowed in triggers

Note 2: not allowed in procedures called from SQL*Forms

Note 3: not allowed in read-only transactions

Note 4: not allowed in remote (coordinated) sessions

Note 5: not allowed in recursive sessions

Note 6: not allowed in stored procedures

Table 10–4 DBMS_SESSION

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_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.
	et_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.

Note 1: not allowed in triggers

Note 2: not allowed in procedures called from SQL*Forms

Note 3: not allowed in read-only transactions

Note 4: not allowed in remote (coordinated) sessions

Note 5: not allowed in recursive sessions

Note 6: not allowed in stored procedures

Table 10–5 DBMS_TRANSACTION

Package	Procedure(Arguments)	SQL Command Equivalent
DBMS_ TRANSACTION	advise_commit	ALTER SESSION ADVISE COMMIT
	dvise_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 Also: *Oracle8 Distributed Database Systems*
	begin_discrete_transaction (notes 1,3,4,5)	See Also: *Oracle8 Tuning*
	local_transaction_id(create_transaction BOOLEAN default FALSE) return VARCHAR2	See Also: *Oracle8 Distributed Database Systems*
	step_id return number	See Also: *Oracle8 Distributed Database Systems*

Table 10–6 DBMS_UTILITYs

Package	Procedure(Arguments)	SQL Command Equivalent
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.
	analyze_part_object(schema in varchar2 default null, object_name in varchar2 default null, object_type in char default 'T', command_type in char default 'E', command_opt in varchar2 default null, sample_clause in varchar2 default 'sample 5 percent')	ANALYZE TABLE | INDEX [<schema>.]<object_name> PARTITION <pname> [<command_type>] [<command_opt>] [<sample_clause>]" for each partition of the object, run in parallel using job queues. This procedure submits a job for each partition; you can control the number of concurrent jobs with the initialization parameter JOB_QUEUE_PROCESSES. Object_type must be T (table) or I (index). Command_type can be: - C (compute statistics) - E (estimate statistics) - D (delete statistics) - V (validate structure). For V, command_opt can be 'CASCADE' when object_type is T. For C or E, command_opt can be FOR table, FOR all LOCAL indexes, FOR all columns or a combination of some of the 'for' options of analyze statistics (table). Sample_clause specifies the sample clause to use when command_type is E.
	format_error_stack return varchar2	This function formats the error stack into a variable.

Table 10–6 (Cont.) DBMS_UTILITYs

Package	Procedure(Arguments)	SQL Command Equivalent
DBMS_UTILITY (continued)	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.
	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 Also: *Oracle8 Distributed Database Systems*.

Note 1: not allowed in triggers

Note 2: not allowed in procedures called from SQL*Forms

Note 3: not allowed in read-only transactions

Note 4: not allowed in remote (coordinated) sessions

Note 5: not allowed in recursive sessions

Note 6: not allowed in stored procedures

For more details on each SQL command equivalent, see the *Oracle8 SQL Reference*.

The COMMIT, ROLLBACK, ROLLBACK... TO SAVEPOINT, and SAVEPOINT procedures are directly supported by PL/SQL; they are included in the DBMS_TRANSACTION package for completeness.

Packages Supporting Additional Functionality

Several packages are supplied with Oracle to extend the functionality of the database (DBMS_* and UTL_* packages). The cross-reference column in Table 10–7 tells you where to look for more information on each of these packages.

Table 10–7 Supplied Packages: Additional Functionality

Package Name	Description	Cross-reference
DBMS_ALERT	Supports asynchronous notification of database events.	Chapter 16, "Signalling Database Events with Alerters"
DBMS_DESCRIBE	Lets you describe the arguments of a stored procedure.	"DBMS_DESCRIBE Package" on page 10-69
DBMS_JOB	Lets you schedule administrative procedures that you want performed at periodic intervals.	*Oracle8 Administrator's Guide*
DBMS_LOCK	Lets you use the Oracle Lock Management services for your applications.	"The DBMS_LOCK Package" on page 3-18
DBMS_OUTPUT	Lets you output messages from triggers, procedures, and packages.	"Output from Stored Procedures and Triggers" on page 12-22
DBMS_PIPE	Allows sessions in the same instance to communicate with each other.	Chapter 12, "PL/SQL Input/Output"
DBMS_SHARED_POOL	Lets you keep objects in shared memory, so that they will not be aged out with the normal LRU mechanism.	*Oracle8 Tuning*
DBMS_APPLICATION_INFO	Lets you register an application name with the database for auditing or performance tracking purposes.	*Oracle8 Tuning*
DBMS_SYSTEM	Provides system-level utilities, such as letting you enable SQL trace for a session.	*Oracle8 Tuning*
DBMS_SPACE	Provides segment space information not available through standard views.	*Oracle8 Administrator's Guide*
DBMS_SQL	Lets you write stored procedures and anonymous PL/SQL blocks using dynamic SQL; lets you parse any DML or DDL statement.	Chapter 14, "Using Dynamic SQL"

Table 10–7 (Cont.) Supplied Packages: Additional Functionality

Package Name	Description	Cross-reference
DBMS_ROWID	Lets you get information about ROWIDs, including the data block number, the object number, and other components.	"The DBMS_ROWID Package" on page 10-79
DBMS_LOB	Lets you manipulate large objects using PL/SQL programs running on the Oracle Server.	"DBMS_LOB Package" on page 6-66
DBMS_AQ	Lets you add a message (of a predefined object type) onto a queue or dequeue a message.	Chapter 11, "Advanced Queuing"
DBMS_AQADM	Lets you perform administrative functions on a queue or queue table for messages of a predefined object type.	Chapter 11, "Advanced Queuing"
DBMS_DISTRIBUTED_TRUST_ADMIN	Lets you maintain the Trusted Servers List, which is used in conjunction with the list at the Central Authority to determine if a privileged database link from a particular server can be accepted.	*Oracle8 Distributed Database Systems*
DMBS_HS	Lets you administer Heterogeneous Services by registering or dropping distributed external procedures, remote libraries, and non-Oracle systems. Also lets you create or drop some initialization variables for non-Oracle systems.	*Oracle8 Distributed Database Systems*
DMBS_HS_EXTPROC	Lets you use Heterogeneous Services to establish security for distributed external procedures.	*Oracle8 Distributed Database Systems*
DMBS_HS_PASSTHROUGH	Lets you use Heterogeneous Services to send pass-through SQL statements to non-Oracle systems.	*Oracle8 Distributed Database Systems*

Table 10–7 (Cont.) Supplied Packages: Additional Functionality

Package Name	Description	Cross-reference
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.	*Oracle8 Replication*
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.	*Oracle8 Replication*
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.	*Oracle8 Replication*
DBMS_REPCAT	Lets you use Oracle's symmetric replication facility. Use of this feature requires the replication option.	*Oracle8 Replication*
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.	*Oracle8 Replication*
UTL_HTTP	Lets you make HTTP callouts from PL/SQL and SQL to access data on the Internet or to call Oracle Web Server Cartridges.	"The UTL_HTTP Package" on page 10-87

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 OCIDescribeAny() 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 the OCIDescribeAny() call, see the *Oracle Call Interface Programmer's Guide*.

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 "Privileges Required to Execute a Procedure" on page 10-38 for information on the necessary privileges for users who will be executing this package.

Security

This package is available to PUBLIC and performs its own security checking based on the schema 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 10–8. 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);
```

Table 10–8 DBMS_DESCRIBE.DESCRIBE_PROCEDURE Parameters

Parameter	Mode	Description
`object_name`	IN	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: ORA-20000 - A package was specified. You can only specify a stored procedure, stored function, packaged procedure, or packaged function. ORA-20001 - The procedure or function that you specified does not exist within the given package. ORA-20002 - The object that you specified is a remote object. This procedure cannot currently describe remote objects. ORA-20003 - The object that you specified is invalid and cannot be described. ORA-20004 - The object was specified with a syntax error.
`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.
`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 *Programmer's Guide to the Oracle Call Interface* write-up of the ODESSP call for an example of its use.

Table 10–8 (Cont.) DBMS_DESCRIBE.DESCRIBE_PROCEDURE Parameters

Parameter	Mode	Description
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, VARCHAR, 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 MLSLABEL 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.

Table 10–8 (Cont.) DBMS_DESCRIBE.DESCRIBE_PROCEDURE Parameters

Parameter	Mode	Description
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.

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   0
       1         2   PERSON       0      250      0    0     0   0
       1         1   PERSON_ID    1        2     22    4     0  10
       1         2   PERSON_NM    1        1     10    0     0   0
       1         3   AMOUNTS      0      251      0    0     0   0
       1         1                1        2     22    0     0   0
       1         4   TRANS_DATE   0       12      0    0     0   0
       2         0                0        2     22    7     2  10
       2         1   ACCOUNT_NO   0        2     22    0     0   0
       2         2   PERSON       0        2     22    4     0  10
       2         3   AMOUNTS      0      251     22    4     0  10
       2         1                1        2      0    0     0   0
       2         4   TRANS_NO     0        2      0    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);
            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 *Oracle8 Reference*.

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
```

The DBMS_ROWID Package

The functions in this package let you get the information that you need about ROWIDs. You can find out the data block number, the object number, and other components of the ROWID without having to write code to interpret the base-64 character external ROWID.

The specification for the DBMS_ROWID package is in the file *dbmsutil.sql*. This package is loaded when you create a database, and run *catproc.sql*.

Some of the functions in this package take a single parameter: a ROWID. This can be a character or a binary ROWID, either restricted or extended, as required. For each function described in this section, both the parameter types and the return type are described.

You can call the DBMS_ROWID functions and procedures from PL/SQL code, and you can also use the functions in SQL statements.

Note: ROWID_INFO is a procedure. It can only be used in PL/SQL code.

SQL Example You can use functions from the DBMS_ROWID package just like any built-in SQL function. That is, you can use them wherever an expression can be used. In this example, the ROWID_BLOCK_NUMBER function is used to return just the block number of a single row in the EMP table:

```
SELECT dbms_rowid.rowid_block_number(rowid)
  FROM emp
```

```
  WHERE ename = 'KING';
```

PL/SQL Example This example returns the ROWID for a row in the EMP table, extracts the data object number from the ROWID, using the ROWID_OBJECT function in the DBMS_ROWID package, then displays the object number:

```
DECLARE
  object_no   INTEGER;
  row_id      ROWID;
  ...
BEGIN
  SELECT ROWID INTO row_id FROM emp
    WHERE empno = 7499;
  object_no := dbms_rowid.rowid_object(row_id);
  dbms_output.put_line('The obj. # is '|| object_no);
  ...
```

Summary

Table 10–9 lists the functions and procedures in the DBMS_ROWID package.

Table 10–9 DBMS_ROWID Functions

Function Name	Description	See Page
ROWID_CREATE	Create a ROWID, for testing only.	81
ROWID_INFO	Procedure that returns the type and components of a ROWID	82
ROWID_TYPE	Returns the ROWID type: 0 is restricted, 1 is extended.	83
ROWID_OBJECT	Returns the object number of the extended ROWID.	83
ROWID_RELATIVE_FNO	Returns the file number of a ROWID.	84
ROWID_BLOCK_NUMBER	Returns the block number of a ROWID.	84
ROWID_ROW_NUMBER	Returns the row number.	84
ROWID_TO_ABSOLUTE_FNO	Returns the absolute file number associated with the ROWID for a row in a specific table.	84
ROWID_TO_EXTENDED	Converts a ROWID from restricted format to extended.	85

Table 10–9 (Cont.) DBMS_ROWID Functions

Function Name	Description	See Page
ROWID_TO_RESTRICTED	Converts an extended ROWID to restricted format.	86
ROWID_VERIFY	Checks if a ROWID can be correctly extended by the ROWID_TO_EXTENDED function.	87

Exceptions

The DBMS_ROWID package functions and procedures can raise the ROWID_INVALID exception. The exception is defined in the DBMS_ROWID package as:

```
PRAGMA EXCEPTION_INIT(ROWID_INVALID, -1410);
```

ROWID_CREATE Function

The ROWID_CREATE function lets you create a ROWID, given the component parts as parameters. This function is mostly useful for testing ROWID operations, since only the Oracle Server can create a valid ROWID that points to data in a database.

Syntax

```
FUNCTION DBMS_ROWID.ROWID_CREATE(
    rowid_type        IN  NUMBER,
    object_number     IN  NUMBER,
    relative_fno      IN  NUMBER,
    block_number      IN  NUMBER,
    row_number        IN  NUMBER)
  RETURN ROWID;
```

Set the ROWID_TYPE parameter to 0 for a restricted ROWID, and to 1 to create an extended ROWID.

If you specify ROWID_TYPE as 0, the required OBJECT_NUMBER parameter is ignored, and ROWID_CREATE returns a restricted ROWID.

Example

Create a dummy extended ROWID:

```
my_rowid := DBMS_ROWID.ROWID_CREATE(1, 9999, 12, 1000, 13);
```

Find out what the ROWID_OBJECT function returns:

```
obj_number := DBMS_ROWID.ROWID_OBJECT(my_rowid);
```

The variable OBJ_NUMBER now contains 9999.

ROWID_INFO Procedure

This procedure returns information about a ROWID, including its type (restricted or extended), and the components of the ROWID. This is a procedure, and cannot be used in a SQL statement.

Syntax

```
DBMS_ROWID.ROWID_INFO(
    rowid_in         IN  ROWID,
    rowid_type       OUT  NUMBER,
    object_number    OUT  NUMBER,
    relative_fno     OUT  NUMBER,
    block_number     OUT  NUMBER,
    row_number       OUT  NUMBER);
```

The IN parameter ROWID_IN determines if the ROWID is a restricted (0) or extended (1) ROWID.

The OUT parameters return the information about the ROWID, as indicated by their names.

For information about the ROWID_TYPE parameter, see the ROWID_TYPE function on page 10-83.

Example

To read back the values for the ROWID that you created in the ROWID_CREATE example:

```
DBMS_ROWID.ROWID_INFO(my_rowid, rid_type, obj_num,
  file_num, block_num, row_num);

DBMS_OUTPUT.PUT_LINE('The type is ' || rid_type);
DBMS_OUTPUT.PUT_LINE('Data object number is ' || obj_num);
-- and so on...
```

ROWID_TYPE Function

This function returns 0 if the ROWID is a restricted ROWID, and 1 if it is extended.

Syntax

```
FUNCTION DBMS_ROWID.ROWID_TYPE(rowid_val IN ROWID)
  RETURN NUMBER;
```

Example

```
IF DBMS_ROWID.ROWID_TYPE(my_rowid) = 1 THEN
  my_obj_num := DBMS_ROWID.ROWID_OBJECT(my_rowid);
```

ROWID_OBJECT Function

This function returns the data object number for an extended ROWID. The function returns zero if the input ROWID is a restricted ROWID.

Syntax

```
DBMS_ROWID.ROWID_OBJECT(rowid_val IN ROWID)
  RETURN NUMBER;
```

Example

```
SELECT dbms_rowid.rowid_object(ROWID)
  FROM emp
  WHERE empno = 7499;
```

ROWID_RELATIVE_FNO Function

This function returns the relative file number of the ROWID specified as the IN parameter. (The file number is relative to the tablespace.)

Syntax

```
DBMS_ROWID.ROWID_RELATIVE_FNO(rowid_val IN ROWID)
  RETURN NUMBER;
```

Example

The example PL/SQL code fragment returns the relative file number:

```
DECLARE
  file_number    INTEGER;
  rowid_val      ROWID;
```

```
BEGIN
  SELECT ROWID INTO rowid_val
    FROM dept
    WHERE loc = 'Boston';
  file_number :=
    dbms_rowid.rowid_relative_fno(rowid_val);
  ...
```

ROWID_BLOCK_NUMBER Function

This function returns the database block number for the input ROWID.

Syntax

```
DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid_val IN ROWID)
  RETURN NUMBER;
```

Example

The example SQL statement selects the block number from a ROWID and inserts it into another table:

```
INSERT INTO T2 (SELECT dbms_rowid.rowid_block_number(ROWID)
  FROM some_table
  WHERE key_value = 42);
```

ROWID_ROW_NUMBER Function

This function extracts the row number from the ROWID IN parameter.

Syntax

```
DBMS_ROWID.ROWID_ROW_NUMBER(rowid_val IN ROWID)
  RETURN NUMBER;
```

Example

Select a row number:

```
SELECT dbms_rowid.rowid_row_number(ROWID)
  FROM emp
  WHERE ename = 'ALLEN';
```

ROWID_TO_ABSOLUTE_FNO Function

This function extracts the absolute file number from a ROWID, where the file number is absolute for a row in a given schema and table. The schema name and the

name of the schema object (such as a table name) are provided as IN parameters for this function.

Syntax

```
DBMS_ROWID.ROWID_TO_ABSOLUTE_FNO(
  rowid_val          IN  ROWID,
  schema_name        IN  VARCHAR2,
  object_name        IN  VARCHAR2)
      RETURN NUMBER;
```

Example

```
DECLARE
  rel_fno        INTEGER;
  rowid_val      CHAR(18);
  object_name    VARCHAR2(20) := 'EMP';
BEGIN
  SELECT ROWID INTO rowid_val
    FROM emp
    WHERE empno = 9999;
  rel_fno := dbms_rowid.rowid_to_absolute_fno(
    rowid_val, 'SCOTT', object_name);
```

ROWID_TO_EXTENDED Function

This function translates a restricted ROWID that addresses a row in a schema and table that you specify to the extended ROWID format.

Syntax

```
DBMS_ROWID.ROWID_TO_EXTENDED(
  restr_rowid        IN ROWID,
  schema_name        IN VARCHAR2,
  object_name        IN VARCHAR2)
       RETURN ROWID;
```

Example

Assume that there is a table called RIDS in the schema SCOTT, and that the table contains a column ROWID_COL that holds ROWIDs (restricted), and a column TABLE_COL that point to other tables in the SCOTT schema. You can convert the ROWIDs to extended format with the statement:

```
UPDATE SCOTT.RIDS
  SET rowid_col =
```

```
dbms_rowid.rowid_to_extended(rowid_col,
  'SCOTT", TABLE_COL);
```

Usage

ROWID_TO_EXTENDED returns the ROWID in the extended character format. If the input ROWID is NULL, the function returns NULL. If a zero-valued ROWID is supplied (00000000.0000.0000), a zero-valued restricted ROWID is returned.

If the schema and object names are provided as IN parameters, this function verifies SELECT authority on the table named, and converts the restricted ROWID provided to an extended ROWID, using the data object number of the table. That ROWID_TO_EXTENDED returns a value, however, does not guarantee that the converted ROWID actually references a valid row in the table, either at the time that the function is called, or when the extended ROWID is actually used.

If the schema and object name are not provided (are passed as NULL), then this function attempts to fetch the page specified by the restricted ROWID provided. It treats the file number stored in this ROWID as the absolute file number. This can cause problems if the file has been dropped, and its number has been reused prior to the migration. If the fetched page belongs to a valid table, the data object number of this table is used in converting to an extended ROWID value. This is very inefficient, and Oracle recommends doing this only as a last resort, when the target table is not known. The user must still know the correct table name at the time of using the converted value.

If an extended ROWID value is supplied, the data object number in the input extended ROWID is verified against the data object number computed from the table name parameter. If the two numbers do not match, the INVALID_ROWID exception is raised. If they do match, the input ROWID is returned.

See the ROWID_VERIFY function on page 10-87 for a method to determine if a given ROWID can be converted to the extended format.

ROWID_TO_RESTRICTED Function

This function converts an extended ROWID into restricted ROWID format.

Syntax

```
DBMS_ROWID.ROWID_TO_RESTRICTED(ext_rowid IN ROWID)
    RETURN ROWID;
```

Example

```
INSERT INTO RID_T2@V7db1
```

```
SELECT dbms_rowid.rowid_to_restricted(ROWID)
  FROM scott.emp@O8db1
  WHERE ename = 'SMITH';
```

ROWID_VERIFY Function

This function returns 0 if the input restricted ROWID can be converted to extended format, given the input schema name and table name, and it returns 1 if the conversion is not possible. Note that you can use this function in a BOOLEAN context in a SQL statement, as shown in the example.

Syntax

```
DBMS_ROWID.ROWID_VERIFY(
  restr_rowid        IN ROWID,
  schema_name        IN VARCHAR2,
  object_name        IN VARCHAR2)
       RETURN ROWID;
```

Example

Considering the schema in the example for the ROWID_TO_EXTENDED function on page 10-85, you can use the following statement to find bad ROWIDs prior to conversion:

```
SELECT ROWID, rowid_col
  FROM SCOTT.RIDS
  WHERE dbms_rowid.rowid_verify(rowid_col, NULL, NULL);
```

The UTL_HTTP Package

The stored package UTL_HTTP makes HTTP (hyper-text transfer protocol) callouts from PL/SQL and SQL. You can use it to access data on the internet, or to call Oracle Web Server Cartridges. The package contains two similar entrypoints, each of which takes a string URL (universal resource locator), contacts that site, and returns the data (typically HTML — hyper-text markup language) obtained from that site.

This is the specification of packaged function UTL_HTTP.REQUEST:

```
function request (url in varchar2) return varchar2;
```

UTL_HTTP.REQUEST returns up to the first 2000 bytes of the data retrieved from the given URL. For example:

```
SVRMGR> select utl_http.request('http://www.oracle.com/') from dual;
UTL_HTTP.REQUEST('HTTP://WWW.ORACLE.COM/')
```

```
------------------------------------------------------------------
<html>
<head><title>Oracle Corporation Home Page</title>
<!--changed Jan. 16, 19
1 row selected.
```

This is the specification of packaged function UTL_HTTP.REQUEST_PIECES, which uses type UTL_HTTP.HTML_PIECES:

```
type html_pieces is table of varchar2(2000) index by binary_integer;
function request_pieces (url in varchar2,
         max_pieces natural default 32767)
  return html_pieces;
```

UTL_HTTP.REQUEST_PIECES returns a PL/SQL-table of 2000-byte pieces of the data retrieved from the given URL. The optional second argument places a bound on the number of pieces retrieved. For example, the following block retrieves up to 100 pieces of data (each 2000 bytes, except perhaps the last) from the URL. It prints the number of pieces retrieved and the total length, in bytes, of the data retrieved.

```
set serveroutput on
/
declare
  x utl_http.html_pieces;
begin
  x := utl_http.request_pieces('http://www.oracle.com/', 100);
  dbms_output.put_line(x.count || ' pieces were retrieved.');
  dbms_output.put_line('with total length ');
  if x.count < 1
  then dbms_output.put_line('0');
  else dbms_output.put_line
  ((2000 * (x.count - 1)) + length(x(x.count)));
  end if;
end;
/
```

Here is the output:

```
Statement processed.
4 pieces were retrieved.
with total length
7687
```

Below is the specification for package UTL_HTTP. It describes the exceptions that can be raised by functions REQUEST and REQUEST_PIECES:

```
create or replace package utl_http is
```

```
-- Package UTL_HTTP contains functions REQUEST and REQUEST_PIECES for
-- making HTTP callouts from PLSQL programs.

-- Function REQUEST takes a URL as its argument.  Its return-type is a
-- string of length 2000 or less, which contains up to the first 2000 bytes
-- of the html result returned from the HTTP request to the argument URL.

function request (url in varchar2) return varchar2;
pragma restrict_references (request, wnds, rnds, wnps, rnps);

-- Function REQUEST_PIECES also takes a URL as its argument.  Its
-- return-type is a PLSQL-table of type UTL_HTTP.HTML_PIECES.  Each
-- element of that PLSQL-table is a string of length  2000. The
-- final element may be shorter than 2000 characters.

type html_pieces is table of varchar2(2000) index by binary_integer;

function request_pieces (url in varchar2,
      max_pieces natural default 32767)
  return html_pieces;
pragma restrict_references (request_pieces, wnds, rnds, wnps, rnps);

-- The elements of the PLSQL-table returned by REQUEST_PIECES are
-- successive pieces of the data obtained from the HTTP request to that
-- URL.  Here is a typical URL:
--     http://www.oracle.com
-- So a call to REQUEST_PIECES could look like the example below. Note the
-- use of the plsql-table method COUNT to discover the number of pieces
-- returned, which may be zero or more:
--
-- declare pieces utl_http.html_pieces;
-- begin
--   pieces := utl_http.request_pieces('http://www.oracle.com/');
--   for i in 1 .. pieces.count loop
--     .... -- process each piece
--   end loop;
-- end;
--

-- The second argument to REQUEST_PIECES, "MAX_PIECES", is optional.  It is
-- the maximum number of pieces (each 2000 characters in length, except for
-- the last, which may be shorter), that REQUEST_PIECES should return.  If
-- provided, that argument should be a positive integer.

-- Exceptional conditions:
```

```
-- If initialization of the http-callout subsystem fails (for
-- environmental reasons, for example, lack of available memory)
-- then exception UTL_HTTP.INIT_FAILED is raised:

init_failed exception;

-- When the HTTP call fails (e.g., because of failure of the HTTP daemon;
-- or because of the argument to REQUEST  or REQUEST_PIECES cannot be
-- interpreted as a URL because it is NULL or has non-HTTP syntax) then
-- exception UTL_HTTP.REQUEST_FAILED is raised.

request_failed exception;

-- Note that the above two exceptions, unless explicitly caught by an
-- exception handler, will be reported by this generic message:
--   ORA-06510: PL/SQL: unhandled user-defined exception
-- which reports them as "user-defined" exceptions, although
-- they are defined in this system package.

-- If any other exception is raised during the processing of the http
-- request (for example, an out-of-memory error), then function REQUEST
-- or REQUEST_PIECES reraises that exception.

-- When no response is received from a request to the given URL
-- (for example, because no site corresponding to that URL is contacted)
-- then a formatted html error message may be returned.  For example:
--
--  <HTML>
--  <HEAD>
--  <TITLE>Error Message</TITLE>
--  </HEAD>
--  <BODY>
--  <H1>Fatal Error 500</H1>
--  Can't Access Document:  http://home.nothing.comm.
--  <P>
--  <B>Reason:</B> Can't locate remote host:  home.nothing.comm.
--  <P>
--
--  <P><HR>
--  <ADDRESS><A HREF="http://www.w3.org">
--   CERN-HTTPD3.0A</A></ADDRESS>
--  </BODY>
--  </HTML>
--
```

```
-- You should not expect for UTL_HTTP.REQUEST or UTL_HTTP.REQUEST_PIECES
-- to succeed in contacting a URL unless you can contact that URL by using
-- a browser on the same machine (and with the same privileges, environment
-- variables, etc.)  If REQUEST or REQUEST_PIECES fails (i.e., if it raises
-- an exception, or returns a HTML-formatted error message, yet you believe
-- that the URL argument is correct), please try contacting that same URL
-- with a browser, to verify network availability from your machine.

end utl_http;
```

11

Advanced Queuing

This chapter has four sections:

- *Introduction to Oracle Advanced Queuing* —
  - describes the relationship between queuing and the requirements for complex information handling in distributed environments
  - lists the features of Oracle AQ
  - details the primary components of Oracle AQ
- *Oracle Advanced Queuing by Example* — takes a step-by-step approach to using Oracle AQ
- *Oracle Advanced Queuing Reference* — contains a detailed description of the technical specifications for Oracle AQ
- *Compatibility and Upgrade* — describes the compatibility of Oracle AQ 8.0.4 with Oracle AQ 8.0.3, and the steps necessary to upgrade to the latest version

---

WARNING:

- **If you purchase the product, *Oracle8*, you will not be able to use Oracle AQ.**
- **If you purchase the product, *Oracle8 Enterprise Edition*, without the *Objects Option*, you will be able to use Oracle AQ with queues of `RAW` type only.**
- **If you purchase the product, *Oracle8 Enterprise Edition*, with the *Objects Option*, you will be able to use the full functionality of Oracle AQ.**

Introduction to Oracle Advanced Queuing

Introduction Overview

This introductory section:

- Introduces the requirements for complex information handling in a distributed environment in terms of three sample scenarios
- Considers two solutions to the problems common to the scenarios:
  - A synchronous communication model
  - The deferred messaging system made possible by Oracle AQ
- Describes the features of Oracle AQ
  - General features
  - Enqueue features
  - Dequeue features
  - Propagation: Enqueuing & Dequeueing
- Details Primary Components of Oracle AQ
  - Queue entities
  - Basic Queuing
  - Multiple-Consumer Dequeuing of the Same Message

Complex Systems

Consider the following application scenarios.

Application Scenario 1: A brokerage firm, *Makers & Breakers,* advertises to the public that its new service will let clients stipulate time as well as price as a parameter i.e. a request to buy or sell is not executed unless it takes place within a specific time period (e.g., within 15 minutes). The campaign is extremely successful and a mutual fund house, *America's Standard Guarantee,* takes advantage of the technology to offer its clients an opportunity to buy and sell units during course of the day rather than at the close of trading. However, *M&B* are informed by the Securities and Exchange Commission that they have received two complaints:

- That in executing the buy / sell orders *M&B* are giving an unfair advantage to large customers, such as the mutual fund house.
- That in managing the time parameter *M&B* are taking advantage of their customers e.g., in a falling market selling earlier in the time period than they inform their clients, and then pocketing the difference.

The problem facing *Makers & Breakers* is not only to answer these unfounded charges but to do so quickly and in such a way as to leave no doubt in the minds of the public regarding the fairness of their practices.

Application Scenario 2: A large state university(35,000 students) decides to automate its class enrollment process. Students will be able to register for classes using web templates from home or at terminals on both the main and satellite campuses for any of the more than a thousand classes offered by *Big U*. The administration announces that the following parameters will apply:

- Priority: registration is on a 'first come' basis except that

- seniors receive priority for upper level courses, followed by juniors, sophomores and first-year students (frosh);

- frosh receive priority for entry level courses, followed by a senior who needs the course to graduate.

- Registration phases: the above priority criteria hold only for specific defined time phases e.g., in the second phase, seniors and juniors are treated as being on an equal 'first come' basis with regard to upper-level classes, but continue to receive priority over juniors and frosh.

- Full-time undergraduates must register for a minimum of three classes and may register for a maximum of four classes without special permission. Students may register for as many as ten classes (ranking their preferences 1-10) in case they are not admitted to their preferred classes, but only the first three choices are regarded as 'live'. In the event that a class becomes full, the student's next choice becomes 'live'. However, should a full class develop a vacancy, the student with the highest priority will be admitted, at that time being removed from the roll of a class of her/his lower preference.

Application Scenario 3: A power utility, *Most Power*, develops a sophisticated model in order to decide how to deploy its resources. The way the system works is that the utility gets ongoing reports from

- numerous weather centers regarding current conditions, and
- power stations regarding ongoing utilization.

It then compares this information to historical data in order to predict demand for power in specific geographic areas for given time periods. A crucial part of this modeling has to do with noting the rapidity and degree of change in the incoming reports as weather changes and power is deployed.

During a prolonged blizzard, matters are complicated by failure of a power station which also forwards weather data. The question facing *Most Power* is whether to purchase power from a neighboring utility, since there is a lead time of five days for making such arrangements.

Queuing and the Intricacy of Message Passing

Although not every application developer will have worked with each of these types of scenario, the basic elements of these problem domains will be familiar. Each of these scenarios describes a situation in which messages come from and are disbursed to multiple clients (nodes) in a distributed computing environment. Messages are not only passed back and forth between client and server but also are intricately interleaved between processes on the server.

If we focus on these scenarios in terms of *messages*, the applications can be viewed as consisting of multi-step processes in which each step is triggered by one or more messages, and gives rises to one or more messages. Another way of saying this is that *messages are events that trigger other message-events.*

Business Process Management, or Workflow, which is based on this notion of the interrelation of messages and events, is becoming recognized as a fundamental technology. *Queuing* is one of the key technologies for this class of application because it implements deferred execution of messages. This decoupling of 'requests for service' from 'supply of services' increases efficiency, and provides the infrastructure for complex scheduling.

Securing Messages in a Vulnerable Environment

Handling the intricacy message-passing is not the only problem. Unfortunately, networks, computing hardware, and software applications will all fail from time to time, as is the case in power utility scenario. Nevertheless, the ACID properties of the information must be preserved. Chaos would quickly follow if buy orders and the order in which they issued were 'lost', or if the changing status of students could not be matched to class availability, or if power could not routed to where the combination of incoming reports an historical patterns of changes in demand indicated it would be most required. In other words, *messaging must be persistent.* By integrating transaction processing with queuing technology, persistent messaging in the form of *queuing* is made possible. The importance of queuing has been proven by TP-monitors that typically include such a facility.

Message Persistence as Extension in Time and Space

The persistence of messages that is required goes beyond the ability to recover information in the event of system failure. Applications may have to deal with multiple unprocessed messages arriving simultaneously from external clients or from programs internal to the application. The communication links between databases may not be available all the time or may be reserved for some other purpose. If the system falls short in its capacity to deal with these messages immediately, the application must be able to store the messages until they can be processed. By the same token, external clients or internal programs may not be ready to receive messages that have been processed.

Even more important, *applications must be able to deal with priorities:* messages arriving later may be of higher priority than messages arriving earlier; messages arriving earlier may have to wait for messages arriving later before actions are executed; the same message may have to be accessed by different processes; and so on. All these cases become more pressing in situations in which messages are communicated between remote locations.

Moreover, priorities are not fixed. One crucial dimension of handling the dynamic aspect of message persistence has to do with *windows of opportunity* that grow and shrink.It may be that messages in a specific queue become more important than

messages in other queues, and so need to be processed with less delay or interference from messages in other queues. Similarly, it may be more pressing to send messages to some destinations than to others. In the case of the share brokerage application, the window for completing the sale shrinks to nothing (i.e. an offer to sell expires) from the time the offer to sell message is received. In the case of the student registration application, different priorities apply during different temporal phases, and data must be re-evaluated with the transition from one phase to another. And in the case of the power utility, the entire decision-making process depends on whether conditions are stable (the persistence of a large window) or dynamic (the rapid appearance and disappearance of windows).

Control Data as Essential Information

What is true for all the scenarios is the time that messages are received or dispatched is a crucial part of the message. This means that the control component of the message — in this case, time markers — is as important as the payload data. Put another way: *the message retains importance as a business asset after it has been executed.*

Persistent messaging thus implies accurate documentation of messages for analysis of historical patterns and future trends. For instance:

- The ability to retrieve the sequence of messages is absolutely critical for the brokerage firm in the first scenario to refute the charges made to the SEC. They must be able to show that the offer to sell made by client_A was matched by the first available offer to buy by client_B.
- With regard to student registration, the withdrawal of a student from a class which is full, requires (1) tracking-down the next student in line based on priority, time-period and specified preference, (2) moving him / her from a class in which he / she is registered into the available spot, and (3) dealing with the resultant repercussions i.e. keeping track of the relationships between messages and navigating from one message to another based on queries
- In the case of the power utility, messages about weather and power utilization need to be preserved over time so as to be able to analyze patterns by querying message warehouses. The utility is specifically concerned with time lapses between events e.g.,
  - between distinguishing where power is needed and distributing the power
  - between sending the message to distribute power and the actual distribution.

Specific Requirements of a Messaging System

What are the key requirements of a persistent messaging system given the above issues?

- In order to deal with increasingly complex applications, developers need mechanisms that provide the ability to receive, hold and disburse messages while preserving ACID properties. These messages must be communicable between programs on clients and servers over networks, or between programs on the server.
- Queuing systems must provide an integrated solution so that messages combine both control information and content. The ideal solution should be able to queue messages, and treat those queued messages as events that may trigger other messages.
- Program optimization requires that the queuing system and the database share the same resource manager, and thereby avoid incurring the overhead of two phase commit.
- The message system should exhibit high performance characteristics as measured by the following metrics:
  - Number of messages enqueued / dequeued per second.
  - Time to evaluate a complex query on a message warehouse.
  - Time to recover / restart the messaging process after a failure.
- The message system should exhibit high scalability. That is, the system should continue to exhibit high performance as the number of programs using the application increase, as the number of messages increase, and as the size of the message warehouse increases.
- Both content (data payload) and control dimension of messages should be available. For instance, the application should be able to implement content-based routing, content-based subscription, and content-based querying.
- Tracking and documentation should be the responsibility of the messaging system, not the developer.

Possible Solutions: Synchronous versus Disconnected/Deferred Communication

Generally, attempts to provide communication between programs can be classified into one of two types: *Synchronous* and *Disconnected/Deferred* Communication.

Synchronous Communication

This model of communication, also called *on-line* or *connected*, is based on the request/reply paradigm. In this model a program sends a request to another program and waits (blocks) until the reply arrives. This model of communication, in which the sender and receiver of the message are tightly coupled, is suitable for programs that need to get a reply before they can proceed with any task. Traditional client/server architectures are based on this model.

The major drawback of the synchronous model of communication is that the programs must be available and running for the application to work. In the event of network or machine failure, or even if the program needed being busy, the entire application grinds to a halt.

Disconnected/Deferred Messaging

In this model programs in the role of producers place requests in a queue and then proceed with their work. Programs in the role of consumers retrieve requests from the queue and acts on them. This model is well suited for applications that can continue with their work after placing a request in the queue because they are not blocked waiting for a reply. It is also suited to applications that can continue with their work until there is a message to retrieve.

For deferred execution to work correctly even in the presence of network, machine and application failures, the requests must be stored persistently, and processed exactly once. This can be achieved by combining persistent queuing with transaction protection. Oracle8 provides a queuing technology that does not depend on the use of *TP-monitors* or any other evolving *Message-Oriented Middleware (MOM)* infrastructure.

Oracle Advanced Queuing — Features

Oracle AQ (Oracle Advanced Queueing) provides message queuing as an integrated part of the Oracle server. Oracle AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database*. By providing an integrated solution Oracle AQ frees application developers to devote their efforts to their specific business logic rather than having to construct a messaging infrastructure.

General Features

- **SQL access:** Messages are placed in normal rows in a database table. They can be queried using standard SQL. Thus, users can use SQL to access the message

properties, the message history and the payload. All available SQL technology, such as indexes, can be used to optimize the access to messages.

- **Integrated database level operational support:** All standard database features such as recovery, restart and enterprise manager are supported. Oracle AQ queues are implemented in database tables, hence all the operational benefits of high availability, scalability and reliability are applicable to queue data. In addition, database development and management tools can be used with queues. For instance, queue tables can be imported and exported.
- **Structured Payload:** Users can use object types to structure and manage the payload. RDBMSs in general have had a far richer typing system than messaging systems. Since Oracle8 is an object-relational DBMS, it supports both traditional relational types as well as user-defined types. Many powerful features are enabled as a result of having strongly typed content i.e. content whose format is defined by an external type system. These include:
  - Content-based routing: an external agent can examine the content and route the message to another queue based on the content.
  - Content-based subscription: a publish and subscribe system built on top of a messaging system which can offer content based on subscription.
  - Querying: the ability to execute queries on the content of the message enables message warehousing.
- **Retention and message history:** Users can specify that messages be retained after consumption. The systems administrator can specify the duration for which messages will be retained. Oracle AQ stores information about the history of each message, preserving the queue and message properties of delay, expiration, and retention for messages destined for local or remote recipients. The information contains the ENQUEUE / DEQUEUE time and the identification of the transaction that executed each request. This allows users to keep a history of relevant messages. The history can be used for tracking, data warehouse and data mining operations.
- **Tracking and event journals:** If messages are retained they can be related to each other. For example: if a message m2 is produced as a result of the consumption of message m1, m1 is related to m2. This allows users to track sequences of related messages. These sequences represent ‘event journals’ which are often constructed by applications. Oracle AQ is designed to let applications create event journals automatically.
- **Integrated transactions:** The integration of control information with content (data payload) simplifies application development and management.

ENQUEUE Features

- **Correlation identifier:** Users can assign an identifier to each message, thus providing a means to retrieve specific messages at a later time.
- **Subscription & Recipient lists:** A single message can be designed to be consumed by multiple consumers. A queue administrator can specify the list of subscribers who can retrieve messages from a queue. Different queues can have different subscribers, and a consumer program can be a subscriber to more than one queue. Further, specific messages in a queue can be directed toward specific recipients who may or may not be subscribers to the queue, thereby overriding the subscriber list.
- **Priority and ordering of messages in enqueuing:** It is possible to specify the priority of the enqueued message. An enqueued message can also have its exact position in the queue specified. This means that users have three options to specify the order in which messages are consumed: (a) a sort order specifies which properties are used to order all message in a queue; (b) a priority can be assigned to each message; (c) a sequence deviation allows you to position a message in relation to other messages. Further, if several consumers act on the same queue, a consumer will get the first message that is available for immediate consumption. A message that is in the process of being consumed by another consumer will be skipped.
- **Message grouping:** Messages belonging to one queue can be grouped to form a set that can only be consumed by one user at a time. This requires the queue be created in a queue table that is enabled for message grouping. All messages belonging to a group have to be created in the same transaction and all messages created in one transaction belong to the same group. This feature allows users to segment complex messages into simple messages, e.g., messages directed to a queue containing invoices could be constructed as a group of messages starting with the header message, followed by messages representing details, followed by the trailer message.
- **Propagation**: This feature enables applications to communicate with each other without having to be connected to the same database or to the same Queue. Messages can be propagated from one Oracle AQ to another, irrespective of whether these are local or remote. The propagation is done using the familiar database links, and Net8.
- **Time specification and Scheduling:** Delay interval and / or expiration intervals can be specified for an enqueued message, thereby providing windows of execution. A message can be marked as available for processing only after a specified time elapses (a delay time) and has to be consumed before a specified time

limit expires. Messages can be scheduled to propagate from a queue to local or remote destinations. Administrators can specify the start time, the propagation window and a function to determine the next propagation window (for periodic schedules).

DEQUEUE Features

- **Multiple recipients:** A message in queue can be retrieved by multiple recipients without there being multiple copies of the same message.
- **Local and remote recipients**: Designated recipients can be located locally and / or at remote sites.
- **Navigation of messages in dequeuing:** Users have several options to select a message from a queue. They can select the first message or once they have selected a message and established a position, they can retrieve the next. The selection is influenced by the ordering or can be limited by specifying a correlation identifier. Users can also retrieve a specific message using the message identifier.
- **Modes of dequeuing:** a DEQUEUE request can either browse or remove a message. If a message is browsed it remains available for further processing, if a message is removed, it is not available any more for DEQUEUE requests. Depending on the queue properties a removed message may be retained in the queue table.
- **Optimization of waiting for the arrival of messages:** A DEQUEUE could be issued against an empty queue. To avoid polling for the arrival of a new message a user can specify if and for how long the request is allowed to wait for the arrival of a message.
- **Retries with delays:** A message has to be consumed exactly once. If an attempt to dequeue a message fails and the transaction is rolled back, the message will be made available for reprocessing after some user specified delay elapses. Reprocessing will be attempted up to the user-specified limit.
- **Optional transaction protection:** ENQUEUE / DEQUEUE requests are normally part of a transaction that contains the requests, thereby providing the desired transactional behavior. Users can, however, specify that a specific request is a transaction by itself making the result of that request immediately visible to other transactions. This means that messages can be made visible to the external world either as soon as soon as the ENQUEUE or DEQUEUE statement is issued, or only after the transaction is committed.

- **Exception handling:** A message may not be consumed within given constraints, i.e. within the window of execution or within the limits of the retries. If such a condition arises, the message will be moved to a user-specified exception queue.

Propagation Features

- **Automated coordination of enqueuing and dequeuing**: As already noted, recipients can be local or remote. Oracle 8.0.4 does not support distributed object types, hence remote enqueuing or dequeuing using a standard database link does not work. However, in Oracle 8.0.4 customers can use AQ's message propagation to enqueue to a remote queue.

  For example, you can connect to database X and enqueue the message in a queue, say "DROPBOX" located in database X. You can configure AQ so that all messages enqueued in queue "DROPBOX" will be automatically propagated to another queue in a database Y, regardless whether database Y is local or remote. AQ will automatically check if the type of the remote queue in database Y is structurally equivalent to the type of the local queue in database X, and propagate the message.

  Recipients of propagated messages can be either applications or queues. If the recipient is a queue, the actual recipients will be determined by the subscription list associated with the recipient queue.If the queues are remote, messages will be propagated using the specified database link. Only AQ to AQ message propagation is supported.

Oracle Advanced Queuing — Primary Components

Queuing Entities

Message A message is the smallest unit of information inserted into and retrieved from a queue. A message consists of control information (metadata) and payload (data). The control information represents message properties used by AQ to manage messages. The payload data is the information stored in the queue and is transparent to Oracle AQ. A message can reside in only one queue. A message is created by the enqueue call and consumed by the dequeue call.

Queue A queue is a repository for messages. There are two types of queues: user queues, also known as normal queues, and exception queues. The user queue is for normal message processing. Messages are transferred to an exception queue if they can not be retrieved and processed for some reason. Queues can be created, altered, started, stopped, and dropped by using the *Oracle AQ administrative interfaces.*

Queue Table Queues are stored in queue tables. Each queue table is a database table and contains one or more queues. Each queue table contains a default exception queue.

The following figure shows the relationship between messages, queues, and queue tables. The columns represent message queues, with rows representing individual messages.

Agents An agent is a queue user. There are two types of agents: producers who place messages in a queue (enqueuing) and consumers who retrieve messages (dequeuing). Any number of producers and consumers may be accessing the queue at a given time. Agents insert messages into a queue and retrieve messages from the queue by using the *Oracle AQ operational interfaces*

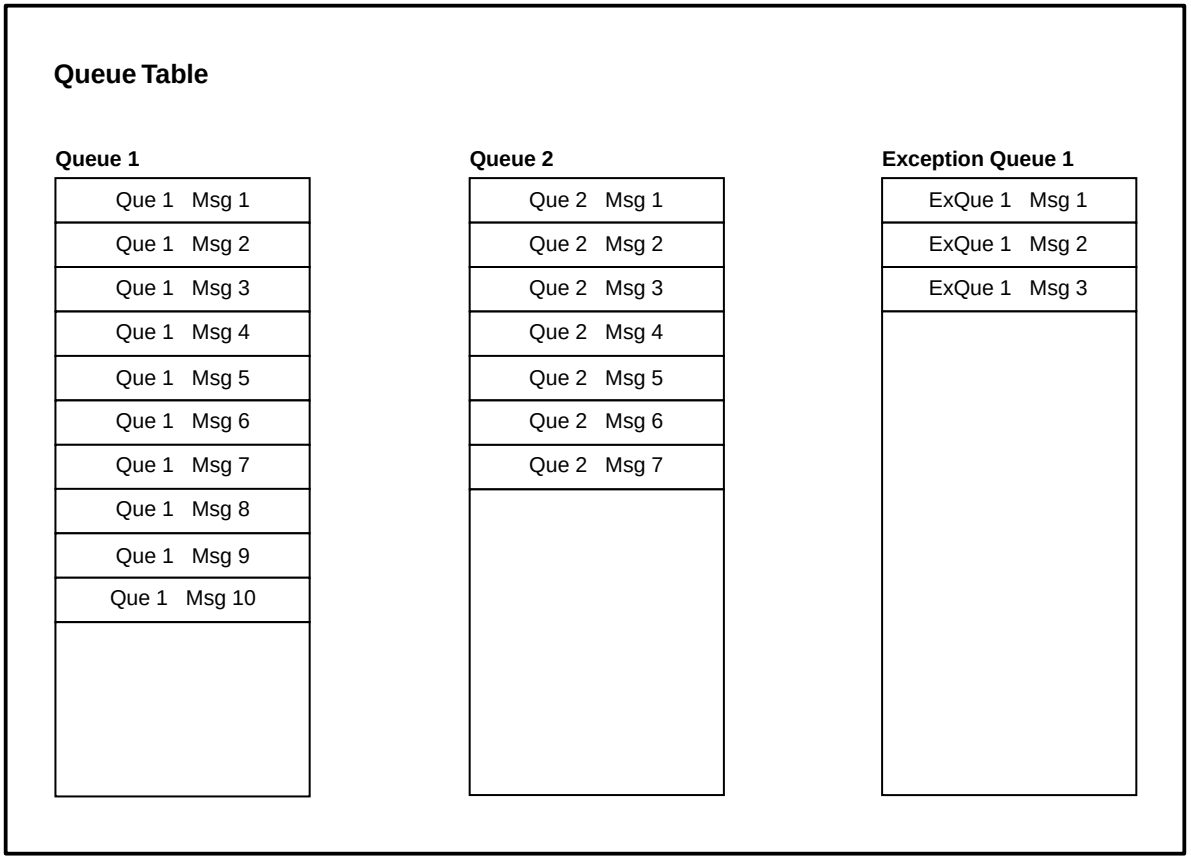
An agent is identified by its *name*, *address* and *protocol*. The address field is a character field of up to 1024 bytes that is interpreted in the context of the protocol. For instance, the default value for the protocol is 0, signifying a database link addressing. In this case, the address for this protocol is of the form

```
queue_name@dblink
```

where `queue_name` is of the form `[schema.]queue` and `dblink` may either be a fully qualified database link name or the database link name without the domain name. The only supported protocol value is 0 at this time.

Queue Monitor The queue monitor is a background process that monitors the messages in the queue. It provides the mechanism for message expiration, retry and delay.

Figure 11–1 Modeling Queue Entities



Modeling Queue Entities

Figure 11-1 (above) portrays a queue table that contains two queues, and one exception queue:

- Queue1 — contains 10 messages.
- Queue2 — contains 7 messages.
- ExceptionQueue1 — contains 3 messages.

Basic Queuing

Basic Queuing — One Producer, One Consumer

At its most basic, one producer may enqueues different messages into one queue. Each message will be dequeued and processed once by one of the consumers. A message will stay in the queue until a consumer dequeues it or the message expires. A producer may stipulate a delay before the message is available to be consumed, and a time after which the message expires. Likewise, a consumer may wait when trying to dequeue a message if no message is available. Note that an agent program, or application, can act as both a producer and a consumer.

Basic Queueing — Many Producers, One Consumer

At a slightly higher level of complexity, many producers may enqueue messages into a queue, all of which are processed by one consumer.

Basic Queueing — Many Producers, Many Consumers of Discrete Messages

In this next stage, many producers may enqueue messages, each message being processed by a different consumer depending on type and correlation identifier. The figure below shows this scenario.

Illustrating Basic Queuing

Figure 11-2 (below) portrays a queue table that contains one queue into which messages are being enqueued and from which messages are being dequeued.

Producers

The figure indicates that there are 6 producers of messages, although only four are shown. This assumes that two other producers (P4 and P5) have the right to enqueue messages even though there are no messages enqueued by them at the moment portrayed by the figure. The figure shows:

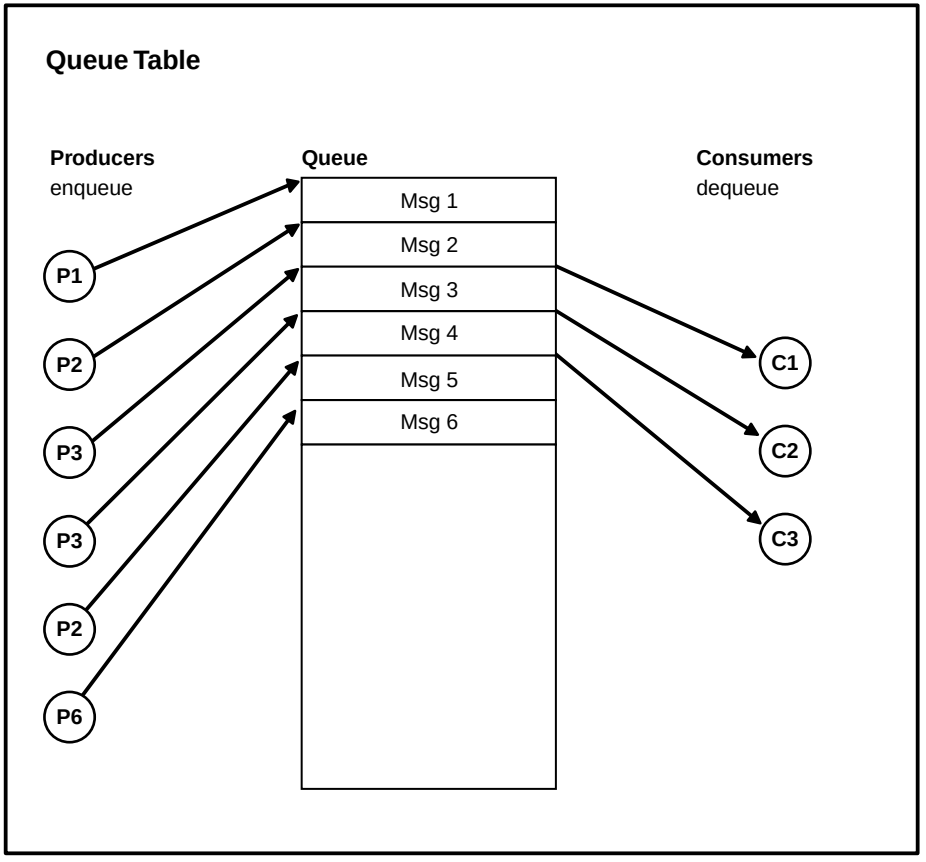
- that a single producer may enqueue one or more messages.
- that producers may enqueue messages in any sequence.

Consumers

According to the figure, there are 3 consumers of messages, representing the total population of consumers. The figure shows:

- messages are not necessarily dequeued in the order in which they are enqueued.
- messages may be enqueued without being dequeued.

Figure 11–2 Modeling Basic Queuing

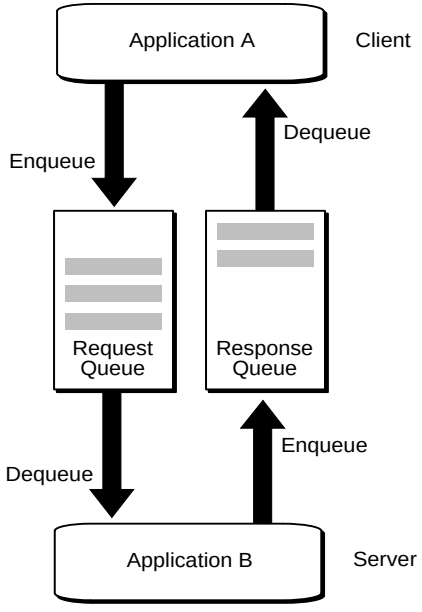


Illustrating Client-Server Communication Using AQ

The previous figure portrayed the enqueuing of multiple messages by a set of producers, and the dequeuing of messages by a set of consumers. What may not be readily evident in that sketch is the notion of *time*, and the advantages offered by Oracle AQ.

Client-Server applications normally execute in a synchronous manner, with all the disadvantages of that tight coupling described above. Figure 11–3 demonstrates the asynchronous alternative using AQ. In this example *Application B* (a server) provides service to *Application A* (a client) using a request/response queue.

Figure 11–3 Client-Server Communication Using AQ



1. *Application A* enqueues a request into the request queue.
2. *Application B* dequeues the request.
3. *Application B* processes the request.
4. *Application B* enqueues the result in the response queue.
5. *Application A* dequeues the result from the response queue.

In this way the client does not have to wait to establish a connection with the server, and the server dequeues the message at its own pace. When the server is finished processing the message, there is no need for the client to be waiting to receive the result. In this way a process of double-deferral frees both client and server.

Note: The various enqueue and dequeue operations are part of different transactions.

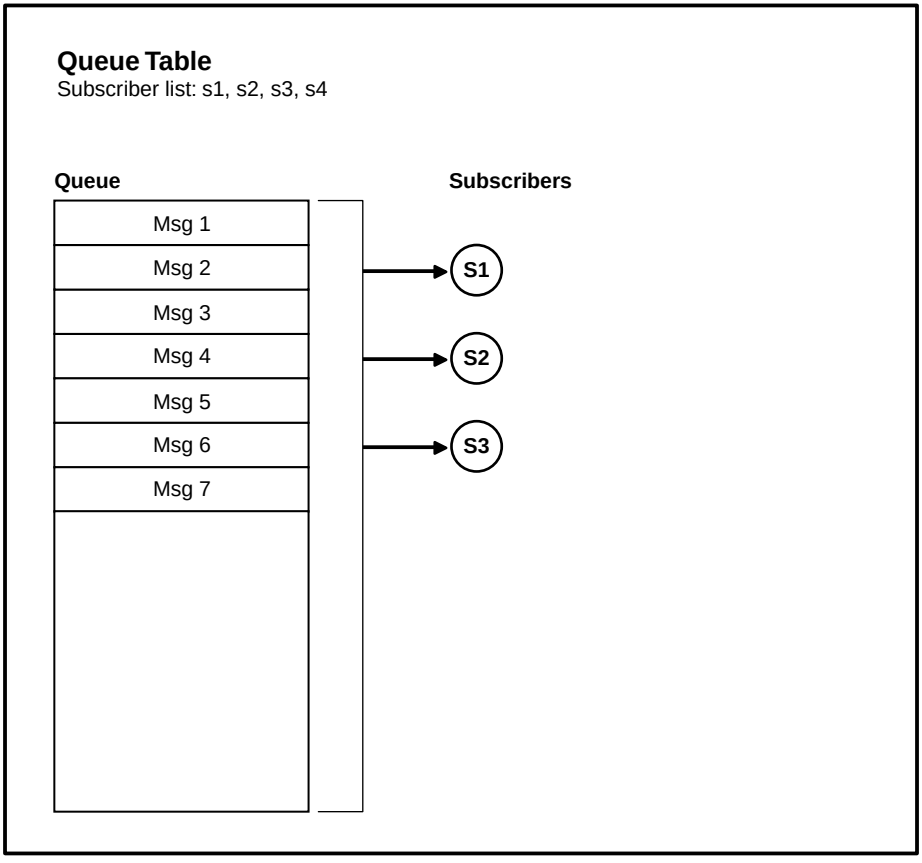
Multiple-Consumer Dequeuing of the Same Message

A message can only be enqueued into one queue at a time. If a producer had to insert the same message into several queues in order to reach different consumers, this would require management of a very large number of queues. Oracle AQ provides two mechanisms to allow for multiple consumers to dequeue the same message: *queue subscribers* and *message recipients*. The queue must reside in a queue table that is created with multiple consumer option to allow for subscriber and recipient lists. Each message remains in the queue until it is consumed by all its intended consumers.

Queue Subscribers Using this approach, multiple consumer-subscribers are associated with a queue. This will cause all messages enqueued in the queue to be made available to be consumed by each of the queue subscribers. The subscribers to the queue can be changed dynamically without any change to the messages or message producers. Subscribers to the queue are added and removed by using the Oracle AQ administrative package. The diagram below shows multiple producers enqueuing messages into queue, each of which is consumed by multiple consumer-subscribers.

Message Recipients A message producer can submit a list of recipients at the time a message is enqueued. This allows for a unique set of recipients for each message in the queue. The recipient list associated with the message overrides the subscriber list associated with the queue, if there is one. The recipients need not be in the subscriber list. However, recipients may be selected from among the subscribers.

Figure 11–4 Multiple-Consumer Dequeuing of the Same Message



Illustrating Multiple-Consumer Dequeuing of the Same Message

Figure 11–4 describes the case in which three consumers are all listed as subscribers of a queue. This is the same as saying that they all subscribe to all the messages that might ever be enqueued into that queue. The drawing illustrates a number of important points:

- The figure portrays the situation in which the 3 consumers are subscribers to 7 messages that have already been enqueued, and that they might become subscribers to messages that have not yet been enqueued.

- Every message will eventually be dequeued by every subscriber.
- There is no priority among subscribers. This means that there is no way of saying which subscriber will dequeue which message first, second, and so on. Or, put more formally: the order of dequeuing by subscribers is undetermined.
- We have no way of knowing from the figure about messages they might already have been dequeued, and which were then removed from the queue.

Figure 11–5 Communication Using a Multi-Consumer Queue

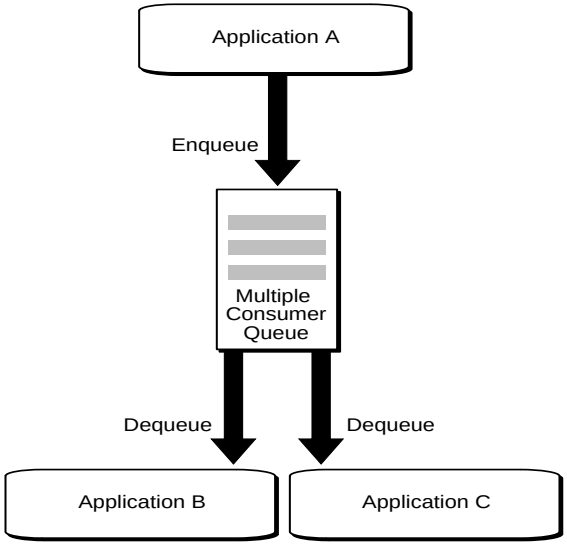
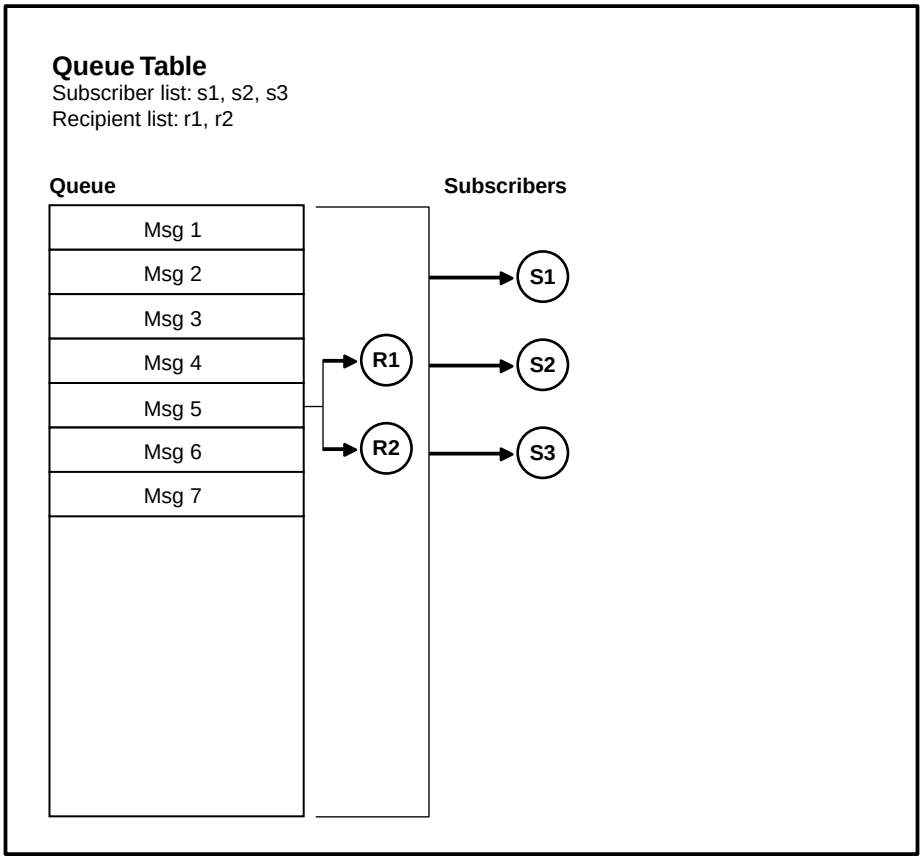


Figure 11–5 illustrates the same technology from a dynamic perspective. This examples concerns a scenario in which more than one application needs the result produced by an application. Every message enqueued by *Application A* is dequeued by *Application B* and *Application C*. To make this possible, the multiple consumer queue is specially configured with *Application B* and *Application C* as queue subscribers. Consequently, they are implicit recipients of every message placed in the queue.

Note: Queue subscribers can be applications or other queues.

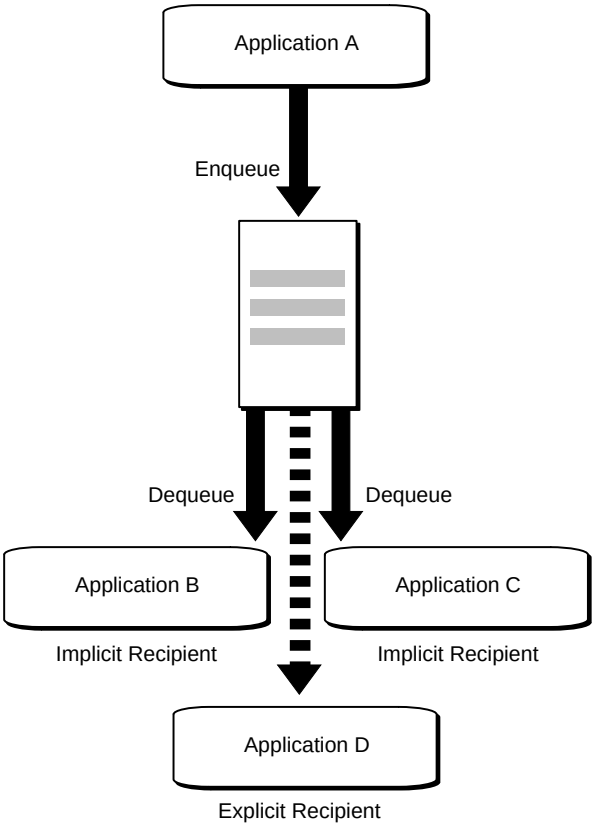
Figure 11–6 Dequeuing of Specified Messages by Specified Recipients



Illustrating Dequeuing of Specified Messages by Specified Recipients

Figure 11–6 shows how a message can be specified for one or more recipients. In this case, *Message 5* is specified to be dequeued by *Recipient-1* and *Recipient-2.* As described by the drawing, neither of the recipients is one of the 3 subscribers to the queue.

Figure 11–7 Explicit and Implicit Recipients of Messages



We earlier referred to *subscribers* as "implicit recipients" in that they are able to dequeue all the messages placed into a specific queue. This is like subscribing to a magazine and thereby implicitly gaining access to all its articles. The category of consumers that we have referred to as *recipients* may also be viewed as "explicit recipients" in that they are designated targets of particular messages.

Figure 11–7 shows how Oracle AQ can adjust dynamically to accommodate both kinds of consumers. In this scenario *Application B* and *Application C* are implicit recipients (subscribers). But messages can also be explicitly directed toward specific

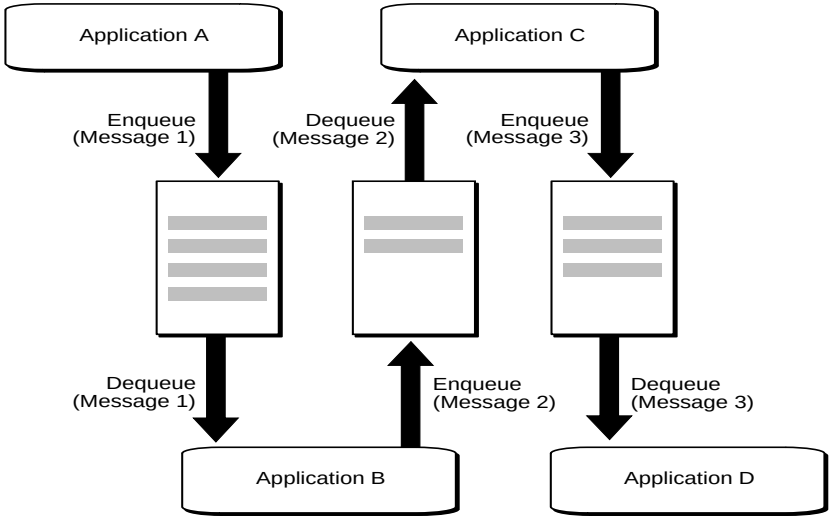
consumers (recipients) who may or may not be subscribers to the queue. The list of such recipients is specified in the enqueue call for that message and overrides the list of subscribers for that queue. In the figure, *Application D* is specified as the sole recipient of a message enqueued by *Application A*.

Note: Multiple producers may simultaneously enqueue messages aimed at different targeted recipients.

Illustrating the Implementation of Workflows using AQ

Figure 11–8 illustrates the use of AQ for implementing workflows, also knows as chained application transactions. It shows a workflow consisting of 4 steps performed by *Applications A*, *B*, *C* and *D*. The queues are used to buffer the flow of information between different processing stages of the business process. By specifying delay interval and expiration time for a message, a window of execution can be provided for each of the applications.

Figure 11–8 Implementing Workflows using AQ



From a workflow perspective, the passing of messages is a business asset above and beyond the value of the payload data. Hence, AQ supports the optional retention of messages for analysis of historical patterns and prediction of future trends. For instance, two of the three application scenarios at the head of the chapter are founded in an implementation of workflow analysis.

Note: The contents of the messages 1, 2 and 3 can be the same or different. Even when they are different, messages may contain parts of the of the contents of previous messages.

Message Propagation

Fanning-out of messages

In AQ, message recipients can be either consumers or other queues. If the message recipient is a queue, the actual recipients are determined by the subscribers to the queue (which may in turn be other queues). Thus it is possible to fan-out messages to a large number of recipients without requiring them all to dequeue messages from a single queue.

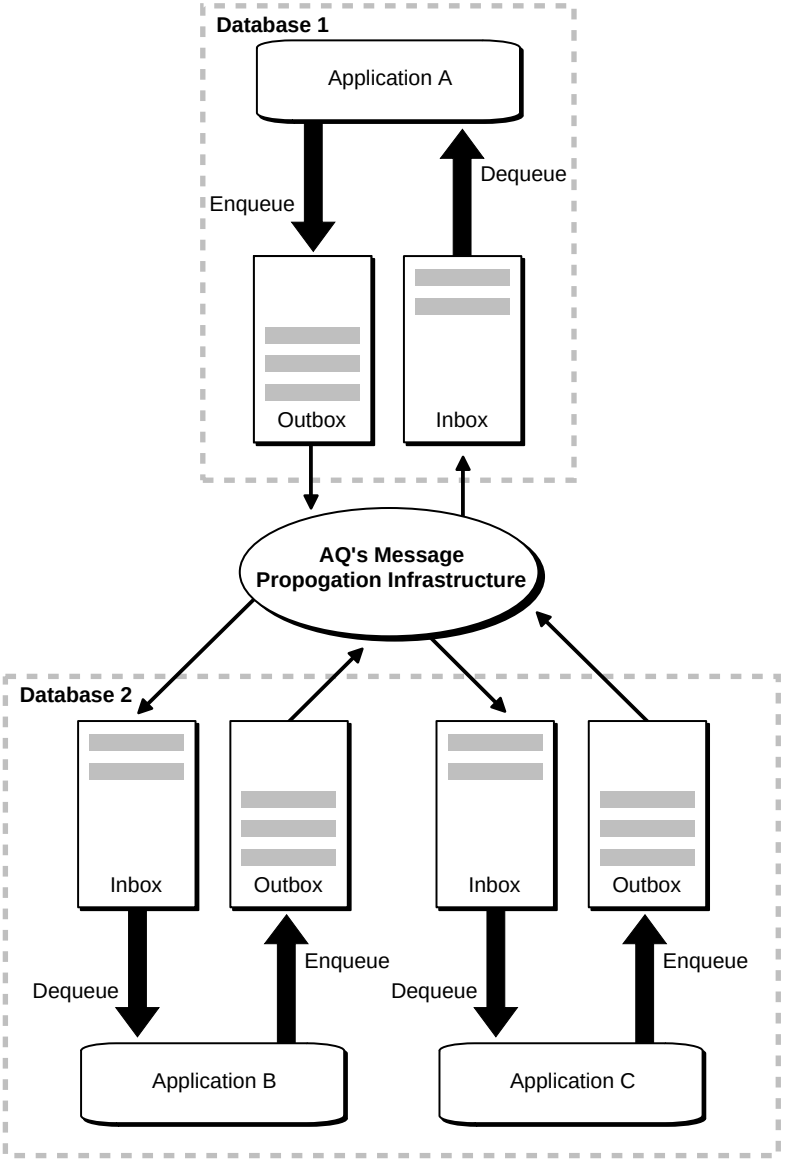
For example: A queue, *Source,* may have as its as its subscribers queues *dispatch1@dest1* and *dispatch2@dest2.* Queue *dispatch1@dest1* may in turn have as its subscribers the queues *outerreach1@dest3* and *outerreach2@dest4,* while queue *dispatch2@dest2* has as subscribers the queue *outerreach3@dest21* and *outerreach4@dest4.* In this way, messages enqueued in *Source* will be propagated to all the subscribers of four different queues.

Funneling-in of messages

Another use of queues as a message recipient is the ability to combine messages from different queues into a single queue. This process is sometimes described as "compositing"

For example, if queue *composite@endpoint* is a subscriber to both queues *funnel1@source1* and *funnel2@source2* then the subscribers to queue *composite@endpoint* can get all messages enqueued in those queues as well as messages enqueued directly into itself.

Figure 11–9 IMessage Propagation



llustration of Message Propagation

Figure 11–9 illustrates applications on different databases communicating via AQ. Each application has an inbox and an outbox for handling incoming and outgoing messages. An application enqueues a message into its outbox irrespective of whether the message has to be sent to an application that is local (on the same node) or remote (on a different node).

Likewise, an application is not concerned as to whether a message originates locally or remotely. In all cases, an application dequeues messages from its inbox.

Oracle AQ facilitates all this interchange, treating messages on the same basis.

Oracle Advanced Queuing by Example

Overview Summary

Oracle AQ by Example guides users by means of a step-by-step approach.

- Assign roles and privileges
- Create tables and queues
  - of object type
  - of RAW type
  - of object type for prioritized messages
  - of object type for multiple consumers
- Enqueue and dequeue of object type messages
- Enqueue and dequeue of object type messages using Pro*C/C++
- Enqueue and dequeue of object type messages using OCI
- Enqueue and dequeue of RAW type messages
- Enqueue and dequeue of RAW type messages using Pro*C/C++
- Enqueue and dequeue of RAW type messages using OCI
- Enqueue and dequeue by priority
- Dequeue messages after preview by criterion
- Enqueue and dequeue messages with time delay and expiration
- Enqueue and dequeue messages by correlation and message id using Pro*C/C++
- Enqueue and dequeue messages by correlation and message id using OCI
- Enqueue and dequeue of messages to/from a multiconsumer queue
  - stipulating subscribers and specific message recipients using PL/SQL
  - stipulating subscribers and specific message recipients using OCI
- Enqueue of messages for remote subscribers/recipients to a multiconsumer queue and propagation scheduling
- Unscheduling propagation

- Enqueue and Dequeue using Message Grouping
- Drop AQ Objects
  - by stopping and dropping queues and queue tables of object type
  - by stopping and dropping queues and queue tables of RAW type
- Revoke roles and privileges

Assign Roles and Privileges

```
/* Create user and grant privileges: */
CONNECT sys/change_on_install as sysdba;
CREATE user aq identified by AQ;
GRANT AQ_ADMINISTRATOR_ROLE TO aq;
GRANT CONNECT TO aq;
GRANT RESOURCE TO aq;
EXECUTE dbms_aqadm.grant_type_access('aq');
CONNECT aq/AQ;

SET ECHO ON;
SET SERVEROUTPUT ON;
```

Create Queue Tables and Queues

Create a queue table and queue of object type

```
/* Create a message type: */
CREATE type aq.message_type as object (
subject     VARCHAR2(30),
text        VARCHAR2(80));

/* Create a object type queue table and queue: */
EXECUTE dbms_aqadm.create_queue_table (
queue_table        => 'aq.msg',
queue_payload_type => 'aq.message_type');

EXECUTE dbms_aqadm.create_queue (
queue_name         => 'msg_queue',
queue_table        => 'aq.msg');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'msg_queue');
```

Create a queue table and queue of raw type

```
/* Create a RAW type queue table and queue: */
EXECUTE dbms_aqadm.create_queue_table (
queue_table          => 'aq.raw_msg',
queue_payload_type   => 'RAW');

EXECUTE dbms_aqadm.create_queue (
queue_name           => 'raw_msg_queue',
queue_table          => 'aq.raw_msg');

EXECUTE dbms_aqadm.start_queue (
queue_name           => 'raw_msg_queue');
```

Create a prioritized message queue table and queue

```
EXECUTE dbms_aqadm.create_queue_table (
queue_table        => 'aq.priority_msg',
sort_list          => 'PRIORITY,ENQ_TIME',
queue_payload_type => 'aq.message_type');

EXECUTE dbms_aqadm.create_queue (
queue_name         => 'priority_msg_queue',
queue_table        => 'aq.priority_msg');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'priority_msg_queue');
```

Create a multiple consumer queue table and queue

```
EXECUTE dbms_aqadm.create_queue_table (
queue_table        => 'aq.msg_multiple',
multiple_consumers => TRUE,
queue_payload_type => 'aq.message_type');

EXECUTE dbms_aqadm.create_queue (
queue_name         => 'msg_queue_multiple',
queue_table        => 'aq.msg_multiple');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'msg_queue_multiple');
```

Create a queue to demonstrate propagation

```
EXECUTE dbms_aqadm.create_queue (
queue_name        => 'another_msg_queue',
queue_table       => 'aq.msg_multiple');

EXECUTE dbms_aqadm.start_queue (
queue_name         => 'another_msg_queue');
```

Enqueue and Dequeue of Object Type Messages

To enqueue a single message without any other parameters specify the queue name and the payload.

```
/* Enqueue to msg_queue: */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
message := message_type('NORMAL MESSAGE',
'enqueued to msg_queue first.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

COMMIT;
END;
/

/* Dequeue from msg_queue: */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
dbms_aq.dequeue(queue_name => 'msg_queue',
              dequeue_options    => dequeue_options,
```

```
                message_properties => message_properties,
                payload            => message,
                msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
                                        ' ... ' || message.text );
COMMIT;
END;
/
```

Enqueue and Dequeue of Object Type Messages Using Pro*C/C++

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>
/* The header file generated by processing
object type 'aq.message_type': */
#include  "pceg.h"

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s\n", msg);
printf("\n% .800s \n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
message_type     *message = (message_type*)0;  /* payload */
char             user[60]="aq/AQ";  /* user logon password */
char             subject[30];  /* components of the */
char             txt[80];  /* payload type */

/* ENQUEUE and DEQUEUE to an OBJECT QUEUE */

/*  Connect to database: */
EXEC SQL CONNECT :user;

/* On an oracle error print the error number :*/
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");
```

```
/* Allocate memory for the host variable from the object cache : */
EXEC SQL ALLOCATE :message;

/* ENQUEUE */

strcpy(subject, "NORMAL ENQUEUE");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* Initialize the components of message : */
EXEC SQL OBJECT SET SUBJECT, TEXT OF  :message TO :subject, :txt;

/* Embedded PLSQL call to the AQ enqueue procedure : */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
 /* Bind the host variable 'message' to the payload: */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work */
EXEC SQL COMMIT;

printf("Enqueued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* Dequeue */

/* Embedded PLSQL call to the AQ dequeue procedure : */
EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* Return the payload into the host variable 'message':  */
dbms_aq.dequeue(queue_name => 'msg_queue',
```

```
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
 /* Commit work :*/
EXEC SQL COMMIT;

/* Extract the components of message: */
EXEC SQL OBJECT GET SUBJECT,TEXT FROM :message INTO :subject,:txt;

printf("Dequeued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);
}
```

Enqueue and Dequeue of Object Type Messages Using OCI

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
```

```
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* obtain TDO of message_type */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
       (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
       (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* prepare the message payload */
  mesg->subject = (OCIString *)0;
  mesg->data = (OCIString *)0;
  OCIStringAssignText(envhp, errhp,
```

```
    (CONST text *)"NORMAL MESSAGE", strlen("NORMAL MESSAGE"),
    &mesg->subject);
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"OCI ENQUEUE", strlen("OCI ENQUEUE"),
    &mesg->data);
  nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

  /* enqueue into the msg_queue */
  OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* dequeue from the msg_queue */
  OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
  printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
  OCITransCommit(svchp, errhp, (ub4) 0);

}
```

Enqueue and Dequeue of RAW Type Messages

```
/* Enqueue a message containing a RAW: */
DECLARE
      enqueue_options     dbms_aq.enqueue_options_t;
      message_properties  dbms_aq.message_properties_t;
      message_handle      RAW(16);
      message             RAW(4096);

BEGIN
      message :=  hextoraw(rpad('FF',4095,'FF'));
      dbms_aq.enqueue(queue_name => 'raw_msg_queue',
              enqueue_options    => enqueue_options,
              message_properties => message_properties,
              payload  => message,
              msgid              => message_handle);

      COMMIT;
END;
/* Dequeue from raw_msg_queue: */
DECLARE
      dequeue_options     dbms_aq.dequeue_options_t;
      message_properties  dbms_aq.message_properties_t;
```

```
      message_handle      RAW(16);
      message             RAW(4096);

BEGIN
 dbms_aq.dequeue(queue_name => 'raw_msg_queue',
dequeue_options    => dequeue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

COMMIT;
END;
```

Enqueue and Dequeue of RAW Type Messages using Pro*C/C++

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s\n", msg);
printf("\n% .800s \n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
OCIEnv         *oeh;   /* OCI Env handle */
OCIError       *err;   /* OCI Err handle */
OCIRaw         *message= (OCIRaw*)0;   /* payload */
ub1            message_txt[100];   /* data for payload */
char           user[60]="aq/AQ";   /* user logon password */
int            status;   /* returns status of the OCI call */

/* Enqueue and dequeue to a RAW queue */

/* Connect to database: */
EXEC SQL CONNECT :user;
```

```
/* On an oracle error print the error number: */
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* Get the OCI Env handle: */
if (SQLEnvGet(SQL_SINGLE_RCTX, &oeh) != OCI_SUCCESS)
{
printf(" error in SQLEnvGet \n");
exit(1);
}
/* Get the OCI Error handle: */
if (status = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
(ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0))
{
printf(" error in OCIHandleAlloc %d \n", status);
exit(1);
}

/* Enqueue */
/* The bytes to be put into the raw payload:*/
strcpy(message_txt, "Enqueue to a Raw payload queue ");

/* Assign bytes to the OCIRaw pointer :
Memory needs to be allocated explicitly to OCIRaw*: */
if (status=OCIRawAssignBytes(oeh, err, message_txt, 100,
 &message))
{
printf(" error in  OCIRawAssignBytes  %d \n", status);
exit(1);
}

/*  Embedded PLSQL call to the AQ enqueue procedure : */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
/* Bind the host variable message to the raw payload: */
dbms_aq.enqueue(queue_name => 'raw_msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
```

```
/* Commit work: */
EXEC SQL COMMIT;

/* Dequeue */
/*  Embedded PLSQL call to the AQ dequeue procedure :*/
EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* Return the raw payload into the host variable 'message':*/
dbms_aq.dequeue(queue_name => 'raw_msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;
}
```

Enqueue and Dequeue of RAW Type Messages using OCI

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  char  msg_text[100];
  OCIRaw  *mesg = (OCIRaw *)0;
  OCIRaw*deqmesg = (OCIRaw *)0;
```

```
  OCIInd   ind = 0;
  dvoid  *indptr = (dvoid *)&ind;
  inti;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* obtain the TDO of the RAW data type */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"SYS", strlen("SYS"),
       (CONST text *)"RAW", strlen("RAW"),
       (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* prepare the message payload */
  strcpy(msg_text, "Enqueue to a RAW queue");
  OCIRawAssignBytes(envhp, errhp, msg_text, strlen(msg_text), &mesg);

  /* enqueue the message into raw_msg_queue */
  OCIAQEnq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&mesg, (dvoid **)&indptr, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* dequeue the same message into C variable deqmesg */
  OCIAQDeq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
   mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&indptr, 0, 0);
  for (i = 0; i < OCIRawSize(envhp, deqmesg); i++)
```

```
    printf("%c", *(OCIRawPtr(envhp, deqmesg) + i));
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Enqueue and Dequeue of Messages by Priority

When two messages are enqued with the same priority, the message which was enqued earlier will be dequeued first. However, if two messages are of different priorities, the message with the lower value (higher priority) will be dequeued first.

```
/* Enqueue two messages with priority 30 and 5: */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
      message             aq.message_type;

BEGIN
message := message_type('PRIORITY MESSAGE',
enqued at priority 30.');

message_properties.priority := 30;

dbms_aq.enqueue(queue_name => 'priority_msg_queue',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

message := message_type('PRIORITY MESSAGE',
'Enqueued at priority 5.');

message_properties.priority := 5;

      dbms_aq.enqueue(queue_name => 'priority_msg_queue',
              enqueue_options    => enqueue_options,
              message_properties => message_properties,
              payload            => message,
              msgid              => message_handle);
END;
/
```

```
/* Dequeue from priority queue: */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
dbms_aq.dequeue(queue_name  => 'priority_msg_queue',
dequeue_options       => dequeue_options,
message_properties    => message_properties,
payload               => message,
msgid                 => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );

COMMIT;

dbms_aq.dequeue(queue_name => 'priority_msg_queue',
dequeue_options      => dequeue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );
COMMIT;
END;
/
```

On return, the second message with priority set to 5 will be retrieved before the message with priority set to 30 since priority takes precedence over enqueue time.

Dequeue of Messages after Preview by Criterion

An application can preview messages in browse mode or locked mode without deleting the message. The message of interest can then be removed from the queue.

```
/* Enqueue 6 messages to msg_queue
— GREEN, GREEN, YELLOW, VIOLET, BLUE, RED */
```

```
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
message := message_type('GREEN',
'GREEN enqueued to msg_queue first.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

message := message_type('GREEN',
'GREEN also enqueued to msg_queue second.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

message := message_type('YELLOW',
'YELLOW enqueued to msg_queue third.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

dbms_output.put_line ('Message handle: ' || message_handle);

message := message_type('VIOLET',
'VIOLET enqueued to msg_queue fourth.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);
```

```
message := message_type('BLUE',
'BLUE enqueued to msg_queue fifth.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

message := message_type('RED',
'RED enqueued to msg_queue sixth.');

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload              => message,
msgid                => message_handle);

      COMMIT;
END;
/

/* Dequeue in BROWSE mode until RED is found,
and remove RED from queue: */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
dequeue_options.dequeue_mode := dbms_aq.BROWSE;

      LOOP
dbms_aq.dequeue(queue_name          => 'msg_queue',
                dequeue_options     => dequeue_options,
                message_properties => message_properties,
                payload             => message,
                msgid               => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
                                        ' ... ' || message.text );

EXIT WHEN message.subject = 'RED';
```

```
END LOOP;

dequeue_options.dequeue_mode := dbms_aq.REMOVE;
dequeue_options.msgid        := message_handle;

dbms_aq.dequeue(queue_name => 'msg_queue',
dequeue_options    => dequeue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );

      COMMIT;
END;
/

/* Dequeue in LOCKED mode until BLUE is found,
and remove BLUE from queue: */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
dequeue_options.dequeue_mode := dbms_aq.LOCKED;

      LOOP

dbms_aq.dequeue(queue_name => 'msg_queue',
                dequeue_options    => dequeue_options,
                message_properties => message_properties,
                payload            => message,
                msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
          ' ... ' || message.text );

EXIT WHEN message.subject = 'BLUE';
      END LOOP;

dequeue_options.dequeue_mode := dbms_aq.REMOVE;
dequeue_options.msgid        := message_handle;
```

```
dbms_aq.dequeue(queue_name => 'msg_queue',
dequeue_options      => dequeue_options,
message_properties   => message_properties,
payload              => message,
msgid => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
' ... ' || message.text );

     COMMIT;
END;
/
```

Enqueue and Dequeue of Messages with Time Delay and Expiration

An enqueue can specify the time before which a message cannot be retrieved by a dequeue call. To do this, the producer (i.e the agent enqueuing the message) can also specify the time when a message expires, at which time the message is can use the parameter "delay" when enqueuing the message. The producer can also specify the time when a message expires, at which time the message is moved to an exception queue.

> **Note:** Expiration is calculated from the earliest dequeue time. So, if an application wants a message to be dequeued no earlier than a week from now, but no later than 3 weeks from now, this requires setting the expiration time for 2 weeks. This scenario is described in the following code segment.

```
/* Enqueue message for delayed availability: */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
message := message_type('DELAYED',
'This message is delayed one week.');
message_properties.delay := 7*24*60*60;
message_properties.expiration := 2*7*24*60*60;
```

```
dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload            => message,
msgid              => message_handle);

      COMMIT;
END;
```

Enqueue and Dequeue of Messages by Correlation and Message Id Using Pro*C/C++

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>
/* The header file generated by processing
object type 'aq.message_type': */
#include  "pceg.h"

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s\n", msg);
printf("\n% .800s \n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
OCIEnv            *oeh;  /* OCI Env Handle */
OCIError          *err;  /* OCI Error Handle */
message_type      *message = (message_type*)0; /* queue  payload */
OCIRaw            *msgid = (OCIRaw*)0;  /* message id */
ub1               msgmem[16]="";  /* memory for msgid */
char              user[60]="aq/AQ";  /* user login password */
char              subject[30];  /* components of */
char              txt[80];  /* message_type */
char              correlation1[30];  /* message correlation  */
char              correlation2[30];
int               status;  /* code returned by the OCI calls */
```

```
/* Dequeue by correlation and msgid */

/* Connect to the database: */
EXEC SQL CONNECT :user;
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* Allocate space in the object cache for the host variable: */
EXEC SQL ALLOCATE :message;

/* Get the OCI Env handle: */
if (SQLEnvGet(SQL_SINGLE_RCTX, &oeh) != OCI_SUCCESS)
{
 printf(" error in SQLEnvGet \n");
 exit(1);
}
/* Get the OCI Error handle: */
if (status = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
(ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0))
{
printf(" error in OCIHandleAlloc %d \n", status);
exit(1);
}


/* Assign memory for msgid:
Memory needs to be allocated explicitly to OCIRaw*: */
if (status=OCIRawAssignBytes(oeh, err, msgmem, 16, &msgid))
{
printf(" error in  OCIRawAssignBytes  %d \n", status);
exit(1);
}

/* First enqueue */

strcpy(correlation1, "1st message");
strcpy(subject, "NORMAL ENQUEUE1");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* Initialize the components of message: */
EXEC SQL OJECT SET SUBJECT, TEXT OF :message TO :subject, :txt;

/* Embedded PLSQL call to the AQ enqueue procedure: */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
```

```
enqueue_options      dbms_aq.enqueue_options_t;
BEGIN
/* Bind the host variable 'correlation1': to message correlation*/
message_properties.correlation := :correlation1;

/* Bind the host variable 'message' to payload and
 return message id into host variable 'msgid': */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => :msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;

printf("Enqueued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* Second enqueue */

strcpy(correlation2, "2nd message");
strcpy(subject, "NORMAL ENQUEUE2");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* Initialize the components of message: */
EXEC SQL OBJECT SET SUBJECT, TEXT OF :messsage TO :subject,:txt;

/* Embedded PLSQL call to the AQ enqueue procedure: */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
/* Bind the host variable 'correlation2':  to message correlaiton */
message_properties.correlation := :correlation2;

/* Bind the host variable 'message': to payload */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
```

```
msgid => msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;
printf("Enqueued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* First dequeue - by  correlation */

EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* Dequeue by  correlation in host variable 'correlation2': */
dequeue_options.correlation := :correlation2;

/* Return the payload into host variable 'message': */
dbms_aq.dequeue(queue_name => 'msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work : */
EXEC SQL COMMIT;

/* Extract the values of the components of message: */
EXEC SQL OBJECT GET SUBJECT, TEXT FROM :message INTO :subject,:txt;

printf("Dequeued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* SECOND DEQUEUE - by MSGID  */

EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
```

```
BEGIN
/* Dequeue by msgid in host variable 'msgid': */
dequeue_options.msgid := :msgid;

/* Return the payload into host variable 'message':  */
dbms_aq.dequeue(queue_name => 'msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;
}
```

Enqueue and Dequeue of Messages by Correlation and Message ID using OCI

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
  OCIEnv *envhp;
```

```
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* obtain TDO of message_type */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
       (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
       (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* prepare the message payload */
  mesg->subject = (OCIString *)0;
  mesg->data = (OCIString *)0;
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"NORMAL MESSAGE", strlen("NORMAL MESSAGE"),
```

```
    &mesg->subject);
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"OCI ENQUEUE", strlen("OCI ENQUEUE"),
    &mesg->data);
  nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

  /* enqueue into the msg_queue */
  OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* dequeue from the msg_queue */
  OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
       mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
  printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
  OCITransCommit(svchp, errhp, (ub4) 0);

}
```

Enqueue and Dequeue of Messages to/from a Multiconsumer Queue using PL/SQL

```
/* Create subscriber list: */
DECLARE
subscriber sys.aq$_agent;

/* Add subscribers RED and GREEN to the suscriber list: */
BEGIN
subscriber := sys.aq$_agent('RED', NULL, NULL);
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);

subscriber := sys.aq$_agent('GREEN', NULL, NULL);
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);
END;
/
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
recipients          dbms_aq.aq$_recipient_list_t;
message_handle      RAW(16);
message             aq.message_type;
```

```
/* Enqueue MESSAGE 1 for subscribers to the queue
i.e. for RED and GREEN: */
BEGIN
message := message_type('MESSAGE 1',
'This message is queued for queue subscribers.');

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

/* Enqueue MESSAGE 2 for specified recipients i.e. for RED and BLUE.*/
message := message_type('MESSAGE 2',
'This message is queued for two recipients.');
recipients(1) := sys.aq$_agent('RED', NULL, NULL);
recipients(2) := sys.aq$_agent('BLUE', NULL, NULL);
message_properties.recipient_list := recipients;

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

COMMIT;
END;
/
```

Note that RED is both a subscriber to the queue, as well as being a specified recipient of MESSAGE 2. By contrast, GREEN is only a subscriber to those messages in the queue (in this case, MESSAGE) for which no recipients have been specified. BLUE, while not a subscriber to the queue, is nevertheless specified to receive MESSAGE 2.

```
/* Dequeue messages from msg_queue_multiple: */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;
no_messages   exception;
pragma exception_init (no_messages, -25228);
```

```
BEGIN

dequeue_options.wait := dbms_aq.NO_WAIT;

/* Consumer BLUE will get MESSAGE 2: */
dequeue_options.consumer_name := 'BLUE';

LOOP

dbms_aq.dequeue(queue_name   => 'msg_queue_multiple',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload            => message,
          msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
         ' ... ' || message.text );

END LOOP;
EXCEPTION
WHEN no_messages THEN
dbms_output.put_line ('No more messages for BLUE');
COMMIT;
END;

BEGIN
/* Consumer RED will get MESSAGE 1 and MESSAGE 2: */
dequeue_options.consumer_name := 'RED';

LOOP
dbms_aq.dequeue(queue_name   => 'msg_queue_multiple',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload            => message,
          msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
                                        ' ... ' || message.text );
END LOOP;
EXCEPTION
WHEN no_messages THEN
dbms_output.put_line ('No more messages for RED');
COMMIT;
END;
```

```
BEGIN
/* Consumer GREEN will get MESSAGE 1: */
dequeue_options.consumer_name := 'GREEN';

LOOP
dbms_aq.dequeue(queue_name   => 'msg_queue_multiple',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload            => message,
          msgid              => message_handle);

dbms_output.put_line ('Message: ' || message.subject ||
          ' ... ' || message.text );
END LOOP;
EXCEPTION
WHEN no_messages THEN
dbms_output.put_line ('No more messages for GREEN');
COMMIT;
END;
/
```

Enqueue and Dequeue of Messages to/from a Multiconsumer Queue using OCI

```
#ifndef SL_ORACLE
#include <sl.h>
#endif

#ifndef OCI_ORACLE
#include <oci.h>
#endif

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
```

```
typedef struct null_message null_message;

int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;
  OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
  OCIAQAgent *agents[2];
  OCIAQDeqOptions *deqopt = (OCIAQDeqOptions *)0;
  ub4wait = OCI_DEQ_NO_WAIT;
  ub4 navigation = OCI_DEQ_FIRST_MSG;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
(dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
   52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
   52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
   (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);
```

```
  /* obtain TDO of message_type */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
       (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
       (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* prepare the message payload */
  mesg->subject = (OCIString *)0;
  mesg->data = (OCIString *)0;
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"MESSAGE 1", strlen("MESSAGE 1"),
    &mesg->subject);
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"mesg for queue subscribers",
    strlen("mesg for queue subscribers"), &mesg->data);
  nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

  /* enqueue MESSAGE 1 for subscribers to the queue i.e. for RED and GREEN */
  OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, 0,
       mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

  /* enqueue MESSAGE 2 for specified recipients i.e. for RED and BLUE */
  /* prepare message payload */
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"MESSAGE 2", strlen("MESSAGE 2"),
    &mesg->subject);
  OCIStringAssignText(envhp, errhp,
    (CONST text *)"mesg for two recipients",
    strlen("mesg for two recipients"), &mesg->data);

  /* allocate AQ message properties and agent descriptors */
  OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
  OCIDescriptorAlloc(envhp, (dvoid **)&agents[0],
  OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
  OCIDescriptorAlloc(envhp, (dvoid **)&agents[1],
  OCI_DTYPE_AQAGENT, 0, (dvoid **)0);

  /* prepare the recipient list, RED and BLUE */
  OCIAttrSet(agents[0], OCI_DTYPE_AQAGENT, "RED", strlen("RED"),
     OCI_ATTR_AGENT_NAME, errhp);
  OCIAttrSet(agents[1], OCI_DTYPE_AQAGENT, "BLUE", strlen("BLUE"),
     OCI_ATTR_AGENT_NAME, errhp);
  OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)agents, 2,
      OCI_ATTR_RECIPIENT_LIST, errhp);
```

```
 OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, msgprop,
      mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

 OCITransCommit(svchp, errhp, (ub4) 0);

 /* now dequeue the messages using different consumer names */
 /* allocate dequeue options descriptor to set the dequeue options */
 OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
    (dvoid **)0);

 /* set wait parameter to NO_WAIT so that the dequeue returns immediately */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&wait, 0,
    OCI_ATTR_WAIT, errhp);

 /* set navigation to FIRST_MESSAGE so that the dequeue resets the position */
 /* after a new consumer_name is set in the dequeue options     */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&navigation, 0,
    OCI_ATTR_NAVIGATION, errhp);

 /* dequeue from the msg_queue_multiple as consumer BLUE */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"BLUE", strlen("BLUE"),
    OCI_ATTR_CONSUMER_NAME, errhp);
 while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
 {
   printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
   printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
 }
 OCITransCommit(svchp, errhp, (ub4) 0);

 /* dequeue from the msg_queue_multiple as consumer RED */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"RED", strlen("RED"),
    OCI_ATTR_CONSUMER_NAME, errhp);
 while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
 {
   printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
   printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
 }
 OCITransCommit(svchp, errhp, (ub4) 0);

 /* dequeue from the msg_queue_multiple as consumer GREEN */
 OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,(dvoid *)"GREEN",strlen("GREEN"),
```

```
    OCI_ATTR_CONSUMER_NAME, errhp);
  while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
 mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
 == OCI_SUCCESS)
  {
    printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
  }
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Enqueue of Messages for remote subscribers/recipients to a Multiconsumer Queue and Propagation Scheduling

```
/* Create subscriber list: */
DECLARE
subscriber sys.aq$_agent;

/* Add subscribers RED and GREEN with different addresses  to the suscriber
list: */
BEGIN
/* Add subscriber RED that will dequeue messages from another_msg_queue queue
in the same datatbase */
subscriber := sys.aq$_agent('RED', 'another_msg_queue', NULL);
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);

/* Schedule propagation from msg_queue_multiple to other queues in the same
database: */
dbms_aqadm.schedule_propagation(queue_name => 'msg_queue_multiple');

/* Add subscriber GREEN that will dequeue messages from the msg_queue queue in
another database reached by the database link another_db.world */
subscriber := sys.aq$_agent('GREEN', 'msg_queue@another_db.world', NULL);
dbms_aqadm.add_subscriber(queue_name => 'msg_queue_multiple',
subscriber => subscriber);

/* Schedule propagation from msg_queue_multiple to other queues in the
database "another_database": */
BEGIN
dbms_aqadm.schedule_propagation(queue_name => 'msg_queue_multiple',
destination  => 'another_db.world');
```

```
END;
/

DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
recipients          dbms_aq.aq$_recipient_list_t;
message_handle      RAW(16);
message             aq.message_type;

/* Enqueue MESSAGE 1 for subscribers to the queue
i.e. for RED at address another_msg_queue and GREEN at address
msg_queue@another_db.world: */
BEGIN
message := message_type('MESSAGE 1',
'This message is queued for queue subscribers.');

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

/* Enqueue MESSAGE 2 for specified recipients i.e. for RED at address
another_msg_queue and BLUE.*/
message := message_type('MESSAGE 2',
'This message is queued for two recipients.');
recipients(1) := sys.aq$_agent('RED', 'another_msg_queue', NULL);
recipients(2) := sys.aq$_agent('BLUE', NULL, NULL);
message_properties.recipient_list := recipients;

dbms_aq.enqueue(queue_name => 'msg_queue_multiple',
enqueue_options    => enqueue_options,
message_properties => message_properties,
payload            => message,
msgid              => message_handle);

COMMIT;
END;
/
```

Note: RED at address another_msg_queue is both a subscriber to the queue, as well as being a specified recipient of MESSAGE 2. By contrast, GREEN at address msg_queue@another_db.world is only a subscriber to those messages in the queue (in this case, MESSAGE 1) for which no recipients have been specified. BLUE, while not a subscriber to the queue, is nevertheless specified to receive MESSAGE 2.

Unscheduling Propagation

```
/* unschedule propagation from msg_queue_multiple to the destination
another_db.world */
execute dbms_aqadm.unschedule_propagation(queue_name => 'msg_queue_multiple',
destination => 'another_db.world');
```

Enqueue and Dequeue using Message Grouping

```
CONNECT aq/aq

EXECUTE dbms_aqadm.create_queue_table (
queue_table        => 'aq.msggroup',
queue_payload_type => 'aq.message_type',
message_grouping => dbms_aqadm.TRANSACTIONAL);

EXECUTE dbms_aqadm.create_queue(
queue_name   => 'msggroup_queue',
queue_table        => 'aq.msggroup');

EXECUTE dbms_aqadm.start_queue(queue_name => 'msggroup_queue');

/* Enqueue three messages in each transaction */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

BEGIN
```

```
  /* loop through three times, committing after every iteration */
  FOR txnno in 1..3 LOOP

    /* loop through three times, enqueuing each iteration */
    FOR mesgno in 1..3 LOOP
      message := message_type('GROUP#' || txnno,
      'Message#' || mesgno ||  ' in group' || txnno);

      dbms_aq.enqueue(queue_name      => 'msggroup_queue',
              enqueue_options       => enqueue_options,
      message_properties     => message_properties,
      payload                => message,
      msgid                  => message_handle);
    END LOOP;
    /* commit the transaction */
    COMMIT;
  END LOOP;
END;
/

/* Now dequeue the messages as groups */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_type;

no_messages   exception;
end_of_group  exception;

pragma exception_init (no_messages, -25228);
pragma exception_init (end_of_group, -25235);

BEGIN
dequeue_options.wait    := DBMS_AQ.NO_WAIT;
dequeue_options.navigation := DBMS_AQ.FIRST_MESSAGE;

LOOP
   BEGIN
   dbms_aq.dequeue(queue_name      => 'msggroup_queue',
          dequeue_options    => dequeue_options,
          message_properties => message_properties,
          payload            => message,
          msgid              => message_handle);
```

```
  dbms_output.put_line ('Message: ' || message.subject ||
  ' ... ' || message.text );

  dequeue_options.navigation := DBMS_AQ.NEXT_MESSAGE;

  EXCEPTION
    WHEN end_of_group THEN
      dbms_output.put_line ('Finished processing a group of messages');
      COMMIT;
      dequeue_options.navigation := DBMS_AQ.NEXT_TRANSACTION;
  END;
END LOOP;
EXCEPTION
  WHEN no_messages THEN
    dbms_output.put_line ('No more messages');
END;
/
```

Drop AQ Objects

```
/* Cleans up all objects related to the object type: */
CONNECT aq/AQ;

EXECUTE dbms_aqadm.stop_queue (
queue_name => 'msg_queue');

EXECUTE dbms_aqadm.drop_queue (
queue_name => 'msg_queue');

EXECUTE dbms_aqadm.drop_queue_table (
queue_table => 'aq.msg');

/* Cleans up all objects related to the RAW type: */
EXECUTE dbms_aqadm.stop_queue (
        queue_name      => 'raw_msg_queue');

EXECUTE dbms_aqadm.drop_queue (
        queue_name      => 'raw_msg_queue');

EXECUTE dbms_aqadm.drop_queue_table (
            queue_table => 'aq.raw_msg');

/* Cleans up all objects related to the priority queue: */
EXECUTE dbms_aqadm.stop_queue (
```

```
        queue_name     => 'priority_msg_queue');

EXECUTE dbms_aqadm.drop_queue (
        queue_name     => 'priority_msg_queue');

EXECUTE dbms_aqadm.drop_queue_table (
queue_table    => 'aq.priority_msg');

/* Cleans up all objects related to the multiple-consumer queue: */
EXECUTE dbms_aqadm.stop_queue (
queue_name  => 'msg_queue_multiple');

EXECUTE dbms_aqadm.drop_queue (
queue_name  => 'msg_queue_multiple');

EXECUTE dbms_aqadm.drop_queue_table (
queue_table => 'aq.msg_multiple');

drop type aq.message_type;
```

Revoke Roles and Privileges

```
CONNECT sys/change_on_install;
drop user aq;
```

Oracle Advanced Queuing Reference

Reference Overview

This section contains a detailed description of the technical specifications:

- Init ora Parameter
- Data Structures
- Agent
- Message Properties
- Queue Options
- Operational Interface
- Administrative Interface
- Administration Topics
- Data Objects

INIT.ORA Parameter

AQ_TM_PROCESSES

A parameter called AQ_TM_PROCESSES should be specified in the init.ora PARAMETER *file* if you want to perform time monitoring on queue messages. This will be used for messages which have delay and expiration properties specified. This parameter can be set in a range from 0 to 10. Setting it to any other number will result in an error. If this parameter is set to 1, one queue monitor process will be created as a background process to monitor the messages. If the parameter is not specified, or is set to 0, the queue monitor process is not created. The administrative interfaces to start and stop the queue monitor are only valid if the queue monitor process is started as part of instance startup by specifying this parameter.

Parameter Name:	aq_tm_processes
Parameter Type:	integer
Parameter Class:	Dynamic
Allowable Values:	0 to 10
Syntax:	aq_tm_processes = <0 to 10>
Name of process:	ora_qmon_<oracle sid>

Example: aq_tm_processes = 1

JOB_QUEUE_PROCESSES

Propagation is handled by job queue (SNP) processes. The number of job queue processes started in an instance is controlled by the `init.ora` parameter `JOB_QUEUE_PROCESSES`. The default value of this parameter is 0. In order for message propagation to take place, this parameter must be set to at least 1. The DBA can set it to higher values if there are many queues from which the messages have to be propagated, or if there are many destinations to which the messages have to be propagated, or if there are other jobs in the job queue.

> **See Also:** *Oracle8 Reference* for complete details about `JOB_QUEUE_PROCESSES`.

COMPATIBLE

The `COMPATIBLE init.ora` parameter must be set to 8.0.4 in order to use the AQ propagation feature. Specifically, the `COMPATIBLE` parameter will be checked under the following three conditions:

1. An AQ agent's (see `sys.aq$_agent`) address field is specified in the `DBMS_AQADM.ADD_SUBSCRIBER` command.
2. An AQ agent's (see `sys.aq$_agent`) address field is specified in the `recipient_list` of `dbms_aq.message_properties_t`.
3. The `DBMS_AQADM.SCHEDULE_PROPAGATION` command is used.

Users can downgrade to 8.0.3 after using the 8.0.4 features by using

```
ALTER DATABASE RESET COMPATIBILITY
```

Users will not be allowed to restart the database in 8.0.3 compatible mode under the following conditions:

1. There are messages in queues that have not yet been propagated to their destinations.
2. There are propagation schedules that are still pending, in which case you may use the `DBMS_AQADM.UNSCHEDULE_PROPAGATION` command to remove the schedules.
3. There are queues that have remote subscribers (i.e. a non NULL address field in `sys.aq$_agent`), in which case you may remove remote subscribers by means of the `DBMS_AQADM.REMOVE_SUBSCRIBER` command.

See Also: For more details on compatibility, refer to the upgrade/downgrade section of the migration guide.

Data Structures

The following data structures are used in the *operational* and *administrative* interfaces.

Object name

Purpose:

Naming of database objects. This naming convention applies to queues, queue tables and object types.

Syntax:

```
object_name := VARCHAR2
object_name := [<schema_name>.]<name>
```

Usage:

Names for objects are specified by an optional schema name and a name. If the schema name is not specified then the current schema is assumed. The schema name and the name can each be up to 30 bytes long. However, queue names and queue table names can be a maximum of 24 bytes.

Type name

Purpose:

Defining queue types.

Syntax:

```
type_name := VARCHAR2
type_name := <object_type> | "RAW"
```

Usage:

Table 11–1 Type Name

Parameter	Description
<object_types>	For details on creating object types please refer to Server concepts manual. The maximum number of attributes in the object type is limited to 900.
"RAW"	To store payload of type RAW, AQ will create a queue table with a LOB column as the payload repository. The size of the payload is limited to 32K bytes of data. Because LOB columns are used for storing RAW payload, the AQ administrator can choose the LOB tablespace and configure the LOB storage by constructing a LOB storage string in the storage_clause parameter during queue table creation time.

Agent

Purpose:

To identify a producer or a consumer of a message.

Syntax:

```
TYPE sys.aq$_agent IS OBJECT (
    name            VARCHAR2(30),
    address         VARCHAR2(1024),
    protocol        NUMBER)
```

Usage:

Table 11–2 Agent

Parameter	Description
name	Name of a producer or consumer of a message.
address	Protocol specific address of the recipient. If the protocol is 0 (default) the address is of the form [schema.]queue[@dblink]
protocol	Protocol to interpret the address and propagate the message. The default (and currently the only supported) value is 0.

Message Properties

Purpose:

The *Message Properties* describe the information that is used by AQ to manage individual messages. These are set at enqueue time and their values are returned at dequeue time.

Syntax:

```
TYPE message_properties_t IS RECORD (
    priority          BINARY_INTEGER default 1,
    delay             BINARY_INTEGER default NO_DELAY,
    expiration        BINARY_INTEGER default NEVER,
    correlation       VARCHAR2(128) default NULL,
    attempts          BINARY_INTEGER,
    recipient_list    aq$_recipient_list_t,
    exception_queue   VARCHAR2(51) default NULL,
    enqueue_time      DATE,
    state             BINARY_INTEGER)

TYPE aq$_recipient_list_t IS TABLE OF sys.aq$_agent
    INDEX BY BINARY_INTEGER
```

Usage :

Table 11–3 Message properties

Parameter	Description
priority	Specifies the priority of the message. A smaller number indicates higher priority. The priority can be any number, including negative numbers.
delay	Specifies the delay of the enqueued message. The delay represents the number of seconds after which a message is available for dequeuing. Dequeuing by msgid overrides the delay specification. A message enqueued with delay set will be in the `WAITING` state, when the delay expires the messages goes to the `READY` state. `DELAY` processing requires the queue monitor to be started. Note that delay is set by the producer who enqueues the message. `NO_DELAY`: the message is available for immediate dequeuing. number: the number of seconds to delay the message.

Table 11–3 Message properties

Parameter	Description
expiration	Specifies the expiration of the message. It determines, in seconds, the duration the message is available for dequeuing. This parameter is an offset from the delay. Expiration processing requires the queue monitor to be running. NEVER: message will not expire. number: number of seconds message will remain in READY state. If the message is not dequeued before it expires, it will be moved to the exception queue in the EXPIRED state.
correlation	Specifies the identification supplied by the producer for a message at enqueuing.
attempts	Specifies the number of attempts that have been made to dequeue this message. This parameter can not be set at enqueue time.
recipient_list	For type definition please refer to section titled “Agent”. This parameter is only valid for queues which allow multiple consumers. The default recipients are the queue subscribers. This parameter is not returned to a consumer at dequeue time.
exception_queue	Specifies the name of the queue to which the message is moved if it cannot be processed successfully. Messages are moved in two cases: The number of unsuccessful dequeue attempts has exceeded *max_retries* or the message has expired. All messages in the exception queue are in the EXPIRED state. The default is the exception queue associated with the queue table. If the exception queue specified does not exist at the time of the move the message will be moved to the default exception queue associated with the queue table and a warning will be logged in the alert file. If the default exception queue is used the parameter will return a NULL value at dequeue time.
enqueue_time	Specifies the time the message was enqueued. This value is determined by the system and cannot be set by the user. This parameter can not be set at enqueue time.
state	Specifies the state of the message at the time of the dequeue. This parameter can not be set at enqueue time. 0: The message is ready to be processed. 1: The message delay has not yet been reached. 3: The message has been processed and is retained. 4: The message has been moved to the exception queue.

Queue Options

Enqueue options

Purpose:

To specify the options available for the enqueue operation.

Syntax:

```
TYPE enqueue_options_t IS RECORD (
    visibility          BINARY_INTEGER default ON_COMMIT,
    relative_msgid      RAW(16) default NULL,
    sequence_deviation  BINARY_INTEGER default NULL)
```

Usage:

Table 11–4

Parameter	Description
`visibility`	Specifies the transactional behavior of the enqueue request. `ON_COMMIT`: The enqueue is part of the current transaction. The operation is complete when the transaction commits. This is the default case. `IMMEDIATE`: The enqueue is not part of the current transaction. The operation constitutes a transaction on its own.
`relative_msgid`	Specifies the message identifier of the message which is referenced in the sequence deviation operation. This field is valid if and only if `BEFORE` is specified in *sequence_deviation*. This parameter will be ignored if sequence deviation is not specified.
`sequence_deviation`	Specifies if the message being enqueued should be dequeued before other message(s) already in the queue. `BEFORE`: The message is enqueued ahead of the message specified by *`relative_msgid`*. `TOP`: The message is enqueued ahead of any other messages. `NULL`: Default

Dequeue options

Purpose:

To specify the options available for the dequeue operation.

Syntax:

```
TYPE dequeue_options_t IS RECORD (
    consumer_name    VARCHAR2(30) default NULL,
    dequeue_mode     BINARY_INTEGER default REMOVE,
    navigation       BINARY_INTEGER default NEXT_MESSAGE,
    visibility       BINARY_INTEGER default ON_COMMIT,
    wait             BINARY_INTEGER default FOREVER
    msgid            RAW(16) default NULL,
    correlation      VARCHAR2(128) default NULL)
```

Usage

Table 11–5 DEQUEUE options

Parameter	Description
consumer_name	Name of the consumer. Only those messages matching the consumer name are accessed. If a queue is not set up for multiple consumers, this field should be set to NULL.
dequeue_mode	Specifies the locking behavior associated with the dequeue. BROWSE: Read the message without acquiring any lock on the message. This is equivalent to a select statement. LOCKED: Read and obtain a write lock on the message. The lock lasts for the duration of the transaction. This is equivalent to a select for update statement. REMOVE: Read the message and update or delete it. This is the default. The message can be retained in the queue table based on the retention properties.

Table 11–5 DEQUEUE options

Parameter	Description
navigation	Specifies the position of the message that will be retrieved. First, the position is determined. Second, the search criterion is applied. Finally, the message is retrieved. NEXT_MESSAGE: Retrieve the next message which is available and matches the search criteria. If the previous message belongs to a message group, AQ will retrieve the next available message which matches the search criteria and belongs to the message group. This is the default. NEXT_TRANSACTION: Skip the remainder of the current transaction group (if any) and retrieve the first message of the next transaction group. This option can only be used if message grouping is enabled for the current queue. FIRST_MESSAGE: Retrieves the first message which is available and matches the search criteria. This will reset the position to the beginning of the queue.
visibility	Specifies whether the new message is dequeued as part of the current transaction.The visibility parameter is ignored when using the BROWSE mode. ON_COMMIT: The dequeue will be part of the current transaction. This is the default case. IMMEDIATE: The dequeued message is not part of the current transaction. It constitutes a transaction on its own.
wait	Specifies the wait time if there is currently no message available which matches the search criteria. FOREVER: wait forever. This is the default. NO_WAIT: do not wait number: wait time in seconds
msgid	Specifies the message identifier of the message to be dequeued.
correlation	Specifies the correlation identifier of the message to be dequeued. Special pattern matching characters, such as the percent sign (%) and the underscore (_) can be used. If more than one message satisfies the pattern, the order of dequeuing is undetermined.

Operational Interface

The following interface calls are available to enqueue and dequeue messages from queues.

DBMS_AQ.ENQUEUE

Purpose:

Adds a message to the specified queue. In the simplest case, if the user wants to enqueue a message, without any other parameters, only the queue name and the payload have to be specified.

Syntax:

```
DBMS_AQ.ENQUEUE (
    queue_name          IN      VARCHAR2,
    enqueue_options     IN      enqueue_options_t,
    message_properties  IN      message_properties_t,
    payload             IN      "<type_name>",
    msgid               OUT     RAW)
```

Usage:

Table 11–6 **DBMS_AQ.ENQUEUE**

Parameter	Description
queue_name (IN VARCHAR2)	Specifies the name of the queue to which this message should be enqueued. The queue cannot be an exception queue.
enqueue_options (IN enqueue_option_t)	For the definition please refer to the section titled "ENQUEUE Options."
message_properties (IN message_properties_t)	For the definition please refer to the section titled "Message Properties."
payload (IN "<type_name>")	Not interpreted by Oracle AQ. The payload must be specified according to the specification in the associated queue table. NULL is an acceptable parameter. For the definition of <type_name> please refer to section titled "Type name"
msgid (OUT RAW)	The system generated identification of the message. This is a globally unique identifier that can be used to identify the message at dequeue time.

Using sequence deviation:

The *sequence_deviation* parameter in enqueue_options can be used to change the order of processing between two messages. The identity of the other message, if any, is specified by the enqueue_options parameter relative_msgid. The relationship is identified by the *sequence_deviation* parameter.

Specifying *sequence_deviation* for a message introduces some restrictions for the delay and priority values that can be specified for this message. The delay of this message has to be less than or equal to the delay of the message before which this message is to be enqueued. The priority of this message has to be greater than or equal to the priority of the message before which this message is to be enqueued.

DBMS_AQ.DEQUEUE

Purpose:

Dequeues a message from the specified queue.

Syntax:

```
DBMS_AQ.DEQUEUE (
    queue_name          IN      VARCHAR2,
    dequeue_options     IN      dequeue_options_t,
    message_properties  OUT     message_properties_t,
    payload             OUT    "<type_name>",
    msgid               OUT     raw)
```

Usage:

Table 11–7 DBMS_AQ.DEQUEUE

Parameter	Description
queue_name (IN VARCHAR2)	Specifies the name of the queue.
dequeue_options (IN dequeue_option_t)	For the definition please refer to the section titled "DEQUEUE Options."
message_properties (OUT message_properties_t)	For the definition please refer to the section titled "Message Properties."
payload (OUT "<type_name>")	Not interpreted by Oracle AQ. The payload must be specified according to the specification in the associated queue table. For the definition of <type_name> please refer to section titled "Type name"
msgid (OUT RAW)	The system generated identification of the message.

Search criteria and dequeue order for messages:

The search criteria for messages to be dequeued is determined by the *consumer_name, msgid* and *correlation* parameters in the dequeue_options. Msgid uniquely identifies the message to be dequeued. Correlation identifiers are application-defined identifiers that are not interpreted by AQ.

Only messages in the READY state are dequeued unless a msgid is specified.

The dequeue order is determined by the values specified at the time the queue table is created unless overridden by the msgid and correlation id in dequeue_options.

The database consistent read mechanism is applicable for queue operations. For example, a BROWSE call may not see a message that is enqueued after the beginning of the browsing transaction.

Navigating through a queue:

The default NAVIGATION parameter during dequeue is NEXT_MESSAGE. This means that subsequent dequeues will retrieve the messages from the queue based on the snapshot obtained in the first dequeue. In particular, a message that is enqueued after the first dequeue command will be processed only after processing all the remaining messages in the queue. This is usually sufficient when all the messages have already been enqueued into the queue, or when the queue does not have a priority-based ordering. However, applications must use the FIRST_MESSAGE navigation option when the first message in the queue needs to be processed by every dequeue command. This usually becomes necessary when a higher priority message arrives in the queue while messages already-enqueued are being processed.

Note: It may also be more efficient to use the FIRST_MESSAGE navigation option when there are messages being concurrently enqueued. If the FIRST_MESSAGE option is not specified, AQ will have to continually generate the snapshot as of the first dequeue command, leading to poor performance. If the FIRST_MESSAGE option is specified, AQ will use a new snapshot for every dequeue command.

Dequeue by Message Grouping:

Messages enqueued in the same transaction into a queue that has been enabled for message grouping will form a group. If only one message is enqueued in the transaction, this will effectively form a group of one message. There is no upper limit to the number of messages that can be grouped in a single transaction.

In queues that have not been enabled for message grouping, a dequeue in LOCKED or REMOVE mode locks only a single message. By contrast, a dequeue operation that seeks to dequeue a message that is part of a group will lock the entire group. This is useful when all the messages in a group need to be processed as an atomic unit.

When all the messages in a group have been dequeued, the dequeue returns an error indicating that all messages in the group have been processed. The application can then use the NEXT_TRANSACTION to start dequeuing messages from the next available group. In the event that no groups are available, the dequeue will time-out after the specified WAIT period.

Enumerated Constants in the Operational Interface

When using enumerated constants such as BROWSE, LOCKED, REMOVE, the PL/SQL constants need to be specified with the scope of the packages defining it. All types associated with the operational interfaces have to be prepended with dbms_aq. For example:

```
dbms_aq.BROWSE
```

Table 11–8 Enumerated types in the operational interface

Parameter	Options
visibility	IMMEDIATE, ON_COMMIT
mode	BROWSE, LOCKED, REMOVE
navigation	FIRST_MESSAGE, NEXT_MESSAGE, NEXT_TRANSACTION
state	WAITING, READY, PROCESSED, EXPIRED
sequence_deviation	BEFORE, TOP
wait	FOREVER, NO_WAIT
delay	NO_DELAY
expiration	NEVER

Administrative Interface

Configuration information can be managed through procedures in the DBMS_AQADM package. Because incorrect usage of the administration interface can have substantial performance impact on the database system, the administration interface should be treated as privileged commands, and only the designated queue administrator or privileged users should be granted access to the administration package. Initially, only SYS has the execution privilege for the procedures in DBMS_AQADM and DBMS_AQ.

Privileges and access control

Access to AQ operations are granted to users through roles. These roles provide execution privileges on the AQ procedures. Currently, we do not support fine grained access control at the database object level. This implies that a user with the AQ_USER_ROLE can enqueue and dequeue to any queue in the system.

Administrator role AQ_ADMINISTRATOR_ROLE grants execute privileges to procedures in the DBMS_AQADM and DBMS_AQ packages. These include all the administrative and operational interfaces. The user 'SYS' must grant the AQ_ADMINISTRATOR_ROLE to the AQ administrator.

User role AQ_USER_ROLE grants execute privileges to procedures in the DBMS_AQ packages. These include all the operational interfaces. The AQ administrator must grant the AQ_USER_ROLE TO AQ users.

Access to AQ object types The procedure *grant_type_access* must first be executed by the user 'SYS' to grant access for AQ object types to the AQ administrator. The AQ administrator can then execute this procedure to grant access for AQ object types to other AQ users. The procedure needs to be executed if the user wishes to perform any administrative operation involving a multiple consumer queue. These include CREATE_QUEUE_TABLE, CREATE_QUEUE, ADD_SUBSCRIBER and REMOVE_SUBSCRIBER.

Syntax:

```
PROCEDURE grant_type_access (user_name IN VARCHAR2);
```

Calling DBMS_AQ from a PL/SQL function or procedure If you wish to call DBMS_AQ from a PL/SQL function or procedure, you will need to have been explicitly granted the EXECUTE privilege. You cannot inherit this right from either the AQ_USER_ROLE or the AQ_ADMINISTRATOR_ROLE.

Syntax:

```
GRANT EXECUTE ON DBMS_AQ TO <user>;
```

Example

1. Scott is appointed as the AQ administrator.

```
CONNECT sys/change_on_install
GRANT AQ_ADMINISTRATOR_ROLE to scott with admin option;
execute dbms_aqadm.grant_type_access('scott');
```

2. Scott lets Jones use AQ.

```
CONNECT scott/tiger
GRANT AQ_USER_ROLE to jones;
```

3. Jones wishes to create queue tables that are enabled for multiple dequeues.

```
CONNECT scott/tiger
execute dbms_aqadm.grant_type_access('jones');
```

DBMS_AQADM.CREATE_QUEUE_TABLE

Purpose: Create a queue table for messages of a pre-defined type. The sort keys for dequeue ordering, if any, need to be defined at table creation time. The following objects are created at this time:

1. The default exception queue associated with the queue table called `aq$_<queue_table_name>_e`.
2. A read-only view which is used by AQ applications for querying queue data called `aq$<queue_table_name>`.
3. An index for the queue monitor operations called `aq$_<queue_table_name>_t`.
4. An index or an index organized table (IOT) in the case of multiple consumer queues for dequeue operations called `aq$_<queue_table_name>_i`.

Syntax:

```
DBMS_AQADM.CREATE_QUEUE_TABLE (
    queue_table        IN      VARCHAR2,
    queue_payload_type IN      VARCHAR2,
    storage_clause     IN      VARCHAR2 default NULL,
    sort_list          IN      VARCHAR2 default NULL,
    multiple_consumers IN      BOOLEAN default FALSE,
    message_grouping   IN      BINARY_INTEGER default NONE,
    comment            IN      VARCHAR2 default NULL,
    auto_commit        IN      BOOLEAN default TRUE)
```

Usage

Table 11–9 DBMS_AQADM.CREATE_QUEUE_TABLE

Parameter	Description
`queue_table` `(IN VARCHAR2)`	Specifies the name of a queue table to be created.
`queue_payload_type` `(IN VARCHAR2)`	Specifies the type of the user data stored. Please see section entitled "Type name" for valid values for this parameter.

Table 11–9 (Cont.) DBMS_AQADM.CREATE_QUEUE_TABLE

Parameter	Description
storage_clause	Specifies the storage parameter. The storage parameter will be
(IN VARCHAR2)	included in the 'CREATE TABLE' statement when the queue table is created. The storage parameter can be made up of any combinations of the following parameters:
	PCTFREE, PCTUSED, INITRANS, MAXTRANS, TABLEPSACE, LOB and a table storage clause.
	Please refer to the SQL reference guide for the usage of these parameters.
sort_list	Specifies the columns to be used as the sort key in ascending order.
(IN VARCHAR2)	*Sort_list* has the following format: '<sort_column_1>,<sort_column_2>'.
	The allowed column names are *priority* and *enq_time*. If both columns are specified then <sort_column_1> defines the most significant order.
	Once a queue table is created with a specific ordering mechanism, all queues in the queue table inherit the same defaults. The order of a queue table cannot be altered once the queue table has been created.
	If no sort list is specified all the queues in this queue table will be sorted by the enqueue time in ascending order. This order is equivalent to FIFO order.
	Even with the default ordering defined, a dequeuer is allowed to choose a message to dequeue by specifying its *msgid* or *correlation*. *Msgid*, *correlation* and *sequence_deviation* take precedence over the default dequeueing order if they are specified.
multiple_consumers (IN BOOLEAN)	FALSE: Queues created in the table can only have one consumer per message. This is the default.
	TRUE: Queues created in the table can have multiple consumers per message. The user must have been granted type access by executing the *grant_type_access* procedure.
message_grouping	Specifies the message grouping behavior for queues created in the table.
(IN BINARY_INTEGER)	NONE: Each message is treated individually.
	TRANSACTIONAL: Messages enqueued as part of one transaction are considered part of the same group and can be dequeued as a group of related messages.
comment (IN VARCHAR2)	Specifies the user-specified description of the queue table. This user comment will be added to the queue catalog.

Table 11–9 (Cont.) DBMS_AQADM.CREATE_QUEUE_TABLE

Parameter	Description
auto_commit (IN BOOLEAN)	TRUE: causes the current transaction, if any, to commit before the CREATE_QUEUE_TABLE operation is carried out. The CREATE_QUEUE_TABLE operation becomes persistent when the call returns. This is the default. FALSE: The operation is part of the current transaction and will become persistent only when the caller issues a commit.

DBMS_AQADM.CREATE_QUEUE

Purpose: Create a queue in the specified queue table. All queue names must be unique within a schema. Once a queue is created with CREATE_QUEUE, it can be enabled by calling START_QUEUE. By default, the queue is created with both enqueue and dequeue disabled.

Syntax:

```
DBMS_AQADM.CREATE_QUEUE (
    queue_name          IN      VARCHAR2,
    queue_table         IN      VARCHAR2,
    queue_type          IN      BINARY_INTEGER default
                                NORMAL_QUEUE,
    max_retries         IN      NUMBER default 0,
    retry_delay         IN      NUMBER default 0,
    retention_time      IN      NUMBER default 0,
    dependency_tracking IN      BOOLEAN default FALSE,
    comment             IN      VARCHAR2 default NULL,
    auto_commit         IN      BOOLEAN default TRUE)
```

Usage:

Table 11–10 DBMS_AQADM.CREATE_QUEUE

Parameter	Description
queue_name (IN VARCHAR2)	Specifies the name of the queue that is to be created.
queue_table (IN VARCHAR2)	Specifies the name of the queue table that will contain the queue.

Table 11–10 (Cont.) DBMS_AQADM.CREATE_QUEUE

Parameter	Description
queue_type (IN BINARY_INTEGER)	Specifies whether the queue being created is an exception queue or a normal queue. NORMAL_QUEUE: The queue is a normal queue. This is the default. EXCEPTION_QUEUE: It is an exception queue. Only the dequeue operation is allowed on the exception queue.
max_retries (IN NUMBER)	Limits the number of times a dequeue with the REMOVE mode can be attempted on a message. The count is incremented when the application issues a rollback after executing the dequeue. The message is moved to the exception queue when it is reaches its *max_retries*. Default is 0, which means no retry is allowed.
retry_delay (IN NUMBER)	Specifies the delay time, in seconds before this message is scheduled for processing again after an application rollback. The default is 0, which means the message can be retried as soon as possible. This parameter will have no effect if *max_retries* is set to 0. *Retry_delay* cannot be specified with multiple consumer queues.
retention_time (IN NUMBER)	Specifies the number of seconds for which a message will be retained in the queue table after being dequeued from the queue. INFINITE: Message will be retained forever. number: Number of seconds for which to retain the messages. The default is 0, i.e. no retention.
dependency_tracking (IN BOOLEAN)	Reserved for future use. FALSE: This is the default. TRUE: Not permitted in this release.
comment (IN VARCHAR2)	User-specified description of the queue. This user comment will be added to the queue catalog.
auto_commit (IN BOOLEAN)	TRUE: Causes the current transaction, if any, to commit before the CREATE_QUEUE operation is carried out. The CREATE_QUEUE operation becomes persistent when the call returns. This is the default. FALSE: The operation is part of the current transaction and will become persistent only when the caller issues a commit.

DBMS_AQADM.DROP_QUEUE_TABLE

Purpose:

Drop an existing queue table. All the queues in a queue table have to be stopped and dropped before the queue table can be dropped.

Syntax:

```
DBMS_AQADM.DROP_QUEUE_TABLE (
    queue_table       IN    VARCHAR2,
    force             IN    BOOLEAN default FALSE,
    auto_commit       IN    BOOLEAN default TRUE)
```

Usage:

Table 11–11 DBMS_AQADM.DROP_QUEUE_TABLE

Parameter	Description
queue_table (IN VARCHAR2)	Specifies the name of a queue table to be dropped.
force (IN BOOLEAN)	FALSE: The operation will not succeed if there are any queues in the table.This is the default. TRUE: All queues in the table are stopped and dropped automatically.
auto_commit (IN BOOLEAN)	TRUE: Causes the current transaction, if any, to commit before the DROP_QUEUE_TABLE operation is carried out. The DROP_QUEUE_TABLE operation becomes persistent when the call returns. This is the default. FALSE: The operation is part of the current transaction and will become persistent only when the caller issues a commit.

DBMS_AQADM.DROP_QUEUE

Purpose:

Drops an existing queue. DROP_QUEUE is not allowed unless STOP_QUEUE has been called to disable the queue for both enqueuing and dequeuing. All the queue data is deleted as part of the drop operation.

Syntax:

```
DBMS_AQADM.DROP_QUEUE (
    queue_name        IN          VARCHAR2,
    auto_commit       IN          BOOLEAN default TRUE)
```

Usage:

Table 11–12 DBMS_AQADM.DROP_QUEUE

Parameter	Description
queue_name (IN VARCHAR2)	Specifies the name of the queue that is to be dropped.
auto_commit (IN BOOLEAN)	TRUE: Causes the current transaction, if any, to commit before the DROP_QUEUE operation is carried out. The DROP_QUEUE operation becomes persistent when the call returns. This is the default. FALSE: The operation is part of the current transaction and will become persistent only when the caller issues a commit.

DBMS_AQADM.ALTER_QUEUE

Purpose:

Alter existing properties of a queue. Only max_retries, retry_delay, and retention_time can be altered.

Syntax:

```
DBMS_AQADM.ALTER_QUEUE (
    queue_name        IN    VARCHAR2,
    max_retries       IN    NUMBER default NULL,
    retry_delay       IN    NUMBER default NULL,
    retention_time    IN    NUMBER default NULL,
    auto_commit       IN    BOOLEAN default TRUE)
```

Usage:

Table 11–13 DBMS_AQADM.ALTER_QUEUE

Parameter	Description
queue_name (IN VARCHAR2)	Specifies the name of the queue that is to be altered.
max_retries (IN NUMBER)	Limits the number of times a dequeue with REMOVE mode can be attempted on a message. The count is incremented when the application issues a rollback after executing the dequeue. If the time at which one of the retries has passed the expiration time, no further retries will be attempted. Default is NULL which means that the value will not be altered.
retry_delay (IN NUMBER)	Specifies the delay time in seconds before this message is scheduled for processing again after an application rollback. The default is NULL which means that the value will not be altered.
retention_time (IN NUMBER)	Specifies the retention time in seconds for which a message will be retained in the queue table after being dequeued. The default is NULL which means that the value will not be altered.
auto_commit (IN BOOLEAN)	TRUE: Causes the current transaction, if any, to commit before the ALTER_QUEUE operation is carried out. The ALTER_QUEUE operation become persistent when the call returns. This is the default. FALSE: The operation is part of the current transaction and will become persistent only when the caller issues a commit.

DBMS_AQADM.START_QUEUE

Purpose:

Enables the specified queue for enqueuing and/or dequeueing. After creating a queue the administrator must use START_QUEUE to enable the queue. The default is to enable it for both ENQUEUE and DEQUEUE. Only dequeue operations are allowed on an exception queue. This operation takes effect when the call completes and does not have any transactional characteristics.

Syntax:

```
DBMS_AQADM.START_QUEUE (
    queue_name      IN      VARCHAR2,
    enqueue         IN      BOOLEAN default TRUE,
    dequeue         IN      BOOLEAN default TRUE)
```

Usage

Table 11–14 DBMS_AQADM.START_QUEUE

Parameter	Description
queue_name	Specifies the name of the queue to be enabled.
(IN VARCHAR2)	
enqueue	Specifies whether ENQUEUE should be enabled on this queue.
(IN BOOLEAN)	TRUE: Enable ENQUEUE. This is the default.
	FALSE: Do not alter the current setting.
dequeue	Specifies whether DEQUEUE should be enabled on this queue.
(IN BOOLEAN)	TRUE: Enable DEQUEUE. This is the default.
	FALSE: Do not alter the current setting.

DBMS_AQADM.STOP_QUEUE

Purpose:

Disables enqueuing and/or dequeuing on the specified queue. By default, it disables both ENQUEUEs or DEQUEUEs. A queue cannot be stopped if there are outstanding transactions against the queue. This operation takes effect when the call completes and does not have any transactional characteristics.

Syntax:

```
DBMS_AQADM.STOP_QUEUE (
    queue_name      IN      VARCHAR2,
    enqueue         IN      BOOLEAN default TRUE,
    dequeue         IN      BOOLEAN default TRUE,
    wait            IN      BOOLEAN default TRUE)
```

Usage:

Table 11–15 DBMS_AQADM.STOP_QUEUE

Parameter	Description
queue_name	Specifies the name of the queue to be disabled.
(IN VARCHAR2)	
enqueue	Specifies whether ENQUEUE should be disabled on this queue.
(IN BOOLEAN)	TRUE: Disable ENQUEUE. This is the default.
	FALSE: Do not alter the current setting.
dequeue	Specifies whether DEQUEUE should be disabled on this queue.
(IN BOOLEAN)	TRUE: Disable DEQUEUE. This is the default.
	FALSE: Do not alter the current setting.
wait	The *wait* parameter allows you to specify whether to wait for the completion of outstanding transactions.
(IN BOOLEAN)	TRUE: Wait if there are any outstanding transactions. In this state no new transactions are allowed to enqueue to or dequeue from this queue.
	FALSE: Return immediately either with a success or an error.

DBMS_AQADM.ADD_SUBSCRIBER

Purpose:

Add a default subscriber to a queue. A program can enqueue messages to a specific list of recipients or to the default list of subscribers. This operation will only succeed on queues that allow multiple consumers. This operation takes effect immediately and the containing transaction is committed. Enqueue requests that are executed after the completion of this call will reflect the new behavior. The user must have been granted type access by executing the *grant_type_access* procedure.

Syntax:

```
DBMS_AQADM.ADD_SUBSCRIBER(
    queue_name     IN   VARCHAR2,
    subscriber     IN   sys.aq$_agent)
```

Usage:

Table 11–16 DBMS_AQADM.ADD_SUBSCRIBER

Parameter	Description
queue_name	Specifies the name of the queue.
(IN VARCHAR2)	
subscriber	See definition in section titled 'Agent'.
(IN aq$_agent)	

DBMS_AQADM.REMOVE_SUBSCRIBER

Purpose:

Remove a default subscriber from a queue. This operation takes effect immediately and the containing transaction is committed. All references to the subscriber in existing messages are removed as part of the operation. The user must have been granted type access by executing the *grant_type_access* procedure.

Syntax:

```
DBMS_AQADM.REMOVE_SUBSCRIBER(
    queue_name          IN          VARCHAR2,
    subscriber          IN          sys.aq$_agent)
```

Usage:

Table 11–17

Parameter	Description
queue_name	Specifies the name of the queue.
(IN VARCHAR2)	
subscriber	See definition in section titled ‘Agent’.
(IN aq$_agent)	

DBMS_AQADM.SCHEDULE_PROPAGATION

Purpose:

Schedule propagation of messages from a queue to a destination identified by a specific dblink. Messages may also be propagated to other queues in the same database by specifying a NULL destination. If a message has multiple recipients at the same destination in either the same or different queues the message will be propagated to all of them at the same time.

Syntax:

```
DBMS_AQADM.SCHEDULE_PROPAGATION(
    src_queue_name  IN    VARCHAR2,
    destination     IN    VARCHAR2 default NULL
    start_time      IN    DATE default SYSDATE,
    duration        IN    NUMBER default NULL,
    next_time       IN    VARCHAR2 default NULL,
    latency         IN    NUMBER default 60)
```

Usage:

Table 11–18 DBMS_AQADM.SCHEDULE_PROPAGATION

Parameter	Description
src_queue_name (IN VARCHAR2)	Specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the administrative user.
destination (IN VARCHAR2)	Specifies the destination dblink. Messages in the source queue for recipients at this destination will be propagated. If it is NULL, the destination is the local database and messages will be propagated to other queues in the local database. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.
start_time (IN DATE)	Specifies the initial start time for the propagation window for messages from the source queue to the destination.
duration (IN NUMBER)	Specifies the duration of the propagation window in seconds. A NULL value means the propagation window is forever or until the propagation is unscheduled.

Table 11–18 DBMS_AQADM.SCHEDULE_PROPAGATION

Parameter	Description
next_time (IN VARCHAR2)	date function to compute the start of the next propagation window from the end of the current window. If this value is NULL, propagation will be stopped at the end of the current window. For example, to start the window at the same time every day, next_time should be specified as 'SYSDATE + 1 - duration / 86400'.
latency (IN NUMBER)	maximum wait, in seconds, in the propagation window for a message to be propagated after it is enqueued. For example, if the latency is 60 seconds, then during the propagation window, if there are no messages to be propagated, messages from that queue for the destination will not be propagated for at least 60 more seconds. It will be at least 60 seconds before the queue will be checked again for messages to be propagated for the specified destination. If the latency is 600, then the queue will not be checked for 10 minutes and if the latency is 0, then a job queue process will be waiting for messages to be enqueued for the destination and as soon as a message is enqueued it will be propagated.

DBMS_AQADM.UNSCHEDULE_PROPAGATION

Purpose:

Unscheduled previously scheduled propagation of messages from a queue to a destination identified by a specific dblink.

Syntax:

```
DBMS_AQADM.UNSCHEDULE_PROPAGATION(
    src_queue_name   IN    VARCHAR2,
    destination      IN    VARCHAR2 default NULL)
```

Usage:

Table 11–19 DBMS_AQADM.UNSCHEDULE_PROPAGATION

Parameter	Description
src_queue_name (IN VARCHAR2)	Specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the administrative user.
destination (IN VARCHAR2)	Specifies the destination dblink. Messages in the source queue for recipients at this destination will be propagated. If it is NULL, the destination is the local database and messages will be propagated to other queues in the local database. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.

DBMS_AQADM.VERIFY_QUEUE_TYPES

Purpose:

Verify that the source and destination queues have identical types. The result of the verification is stored in `sys.aq$_message_types tables`, overwriting all previous output of this command.

Syntax:

```
DBMS_AQADM.SCHEDULE_PROPAGATION(
    src_queue_name     IN    VARCHAR2,
    dest_queue_name    IN    VARCHAR2,
    destination        IN    VARCHAR2 default NULL
    rc                 OUT   BINARY_INTEGER)
```

Usage:

Table 11–20 DBMS_AQADM.SCHEDULE_PROPAGATION

Parameter	Description
`src_queue_name` `(IN VARCHAR2)`	Specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the user.
`dest_queue_name` `(IN VARCHAR2)`	Specifies the name of the destination queue where messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the user.
`destination` `(IN VARCHAR2)`	Specifies the destination dblink. the destination queue name is in the database that is specified by the dblink. If the destination is `NULL`, the destination queue is the same database as the source queue. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.
`rc` `(OUT BINARY_INTEGER)`	Return code for the result of the procedure. If there is no error and if the source and destination queue types match the result is 1, if they do not match the result is 0. If an Oracle error is encountered it is returned in rc.

Enumerated Constants in the Administrative Interface

When using enumerated constants such as BROWSE, LOCKED, REMOVE, the symbol needs to be specified with the scope of the packages defining it. All types associated with the administrative interfaces have to be prepended with *dbms_aqadm*. For example:

```
dbms_aqadm.NORMAL_QUEUE
```

Table 11–21 Enumerated types in the administrative interface

Parameter	Options
retention	INFINITE
message_grouping	TRANSACTIONAL, NONE
queue_type	NORMAL_QUEUE, EXCEPTION_QUEUE

Database Objects

Queue table view

This is a view of the queue table in which message data is stored. This view is automatically created with each queue table and is called aq$<queue_table_name>. This view should be used for querying the queue data. The dequeue history data (time, user identification and transaction identification) is only valid for single consumer queues. For dequeue history of messages in a multiple consumer queue please refer to a following section.

The administrator can use any SQL statement or SQL tool to analyze and review the content of a queue or queue table. SQL provides full access to the message metadata and/or payload. Use ENQ_TXN_ID and DEQ_TXN_ID to correlate transactions. If the ENQ_TXN_ID of message m2 is the same as the DEQ_TXN_ID of m1, m2 is created in the transaction that consumed m1. (You may use CONNECT BY in your SQL statements to identify related messages). Remove retained messages that are not automatically removed by AQ. Do not update or modify messages since this may destroy the consistency of the queue metadata. Before you use SQL to correct any error in AQ, please contact the Oracle service representative.

Table 11–22 Queue Table View

Column Name & Description	Null?	Type
QUEUE — queue name		VARCHAR2(30)
MSG_ID — unique identifier of the message		RAW(16)
CORR_ID — user-provided correlation identifier		VARCHAR2(128)
MSG_PRIORITY — message priority		NUMBER
MSG_STATE — state of this message		VARCHAR2(9)
DELAY — number of seconds the message is delayed		DATE
EXPIRATION — number of seconds in which the message will expire after being READY		NUMBER
ENQ_TIME — enqueue time		DATE
ENQ_USER_ID — enqueue user id		NUMBER
ENQ_TXN_ID — enqueue transaction id	NOT NULL	VARCHAR2(30)
DEQ_TIME — dequeue time		DATE
DEQ_USER_ID — dequeue user id		NUMBER
DEQ_TXN_ID — dequeue transaction id		VARCHAR2(30)
RETRY_COUNT — number of retries		NUMBER
EXCEPTION_QUEUE_OWNER — exception queue schema		VARCHAR2(30)
EXCEPTION_QUEUE — exception queue name		VARCHAR2(30)
USER_DATA — user data		BLOB

DBA_QUEUE_TABLES

This view describes the names and types of all queue tables created in the database.

Table 11–23 DBA_QUEUE_TABLES

Column Name & Description	Null?	Type
OWNER — queue table schema		VARCHAR2(30)
QUEUE_TABLE - queue table name		VARCHAR2(30)
TYPE — payload type		VARCHAR2(7)
OBJECT_TYPE — name of object type, if any		VARCHAR2(61)
SORT_ORDER — user specified sort order		VARCHAR2(22)
RECIPIENTS — SINGLE or MULTIPLE		VARCHAR2(8)
MESSAGE_GROUPING — NONE or TRANSAC-TIONAL		VARCHAR2(13)
USER_COMMENT — user comment for the queue table		VARCHAR2(50)

USER_QUEUE_TABLES

This view is the same as DBA_QUEUES_TABLES with the exception that it only shows queue tables in the user's schema. It does not contain a column for OWNER.

DBA_QUEUES

Users can specify operational characteristics for individual queues. DBA_QUEUES contains the view which contains relevant information for every queue in a database.

Table 11–24 DBA_QUEUES

Column Name & Description	Null?	Type
OWNER — queue schema name	NOT NULL	VARCHAR2(30)
NAME — queue name	NOT NULL	VARCHAR2(30)
QUEUE_TABLE — queue table where this queue resides	NOT NULL	VARCHAR2(30)
QID — unique queue identifier	NOT NULL	NUMBER
QUEUE_TYPE — queue type		VARCHAR2(15)
MAX_RETRIES — number of dequeue attempts allowed		NUMBER
RETRY_DELAY — number of seconds before retry can be attempted		NUMBER
ENQUEUE_ENABLED — YES/NO		VARCHAR2(7)
DEQUEUE_ENABLED — YES/NO		VARCHAR2(7)
RETENTION — number of seconds message is retained after dequeue		VARCHAR2(40)
USER_COMMENT — user comment for the queue		VARCHAR2(50)

USER_QUEUES

This view is the same as DBA_QUEUES with the exception that it only shows queues in the user's schema. It does not contain a column for OWNER.

DBMS_AQADM.QUEUE_SUBSCRIBERS

Purpose: To get a list of subscribers for a queue.

Syntax:

```
DBMS_AQADM.QUEUE_SUBSCRIBERS(
    queue_name     IN     VARCHAR2)
RETURN aq$_subscriber_list_t
```

Usage: The function returns a PL/SQL table of aq$_agent. This can be used to get the list of all subscribers for a queue.

Example:

```
DECLARE
    subs            dbms_aqadm.aq$_subscriber_list_t;
    nsubs           BINARY_INTEGER;
    i               BINARY_INTEGER;
BEGIN
    subs  := dbms_aqadm.queue_subscribers('Q1DEF');
    nsubs := subs.COUNT;
    FOR i IN 0..nsubs-1 LOOP
        dbms_output.put_line(subs(i).name);
    END LOOP;
END;
/
```

DBA_QUEUE_SCHEDULES

Purpose: This view describes the current schedules for propagating messages.

Table 11–25 DBA_QUEUE_SCHEDULES

Column Name & Description	Null?	Type
SCHEMA	NOT NULL	VARCHAR2(30)
schema name for the source queue		
QNAME	NOT NULL	VARCHAR2(30)
source queue name		
DESTINATION	NOT NULL	VARCHAR2(128)
destination name, currently limited to be a DBLINK name		
START_DATE		DATE
date to start propagation in the default date format		
START_TIME		VARCHAR2(8)
time of day at which to start propagation in HH:MI:SS format		
PROPAGATION_WINDOW		NUMBER
duration in seconds for the propagation window		
NEXT_TIME		VARCHAR2(128)
function to compute the start of the next propagation window		
LATENCY		NUMBER
maximum wait time to propagate a message during the propagation window.		

Recipients and dequeue history of multiple consumer messages

The queue table view provides the dequeue history for single consumer queue messages. To query the list of recipients or the dequeue history of a message in a multiple-consumer queue you need to execute a SQL query on the queue table for the message of interest.

For example, to view the dequeue history of the message with msgid

'105E7A2EBFF11348E03400400B40F149' in queue table *sys.queue_tab* the following query must be executed. The query will return one row per consumer of the message.

```
SELECT consumer, transaction_id, deq_time, deq_user
FROM THE(select cast(history as sys.aq$_dequeue_history_t)
    FROM sys.queue_tab
    WHERE msgid='105E7A2EBFF11348E03400400B40F149');
```

Error Messages

The error messages for AQ are reported in two ranges:

24000 — 24099

25200 — 25299

Administration Topics

Performance

Queues are stored in database tables. The performance characteristics of queue operations are very similar to the underlying database operations.

Table and index structures

To understand the performance characteristics of queues it is important to understand the tables and index layout for AQ objects.

Creating a queue table creates a database table with approximately 25 columns. These columns store the AQ meta data and the user defined payload. The payload can be of an object type or RAW. The AQ meta data contains object types and scaler types. A view and two indexes are created on the queue table. The view allows users to query the message data. The indexes are used to accelerate access to mes-

sage data. Please refer to the create queue table command for a detailed description of the objects created.

Throughput

The code path of an enqueue operation is comparable to an insert into a multi-column table with two indexes. The code path of a dequeue operation is comparable to a select and delete operation on a similar table. These operations are performed using PL/SQL functions.

Availability

Oracle Parallel Server (OPS) can be used to ensure highly available access to queue data. Queues are implemented using database tables. The *tail* and the *head* of a queue can be extreme hot spots. Since OPS does not scale well in the presence of hot spots it is recommended to limit normal access to a queue from one instance only. In case of an instance failure messages managed by the failed instance can be processed immediately by one of the surviving instances.

Scalability

Queue operation scalability is similar to the underlying database operation scalability. If a dequeue operation with wait option is issued in a Multi-Threaded Server (MTS) environment the shared server process will be dedicated to the dequeue operation for the duration of the call including the wait time. The presence of many such processes could cause severe performance and availability problems and could result in deadlocking the shared server processes. For this reason it is recommended that dequeue requests with wait option be only issued via dedicated server processes. This restriction is not enforced.

Optimizing Propagation

In setting the number of `JOB_QUEUE_PROCESSES`, the DBA should aware that this need is determined by the number of queues *from* which the messages have to be propagated and the number of *destinations* (rather than queues) to which messages have to be propagated.

Reliability and Recoverability

The standard database reliability and recoverability characteristics apply to queue data.

Enterprise Manager Support

Enterprise manager supports GUIs for some of the administrative functions listed in the administrative interfaces section.

These include:

1. Queues as part of schema manager to view properties.
2. Start and stop queue.
3. Schedule and unschedule propagation.
4. Add and remove subscriber.
5. View the current propagation schedule.

Importing and Exporting Queue Data

Queues are implemented on tables. The import/export of queues constitutes the import/export of the underlying queue tables and related dictionary tables. Import and export of queues can only be done at queue table granularity.

When a queue table is exported, both the table definition information and the queue data are exported. When a queue table is imported, export action procedures will maintain the queue dictionary. Because the queue table data is also exported, the user is responsible for maintaining application-level data integrity when queue table data are being transported.

Importing queue data into a queue table with existing data is not recommended. During a table mode import, if the queue table already exists at the import site the old queue table definition, and the old queue definition will be dropped and recreated. Hence, queue table and queue definitions prior to the import will be lost.

Performing EXPORTS and IMPORTS of queue tables with multiple recipients

For every queue table that supports multiple recipients, there is a index-organized table (IOT) that contains important queue metadata. This metadata is essential to the operations of the queue, so the user must export and import this IOT as well as the queue table for the queues in this table to work after import. When the schema containing the queue table is exported, the IOT is also automatically exported. The behavior is similar at import time. Because the metadata table contains rowids of some rows in the queue table, import will issue a note about the rowids being obsolete when importing the metadata table. This message can be ignored, as the queueing system will automatically correct the obsolete rowids as a part of the import

process. However, if another problem is encountered while doing the import (such as running out of rollback segment space), the problem should be corrected and the import should be rerun.

Troubleshooting

This section describes some troubleshooting tips to diagnose problems with message propagation.

Message history

AQ updates the message history when a message has been successfully propagated to a destination. The message history is stored as a collection in the queue table. An administrator can execute a SQL query to determine if a message has been propagated. For example, to check if a message with msgid

105E7A2EBFF11348E03400400B40F149'

in queue table *aqadmn.queue_tab* has been propagated to destination '*boston*', the following query can be executed:

```
SELECT consumer, transaction_id, deq_time, deq_user, propagated_msgid
   FROM THE(select cast(history as sys.aq$_dequeue_history_t)
   FROM adadmn.queue_tab
      WHERE msgid='105E7A2EBFF11348E03400400B40F149')
      WHERE consumer LIKE '%BOSTON%';
```

A non-NULL `transaction_id` indicates that the message was successfully propagated. Further, the `deq_time` indicates the time of propagation, the `deq_user` indicates the userid used for propagation, and the `propagated_msgid` indicates the msgid of the message that was enqueued at the destination. If the message with the msgid cannot be found in the queue table, an administrator can check the exception queue (if the exception queue is in a different queue table) for the message history.

Propagation Schedules

The administrator can check the DBA_QUEUE_SCHEDULES view to check if propagation has been scheduled for a particular combination of source queue and destination. If propagation has been scheduled, the *jobno* of the job used to propagate messages can be determined from the `sys.aq$_schedules` table. The *jobno* can then be used to query the DBA_JOBS view to determine the last time that the propagation was scheduled for the combination of source queue and destination. The DBA_JOBS view also indicates the next time the propagation will be scheduled,

and if the job has been marked as *broken*. If the job has been marked as *broken*, check for errors in trace file(s) generated by the job_queue processes in the $ORACLE_HOME/log directory.

Database link

There are a number of points at which the propagation may break down:

- You may want to determine if the destination is reachable with regard to whether the network connection to the destination is available. You do this by executing a simple distributed query, or by creating a connection descriptor that has the same connect string, and then by trying to connect to the remote database.
- You need to ensure that the userid that scheduled the propagation (using dbms_aqadm.schedule_propagation) has access to the database link for the destination.
- Verify that the userid used to login to the destination through the database link has been granted privileges to use the AQ.
- Check if the queue name specified in the address attribute of the aq$_agent type (in the subscriber list for the source queue or in the recipient list of the enqueuer) both (a) exists at the specified destination, and (b) has been enabled for enqueuing. All these and other errors that the propagator encounters are logged into trace file(s) generated by the job_queue processes in $ORACLE_HOME/log directory.

Type checking

AQ will not propagate messages from one queue to another if the payload-types of the two queues are not equivalent. An administrator can verify if the source and destination's payload types match by executing the DBMS_AQADM.VERIFY_QUEUE_TYPES procedure. The results of the type checking will be stored in the sys.aq$_message_types table. This table can be accessed using the OID of the source queue and the address of the destination queue (i.e. [schema.]queue_name[@destination]).

Dynamic Statistics Views

As you can see, the GV$ view and V$ view are exactly the same:

Table 11–26 GV$AQ

Column Name	Type
QID	NUMBER
WAITING	NUMBER
READY	NUMBER
EXPIRED	NUMBER
TOTAL_WAIT	NUMBER
AVERAGE_WAIT	NUMBER

Table 11–27 V$AQ

Column Name	Type
QID	NUMBER
WAITING	NUMBER
READY	NUMBER
EXPIRED	NUMBER
TOTAL_WAIT	NUMBER
AVERAGE_WAIT	NUMBER

Column Name	Explanation
QID	the identity of the queue. This is the same as the qid in user_queues and dba_queues.
WAITING	the number of messages in the state 'WAITING'.
READY	the number of messages in state 'READY'.
EXPIRED	the number of messages in state 'EXPIRED'.
TOTAL_WAIT	the number of seconds for which messages in the queue have been waiting in state 'READY'
AVERAGE_WAIT	the average number of seconds a message in state 'READY' has been waiting to be dequeued.

The difference between these two views is that the GV$ view gives information about the number of messages in different states for the whole database while the V$ view gives information regarding specific instances. The way this works is that each instance keeps its own AQ statistics information in its own SGA, and does not have knowledge of the statistics gathered by other instances. Then, when a GV$AQ view is queried by an instance, all other instances funnel their AQ statistics information to the instance issuing the query.

Note: If you need to associate a specific queue or queues with a specific instance, you will have to enforce this at the application level.

Reference to Demos

The following demos may be found in the related directories:

```
$ORACLE_HOME/demo/aqdemo00.sql   Main driver of demo
$ORACLE_HOME/demo/aqdemo01.sql   Create queue tables and queues using
                                 AQ administration interface
$ORACLE_HOME/demo/aqdemo02.sql   Load the demo package
$ORACLE_HOME/demo/aqdemo03.sql     Submit the event handler as a job to Job
                                 Queue
$ORACLE_HOME/demo/aqdemo04.sql   Enqueue messages
```

Compatibility & Upgrade

The operational interface in Orcacle AQ 8.0.4 is backward compatible with the 8.0.3 Oracle AQ interface.

New Fields Enabled for the AQ$_AGENT Data Type

In the latest release, the address field is now enabled for the aq$_agent datatype. Consequently, it is now possible for this field to be specified wherever an interface takes an *Agent* as an argument — such as in the recipient list of the message properties, and the DBMS_AQADM.ADD_SUBSCRIBER administrative interface.

The Extended Address Field

The address field in the aq$_agent datatype has been extended to 1024 bytes. To use the extended address field, you will have to complete the following steps:

1. Save the contents of the all existing queues using the Export Utility.
2. Run CATNOQUEUE.SQL to drop the existing dictionary and queue tables:

   ```
   SVRMGRL> @CATNOQUEUE.SQL
   ```

3. Run CATQUEUE.SQL to redefine the new types and dictionary tables:

   ```
   SVRMGRL> @CATQUEUE.SQL
   ```

4. Import the queues you exported using the Import Utility.

Note: If your application does not require you to extend the address field, you need not complete these steps. In that case there will be no need to run the scripts and to perform export and import operations.

New Dictionary Tables

- The upgrade script for 8.0.4 (CAT8004.SQL) creates the additional dictionary tables: SYS.AQ$_MESSAGE_TYPES
- SYSTEM.AQ$_SCHEDULES
- SYS.AQ$_QUEUE_STATISTICS

12

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*.

WARNING: Pipes are independent of transactions. Be careful using pipes when transaction control can be affected.

Summary

Table 12–1 summarizes the procedures you can call in the DBMS_PIPE package.

Table 12–1 DBMS_PIPE Package Functions and Procedures

Function/Procedure	Description	Refer to
CREATE_PIPE	Explicitly create a pipe (necessary for private pipes).	page 12-4
PACK_MESSAGE	Build message in local buffer.	page 12-6
SEND_MESSAGE	Send message on named pipe. Implicitly create a public pipe if named pipe does not exist.	page 12-7
RECEIVE_MESSAGE	Copy message from named pipe into local buffer.	page 12-9
NEXT_ITEM_TYPE	Return datatype of next item in buffer.	page 12-11
UNPACK_MESSAGE	Access next item in buffer.	page 12-11
REMOVE_PIPE	Remove the named pipe.	page 12-12
PURGE	Purge contents of named pipe.	page 12-12
RESET_BUFFER	Purge contents of local buffer.	page 12-13
UNIQUE_ SESSION_NAME	Return unique session name.	page 12-13

Creating the DBMS_PIPE Package

To create the DBMS_PIPE package, submit the DBMSPIPE.SQL and PRVTPIPE.PLB scripts when connected as the user SYS. These scripts are run automatically by the CATPROC.SQL script. See "Privileges Required to Execute a Procedure" on page 10-38 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.

Syntax

The parameters for the CREATE_PIPE function are shown in Table 12–2 and the possible return values and their meanings are described in Table 12–3. 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;
```

Table 12–2 DBMS_PIPE.CREATE_PIPE Function Parameters

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 MAXPIPESIZE is 8192 bytes. The MAXPIPESIZE 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 MAXPIPESIZE 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_MESSAGE.

Table 12–3 DBMS_PIPE.CREATE_PIPE Function Return Values

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.

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 PACK_MESSAGE procedure itself is overloaded to accept items of type VARCHAR2,

NCHAR, 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  NCHAR);
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 12–4 and the possible return values and their meanings are described in Table 12–5. 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;
```

Table 12–4 DBMS_PIPE.SEND_MESSAGE Function Parameters

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	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 MAXPIPESIZE is 8192 bytes. The MAXPIPESIZE 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 MAXPIPESIZE to be increased. Callers with a smaller value simply use the existing, larger value. Specifying MAXPIPESIZE 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 MAXPIPESIZE parameter to override the creation pipe size specification.

Table 12–5 DBMS_PIPE.SEND_MESSAGE Function Return Values

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.
1	Indicates the pipe has timed out. This procedure can timeout 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.
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.

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 12–6 and the possible return values and their meanings are described in Table 12–7. The syntax for this function is shown below.

```
DBMS_PIPE.RECEIVE_MESSAGE(pipename      IN VARCHAR2,
                          timeout       IN INTEGER
                                        DEFAULT maxwait)
RETURN INTEGER;
```

Table 12–6 DBMS_PIPE.RECEIVE_MESSAGE Function Parameters

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 12–7 DBMS_PIPE.RECEIVE_MESSAGE Function Return Values

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.

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 12–8. The syntax for this function is shown below.

```
DBMS_PIPE.NEXT_ITEM_TYPE RETURN INTEGER;
```

Table 12–8 DBMS_PIPE.NEXT_ITEM_TYPE Function Return Values

Return Value	Description
0	no more items
6	NUMBER
9	VARCHAR2
12	DATE

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, NCHAR, 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 NCHAR);
DBMS_PIPE.UNPACK_MESSAGE       (item  OUT NUMBER);
DBMS_PIPE.UNPACK_MESSAGE       (item  OUT DATE);
DBMS_PIPE.UNPACK_MESSAGE_RAW   (item  OUT RAW);
DBMS_PIPE.UNPACK_MESSAGE_ROWID (item  OUT ROWID);
```

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 12–9. The syntax for this function is

```
DBMS_PIPE.REMOVE_PIPE(pipename  IN  VARCHAR2)
RETURN INTEGER;
```

Table 12–9 DBMS_PIPE.REMOVE_PIPE Function Return Values

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.

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 Enterprise 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 Enterprise Manager session from which you call it.

The DAEMON.SQL also uses the DBMS_OUTPUT package to display the results. For this example to work, you must have execute privileges on this package.

> **See Also:** "Output from Stored Procedures and Triggers" on page 12-22.

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.

Enterprise 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 Enterprise 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 Enterprise 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 12–10 shows the procedures that are callable from the DBMS_OUTPUT package:

Table 12–10 DBMS_OUTPUT Package Functions and Procedures

Function/Procedure	Description	Refer to
ENABLE	enable message output	page 12-23
DISABLE	disable message output	page 12-24
PUT_LINE	place a line in the buffer	page 12-24
PUT	place partial line in buffer	page 12-24
NEW_LINE	terminate a line created with PUT	page 12-38
GET_LINE	retrieve one line of information from buffer	page 12-25
GET_LINES	retrieve array of lines from buffer	page 12-25

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 Also:** "Privileges Required to Execute a Procedure" on page 10-38 for information on the necessary privileges for 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 Enterprise 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 SERVEROUT-PUT option of Enterprise 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.

Note: 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, Enterprise 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 Enterprise 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 12–11. The syntax for this procedure is shown below.

```
DBMS_OUTPUT.GET_LINE(line   OUT VARCHAR2,
                     status OUT INTEGER);
```

Table 12–11 DBMS_OUTPUT.GET_LINE Procedure Parameters

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.

The parameters for the GET_LINES procedure are described in Table 12–12. 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.

Table 12–12 DBMS_OUTPUT.GET_LINE Procedure Parameters

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.

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 Enterprise 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 indention 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 12–13 summarizes the procedures you can call in the UTL_FILE package.

Table 12–13 UTL_FILE Procedures

Function/Procedure	Description	Refer to
FOPEN	Open a file for input or output. Create an output file if it does not exist.	page 12-33
IS_OPEN	Determine if a file handle refers to an open file.	page 12-34
FCLOSE	Close a file.	page 12-35
FCLOSE_ALL	Close all open file handles.	page 12-36
GET_LINE	Read a line of text from an open file.	page 12-36
PUT	Write a line to a file. Do not append a line terminator.	page 12-37
PUT_LINE	Write a line to a file. Append an OS-specific line terminator.	page 12-39
PUTF	A PUT procedure with formatting.	page 12-39
NEW_LINE	Write one or more OS-specific line terminators to a file.	page 12-38
FFLUSH	Physically write all pending output to a file.	page 12-41

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.**
- **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 writable 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
```

WARNING: '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 12–14.

Table 12–14 UTL_FILE Package Exceptions

Exception Name	Description
INVALID_PATH	File location or filename was invalid.
INVALID_MODE	The open_mode 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.

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 12–15, and the syntax is shown below.

Syntax

```
FUNCTION FOPEN(location  IN VARCHAR2,
               filename  IN VARCHAR2,
               open_mode IN VARCHAR2)
    RETURN UTL_FILE.FILE_TYPE;
```

Table 12–15 FOPEN Function Parameters

Parameters	Description
location	The operating system-specific string that specifies the directory or area in which to open the file.
filename	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 '/'.)
open_mode	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: 'r' read text (GET_LINE) 'w' write text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH) 'a' append text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)

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.

Note:

- The *file location* and *file nam*e 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.
- 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 12–16, and the syntax is shown below.

Syntax

```
FUNCTION IS_OPEN(file_handle  IN FILE_TYPE)
    RETURN BOOLEAN;
```

Table 12–16 IS_OPEN Function Parameters

Parameter	Description
file_handle	An active file handle returned by an FOPEN call.

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 12–17, and the syntax is shown below.

Syntax

```
PROCEDURE FCLOSE (file_handle IN OUT FILE_TYPE);
```

Table 12–17 FCLOSE Procedure Parameters

Parameter	Description
file_handle	An active file handle returned by an FOPEN call.

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.

> **Note:** 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 12–18, and the syntax is shown below.

Syntax

```
PROCEDURE GET_LINE(file_handle     IN  FILE_TYPE,
                   buffer          OUT VARCHAR2);
```

Table 12–18 GET_LINE Procedure Parameters

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.

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 12–19, and the syntax is shown below.

Syntax

```
PROCEDURE PUT(file_handle  IN FILE_TYPE,
              buffer       IN VARCHAR2);
```

Table 12–19 PUT Procedure Parameters

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.

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 12–20, and the syntax is shown below.

Syntax

```
PROCEDURE NEW_LINE (file_handle  IN FILE_TYPE,
                    lines        IN NATURAL := 1);
```

Table 12–20 NEW_LINE Procedure Parameters

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.

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 12–21, and the syntax is shown below.

Syntax

```
PROCEDURE PUT_LINE(file_handle  IN FILE_TYPE,
                   buffer       IN VARCHAR2);
```

Table 12–21 PUT_LINE Procedure Parameters

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.

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 12–22, and the syntax is shown below.

Syntax

```
PROCEDURE PUTF(file_handle  IN FILE_TYPE,
               format       IN VARCHAR2,
               [arg1        IN VARCHAR2,
                . . .
                arg5        IN VARCHAR2]);
```

Table 12–22 PUTF Procedure Parameters

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 formatting 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.

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 12–23, and the syntax is shown below.

Syntax

```
PROCEDURE FFLUSH (file_handle  IN FILE_TYPE);
```

Table 12–23 FFLUSH Procedure Parameters

Parameters	Description
file_handle	An active file handle returned by an FOPEN call.

Exceptions

FFLUSH can raise any of the following exceptions:

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

13

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:

- Designing Triggers
- Creating Triggers
- When Triggers Are Compiled
- Debugging a Trigger
- Modifying a Trigger
- Enabling and Disabling Triggers
- Listing Information About Triggers
- Examples of Trigger Applications

Note: If you are using Trusted Oracle, see the *Trusted Oracle* documentation 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 Enterprise 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 13-22.

Prerequisites for Creating Triggers

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.

> **Note:** 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 and AFTER Options

The BEFORE or AFTER option in the CREATE TRIGGER statement specifies exactly when to file the trigger body in relation to the triggering statement that is 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.

The INSTEAD OF Option

The INSTEAD OF option in the CREATE TRIGGER statement is an alternative to the BEFORE and AFTER options. INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through UPDATE, INSERT, and DELETE statements. These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement. The trigger performs update, insert, or delete operations directly on the underlying tables.

Users write normal UPDATE, INSERT, and DELETE statements against the view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place.

By default, INSTEAD OF triggers are activated for each row (see “FOR EACH ROW Option” on page 13-7).

Views That Are Not Modifiable

A view cannot be modified by UPDATE, INSERT, or DELETE statements if the view query contains any of the following constructs:

- set operators
- group functions
- GROUP BY, CONNECT BY, or START WITH clauses
- the DISTINCT operator
- joins (a subset of join views are updatable)

If a view contains pseudocolumns or expressions, you can only update the view with an UPDATE statement that does not refer to any of the pseudocolumns or expressions.

Example of an INSTEAD OF Trigger

The following example shows an INSTEAD OF trigger for inserting rows into the MANAGER_INFO view.

```
CREATE VIEW manager_info AS
    SELECT e.name, e.empno, d.dept_type, d.deptno, p.level,
           p.projno
        FROM   emp e, dept d, project p
        WHERE  e.empno =  d.mgr_no
        AND    d.deptno = p.resp_dept;

CREATE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info
REFERENCING NEW AS n                 -- new manager information

FOR EACH ROW
BEGIN
  IF NOT EXISTS SELECT * FROM emp
        WHERE emp.empno = :n.empno
  THEN
        INSERT INTO emp
            VALUES(:n.empno, :n.name);
  ELSE
        UPDATE emp SET emp.name = :n.name
        WHERE emp.empno = :n.empno;
 END IF;

 IF NOT EXISTS SELECT * FROM dept
     WHERE dept.deptno = :n.deptno
 THEN
     INSERT INTO dept
        VALUES(:n.deptno, :n.dept_type);
 ELSE
     UPDATE dept SET dept.dept_type = :n.dept_type
        WHERE dept.deptno = :n.deptno;
 END IF;

 IF NOT EXISTS SELECT * FROM project
     WHERE project.projno = :n.projno
 THEN
     INSERT INTO project
        VALUES(:n.projno, :n.project_level);
 ELSE
        UPDATE project SET project.level = :n.level
            WHERE project.projno = :n.projno;
```

```
  END IF;
END;
```

The actions shown for rows being inserted into the MANAGER_INFO view first test to see if appropriate rows already exist in the base tables from which MANAGER_INFO is derived. The actions then insert new rows or update existing rows, as appropriate. Similar triggers can specify appropriate actions for UPDATE and DELETE.

Object Views and INSTEAD OF Triggers

INSTEAD OF triggers provide the means to modify object view instances on the client-side through OCI calls. (See *Oracle Call Interface Programmer's Guide* for more information.) To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, the user must specify INSTEAD OF triggers, unless the object view is modifiable. If the object is read only, however, it is not necessary to define triggers to pin it.

> **Note Also:** For more examples of INSTEAD OF triggers, see "Updating the Object Views" on page 8-6.

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:** Exactly one table (but not a view) can be specified in the triggering statement.

For example, the PRINT_SALARY_CHANGES trigger on page 13-2 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;
```

See Also: For the order of trigger firing, see *Oracle8 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 REFERENCEING 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.

> **See Also:** For more information about error processing in PL/SQL program units, see "Error Handling" on page 10-29 and "Declaring Exceptions and Exception Handling Routines" on page 10-30.

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 schema 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 Also:** *Oracle8 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.

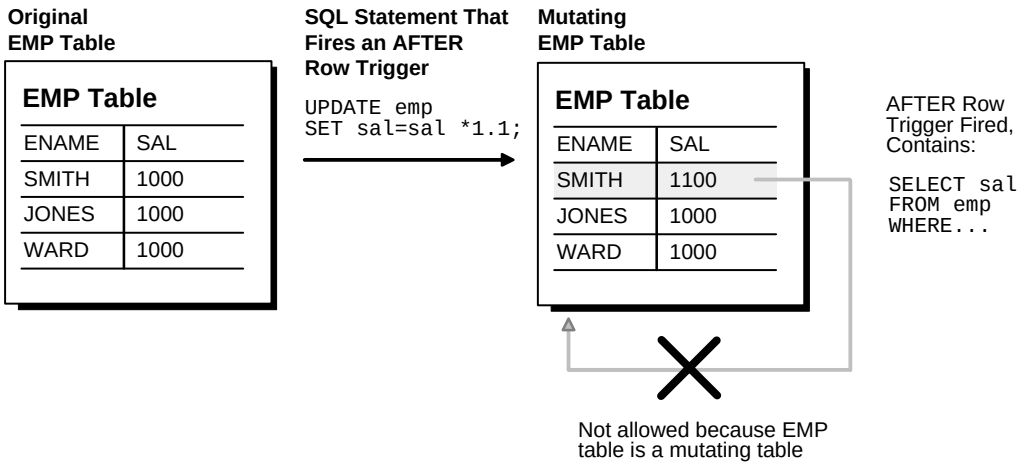
- The SQL statements of a trigger cannot read from (query) or modify a mutating table of the triggering statement.
- The statements of a trigger cannot change the PRIMARY, FOREIGN, or UNIQUE KEY columns of a constraining table of the triggering statement.

  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 13–1 illustrates the restriction placed on mutating tables.

Figure 13–1 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 Also:** *Oracle8 Concepts* 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 Net8 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.

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 Also:** "Privileges Required to Execute a Procedure" on page 10-38.

When Triggers Are Compiled

Triggers are similar to PL/SQL anonymous blocks with the addition of the :NEW and :OLD capabilities, but their compilation is different. 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

Triggers, in contrast, 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. (Runtime trigger errors always cause the DML statement to fail.) You can use the SHOW ERRORS command in SQL*Plus or Enterprise 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.

To recompile a trigger, you must own the trigger or have the ALTER ANY TRIGGER system privilege.

Migration Issues

Non-compiled triggers cannot be fired under compiled trigger releases (such as Oracle 7.3 and Oracle8). If upgrading from a non-compiled trigger release to a compiled trigger release, all existing triggers must be compiled. The upgrade script *cat73xx.sq*l invalidates all triggers so that they are automatically recompiled when first executed. (The *xx* stands for a variable minor release number.)

Downgrading from Oracle 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.

Note Also: "Debugging" on page 10-35

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 that 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 schema 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 *Oracle8 Reference* 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 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.

See Also: Chapter 9, "Maintaining Data Integrity"

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 site 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;
```

WARNING: 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_cascade1 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 13-21 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 dependent 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;
```

14

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 of Dynamic SQL

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 Also:** "Privileges Required to Execute a Procedure" on page 10-38 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 *Oracle Call Interface Programmer's Guide* 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 CANCEL cursor procedures.
- 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 14-30. 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);
```

```
    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 14–1. A general explanation of these procedures follows.

Each of these procedures is described in greater detail starting on page 14-8.

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

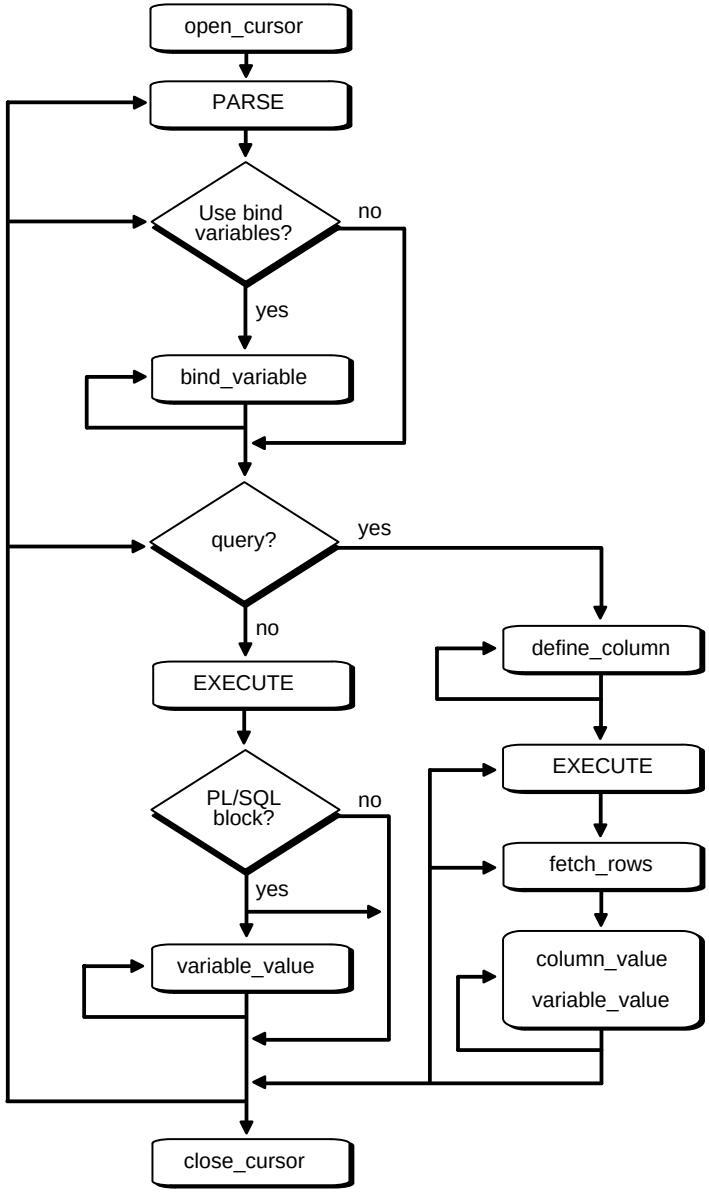
Every SQL statement must be parsed by calling the PARSE procedure. Parsing the statement checks the statement's syntax and associates it with the cursor in your program.

> **See Also:** *Oracle8 Concepts* for an explanation of how SQL statements are parsed.

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.

> **Note:** 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.

Figure 14–1 DBMS_SQL Execution Flow



BIND_VARIABLE or BIND_ARRAY

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 one of the bind procedures, BIND_VARIABLE or BIND_ARRAY, to supply the value of a variable in your program (or the values of an array) 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.

DBMS_SQL can execute a DML statement multiple times — each time with a different bind variable. The BIND_ARRAY procedure allows you to bind an array of scalars, each value of which will be used as an input variable once per EXECUTE. This is similar to the array interface supported by the OCI.

DEFINE_COLUMN, DEFINE_COLUMN_LONG, or DEFINE_ARRAY

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 one of the define procedures (DEFINE_COLUMN, DEFINE_COLUMN_LONG, or DEFINE_ARRAY) to specify the variables that are to receive the SELECT values, much the way an INTO clause does for a static query.

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 procedure to fetch from the LONG column.

You use the DEFINE_ARRAY procedure to define a PL/SQL array into which you want to fetch rows in a single SELECT statement. You must call DEFINE_ARRAY before using the COLUMN_VALUE procedure to fetch the rows.

EXECUTE

Call the EXECUTE function to execute your SQL statement.

FETCH_ROWS or EXECUTE_AND_FETCH

Call the FETCH_ROWS function 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, COLUMN_VALUE, or COLUMN_VALUE_LONG

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.

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 for Dynamic SQL

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.

Therefore, if a user creates a procedure and grants EXECUTE privilege on it to a second user, the second user must also be granted privileges *explicitly* for any DML operations performed in the body of the stored procedure. (These privileges cannot be granted through a role, because roles are disabled inside stored procedures.) Not granting privileges for the DML operations may result in the error message ORA-01031: "insufficient privileges" at runtime.

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 sys-

tem privileges of the owner of the stored procedure and the union of the mandatory access control (MAC) privileges granted to the stored procedure and the current user.

Procedures and Functions

Table 14–1 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.

> **See Also:** "Examples of Using DBMS_SQL" on page 14-30 for examples of how these functions can be used.

Table 14–1 DBMS_SQL Package Functions and Procedures

Function/Procedure	Description	Refer to
OPEN_CURSOR	Return cursor ID number of new cursor.	page 14-9
PARSE	Parse given statement.	page 14-10
BIND_VARIABLE	Bind a given value to a given variable.	page 14-11
BIND_ARRAY	Bind a given value to a given array.	page 14-11
DEFINE_COLUMN	Define a column to be selected from the given cursor, used only with SELECT statements.	page 14-16
DEFINE_ARRAY	Define an array to be selected from the given cursor, used only with SELECT statements.	page 14-17
DEFINE_COLUMN_LONG	Define a LONG column to be selected from the given cursor, used only with SELECT statements.	page 14-19
EXECUTE	Execute a given cursor.	page 14-20
EXECUTE_AND_FETCH	Execute a given cursor and fetch rows.	page 14-20
FETCH_ROWS	Fetch a row from a given cursor.	page 14-21

Table 14–1 (Cont.) DBMS_SQL Package Functions and Procedures

Function/Procedure	Description	Refer to
COLUMN_VALUE	Returns value of the cursor element for a given position in a cursor.	page 14-21
COLUMN_VALUE_LONG	Returns a selected part of a LONG column, that has been defined using DEFINE_COLUMN_LONG.	page 14-23
VARIABLE_VALUE	Returns value of named variable for given cursor.	page 14-24
IS_OPEN	Returns TRUE if given cursor is open.	page 14-26
DESCRIBE_COLUMNS	Describes the columns for a cursor opened and parsed through DBMS_SQL.	page 14-26
CLOSE_CURSOR	Closes given cursor and frees memory.	page 14-28
LAST_ERROR_POSITION	Returns byte offset in the SQL statement text where the error occurred.	page 14-29
LAST_ROW_COUNT	Returns cumulative count of the number of rows fetched.	page 14-29
LAST_ROW_ID	Returns ROWID of last row processed.	page 14-29
LAST_SQL_FUNCTION_CODE	Returns SQL function code for statement.	page 14-29

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 of OPEN_CURSOR

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. All statements are parsed immediately. This may change in future versions; you should not rely on this behavior.

Syntax of PARSE

The parameters for the PARSE procedure are described in Table 14–2. The syntax for this procedure is:

```
DBMS_SQL.PARSE(
        c                IN INTEGER,
        statement        IN VARCHAR2,
        language_flag    IN INTEGER);
```

The size limit for parsing SQL statements with the syntax above is **size_t**, which is the largest possible contiguous allocation on the machine being used. The PARSE procedure also supports the following syntax for large SQL statements:

```
DBMS_SQL.PARSE(
        c                IN INTEGER,
        statement        IN VARCHAR2S,
        lb               IN INTEGER,
        ub               IN INTEGER,
        lfflg            IN BOOLEAN,
        language_flag    IN INTEGER);
```

The procedure concatenates elements of a PL/SQL table statement and parses the resulting string. You can use this procedure to parse a statement that is longer than the limit for a single VARCHAR2 variable by splitting up the statement.

Table 14–2 DBMS_SQL.PARSE Procedure Parameters

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.
lb	Provide the lower bound for elements in the statement.
ub	Provide the upper bound for elements in the statement.
lfflg	If TRUE, insert a linefeed after each element on concatenation.
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.

The VARCHAR2S Datatype for Parsing Large SQL Strings

To parse SQL statements larger than 32 KB, the DBMS_SQL package makes use of PL/SQL tables to pass a table of strings to the PARSE procedure. These strings are concatenated and then passed on to the Oracle Server.

You can declare a local variable as the VARCHAR2S table-item type, and then use the PARSE procedure to parse a large SQL statement as VARCHAR2S.

The definition of the VARCHAR2S datatype is:

```
type varchar2s is table of varchar2(256) index by binary_integer;
```

Call DBMS_SQL.PARSE to parse a large SQL statement in the given cursor. All statements are parsed immediately.

BIND_VARIABLE and BIND_ARRAY Procedures

Call BIND_VARIABLE or BIND_ARRAY to bind a given value or set of values 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 or an IN array, the given bind value must be valid for the variable or array type. Bind values for OUT variables are ignored.

The bind variables or arrays of a SQL statement are identified by their names. When binding a value to a bind variable or bind array, the string identifying it 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 of BIND_VARIABLE

The parameters for the BIND_VARIABLE procedures and functions are described in Table 14–3. The syntax for these procedures and functions is shown below. Notice that BIND_VARIABLE 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 CHARACTER SET ANY_CS
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

> **See Also:** Chapter 6, "Large Objects (LOBs)" describes the BLOB, CLOB, and BFILE datatypes.

The following syntax is also supported for BIND_VARIABLE. The square brackets [] indicate an optional parameter for the BIND_VARIABLE function.

```
DBMS_SQL.BIND_VARIABLE(
        c               IN INTEGER,
        name            IN VARCHAR2,
        value           IN VARCHAR2 CHARACTER SET ANY_CS
      [,out_value_size IN INTEGER]);
```

To bind CHAR, RAW, and ROWID data, you can use the following variations on the syntax:

```
DBMS_SQL.BIND_VARIABLE_CHAR(
```

```
        c              IN INTEGER,
        name           IN VARCHAR2,
        value          IN CHAR CHARACTER SET ANY_CS
      [,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);
```

Table 14–3 DBMS_SQL.BIND_VARIABLE Procedure Parameters

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.

Bulk Array Binds

Bulk selects, inserts, updates, and deletes can enhance the performance of applications by bundling many calls into one. The DBMS_SQL package allows you to work on arrays of data using the PL / SQL table type.

Table items are unbounded homogeneous collections. In persistent storage, they are like other relational tables and have no intrinsic ordering. But when a table item is brought into the workspace (either by querying or by navigational access of persistent data), or when it is created as the value of a PL / SQL variable or parameter, its elements are given subscripts that can be used with array-style syntax to get and set the values of elements.

The subscripts of these elements need not be dense, and can be any number including negative numbers. For example, a table item can contain elements at locations -10, 2, and 7 only.

When a table item is moved from transient workspace to persistent storage, the subscripts are not stored; the table item is unordered in persistent storage.

At bind time the table is copied out from the PL/SQL buffers into local DBMS_SQL buffers (the same as for all scalar types) and then the table is manipulated from the local DBMS_SQL buffers. Therefore, if you change the table after the bind call, that change will not affect the way the execute behaves.

Types for Scalar and LOB Arrays

You can declare a local variable as one of the following table-item types, which are defined as public types in DBMS_SQL.

```
type Number_Table is table of Number index by binary_integer;
type Varchar2_Table is table of varchar2(2000) index by binary_integer;
type Date_Table is table of Date index by binary_integer;
type Blob_Table is table of Blob index by binary_integer;
type Clob_Table is table of Clob index by binary_integer;
type Bfile_Table is table of Bfile index by binary_integer;
```

Syntax of BIND_ARRAY

The parameters for the BIND_ARRAY procedure to bind an entire array or a range of an array are described in Table 14–4. The syntax for this procedure is shown below. Notice that the BIND_ARRAY procedure is overloaded to accept different datatypes.

```
DBMS_SQL.BIND_ARRAY(
        c                  IN INTEGER,
        name               IN VARCHAR2,
        <table_variable>   IN <datatype>
      [,index1              IN INTEGER,
        index2              IN INTEGER)] );
```

where the <table_variable> and its corresponding <datatype> can be any one of the following matching pairs:

```
<num_tab>      Number_Table
<vchr2_tab>    Varchar2_Table
<date_tab>     Date_Table
<blob_tab>     Blob_Table
<clob_tab>     Clob_Table
```

<bfile_tab> Bfile_Table

Table 14–4 DBMS_SQL.BIND_ARRAY Procedure Parameters

Parameter	Description
c	Specify the ID number of the cursor to which you want to bind a value.
name	Provide the name of the array in the statement.
table_variable	Specify the local variable that has been declared as <datatype>.
index1	Provide the index for the table element that marks the lower bound of the range.
index2	Provide the index for the table element that marks the upper bound of the range.

For binding a range, the table must contain the elements that specify the range — tab(index1) and tab(index2) — but the range does not have to be dense. Index1 must be less than or equal to index2. All elements between tab(index1) and tab(index2) will be used in the bind.

If you do not specify indexes in the bind call and two different binds in a statement specify tables that contain a different number of elements, the number of elements actually used is the minimum number between all tables. This is also the case if you specify indexes — the minimum range is selected between the two indexes for all tables.

Not all bind variables in a query have to be array binds. Some can be regular binds and the same value will be used for each element of the arrays in expression evaluations (and so forth).

> **See Also:** "Examples 3, 4, and 5: Bulk DML" on page 14-32 for examples of how to bind arrays.

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, DEFINE_COLUMN_LONG, or DEFINE_ARRAY.

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 or COLUMN_VALUE_LONG 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 of DEFINE_COLUMN

The parameters for the DEFINE_COLUMN procedure are described in Table 14–5. The syntax for this procedure is shown below. Notice that DEFINE_COLUMN 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
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

> **See Also:** Chapter 6, “Large Objects (LOBs)” describes BLOB, CLOB, and BFILE datatypes.

The following syntax is also supported for the DEFINE_COLUMN procedure:

```
DBMS_SQL.DEFINE_COLUMN(
        c               IN INTEGER,
        position        IN INTEGER,
```

```
        column          IN VARCHAR2 CHARACTER SET ANY_CS,
        column_size     IN INTEGER);
```

To define columns with CHAR, RAW, and ROWID data, you can use the following variations on the procedure syntax:

```
DBMS_SQL.DEFINE_COLUMN_CHAR(
        c               IN INTEGER,
        position        IN INTEGER,
        column          IN CHAR CHARACTER SET ANY_CS,
        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);
```

Table 14–5 DBMS_SQL.DEFINE_COLUMN Procedure Parameters

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.

DEFINE_ARRAY Procedure

Call DEFINE_ARRAY to define the appropriate table variable into which you want to fetch rows (with a FETCH_ROWS call). This procedure allows you to do batch fetching of rows from a single SELECT statement. A single fetch call brings over a number of rows into the PL/SQL aggregate object.

When you fetch the rows, they are copied into DBMS_SQL buffers until you execute a COLUMN_VALUE call, at which time the rows are copied into the table that was passed as an argument to the COLUMN_VALUE call.

Scalar and LOB Types for Arrays

You can declare a local variable as one of the following table-item types, and then fetch any number of rows into it using DBMS_SQL. (These are the same types as you can specify for the BIND_ARRAY procedure.)

```
type Number_Table is table of Number index by binary_integer;
type Varchar2_Table is table of varchar2(2000) index by binary_integer;
type Date_Table is table of Date index by binary_integer;
type Blob_Table is table of Blob index by binary_integer;
type Clob_Table is table of Clob index by binary_integer;
type Bfile_Table is table of Bfile index by binary_integer;
```

Syntax of DEFINE_ARRAY

The parameters for the DEFINE_ARRAY procedure are described in Table 14–6. The syntax for this procedure is shown below. Notice that the DEFINE_ARRAY procedure is overloaded to accept different datatypes.

```
DBMS_SQL.DEFINE_ARRAY(
        c                IN INTEGER,
        position         IN INTEGER,
        <table_variable>  IN <datatype>,
        count            IN INTEGER,
        indx             IN INTEGER);
```

where the <table_variable> and its corresponding <datatype> can be any one of the following matching pairs:

```
<num_tab>      Number_Table
<vchr2_tab>    Varchar2_Table
<date_tab>     Date_Table
<blob_tab>     Blob_Table
<clob_tab>     Clob_Table
<bfile_tab>    Bfile_Table
```

This procedure defines an appropriate “table” for the column at position “position” for cursor “c”. The subsequent FETCH_ROWS call will fetch “count” rows. When the COLUMN_VALUE call is made, these rows will be placed in positions indx, indx+1, indx+2, and so on. While there are still rows coming, the user keeps issuing FETCH_ROWS/COLUMN_VALUE calls. The rows keep accumulating in the table specified as an argument in the COLUMN_VALUE call.

Table 14–6 DBMS_SQL.DEFINE_ARRAY Procedure Parameters

Parameter	Description
c	Specify the ID number of the cursor to which you want to bind an array.
name	Provide the name of the define variable in the statement.
table_variable	Specify the local variable that was declared as <datatype>.
position	The relative position of the column in the array being defined. The first column in a statement has position 1.
indx	Provide the starting position for where the rows should be placed.

The "count" has to be an integer greater than zero, otherwise an exception is raised. The "indx" can be positive, negative, or zero. A query on which a DEFINE_ARRAY call was issued cannot contain array binds.

> **See Also:** "Examples 6 and 7: Defining an Array" on page 14-34 for examples of how to define arrays.

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 of DEFINE_COLUMN_LONG

The parameters of DEFINE_COLUMN_LONG are described in Table 14–7. The syntax is:

```
DBMS_SQL.DEFINE_COLUMN_LONG(
        c               IN INTEGER,
        position        IN INTEGER);
```

Table 14–7 DBMS_SQL.DEFINE_COLUMN_LONG Procedure Parameters

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.

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 of EXECUTE

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 of EXECUTE_AND_FETCH

The EXECUTE_AND_FETCH function returns the number of rows actually fetched. The parameters for this procedure are described in Table 14–8, and the syntax is shown below.

```
DBMS_SQL.EXECUTE_AND_FETCH(
        c               IN INTEGER,
        exact           IN BOOLEAN DEFAULT FALSE)
RETURN INTEGER;
```

Table 14–8 DBMS_SQL.EXECUTE_AND_FETCH Function Parameters

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.

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 of FETCH_ROWS

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 of COLUMN_VALUE

The parameters for the COLUMN_VALUE procedure are described in Table 14–9. 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 CHARACTER SET ANY_CS
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

> **See Also:** Chapter 6, "Large Objects (LOBs)" describes the BLOB, CLOB, and BFILE datatypes.

The following syntax is also supported for the COLUMN_VALUE procedure:

```
DBMS_SQL.COLUMN_VALUE(
        c                 IN  INTEGER,
        position          IN  INTEGER,
        <table_variable>  IN  <datatype>);
```

where the <table_variable> and its corresponding <datatype> can be any one of these matching pairs:

```
<num_tab>      Number_Table
<vchr2_tab>    Varchar2_Table
<date_tab>     Date_Table
<blob_tab>     Blob_Table
<clob_tab>     Clob_Table
<bfile_tab>    Bfile_Table
```

For columns containing CHAR, RAW, and ROWID data, you can use the following variations on the syntax:

```
DBMS_SQL.COLUMN_VALUE_CHAR(
        c               IN  INTEGER,
        position        IN  INTEGER,
        value           OUT CHAR CHARACTER SET ANY_CS
      [,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]);
```

Table 14–9 DBMS_SQL.COLUMN_VALUE Procedure Parameters

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.
table_variable	IN	Specify the local variable that has been declared as <datatype>.
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.

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 of COLUMN_VALUE_LONG

The parameters of the COLUMN_VALUE_LONG procedure are described in Table 14–10. 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);
```

Table 14–10 DBMS_SQL.COLUMN_VALUE_LONG Procedure Parameters

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.

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 of VARIABLE_VALUE

The parameters for the VARIABLE_VALUE procedure are described in Table 14–11. The syntax for this procedure is:

```
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 CHARACTER SET ANY_CS
BLOB
CLOB CHARACTER SET ANY_CS
BFILE
```

For variables containing CHAR, RAW, and ROWID data, you can use the following variations on the syntax:

```
DBMS_SQL.VARIABLE_VALUE_CHAR(
        c               IN  INTEGER,
        name            IN  VARCHAR2,
        value           OUT CHAR CHARACTER SET ANY_CS);
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);
```

Table 14–11 DBMS_SQL.VARIABLE_VALUE Procedure Parameters

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.
position	IN	Specify the relative position of the column in the cursor. The first column in a statement has position 1.

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 14-20.
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 14-24.

IS_OPEN Function

The IS_OPEN function returns TRUE if the given cursor is currently open.

Syntax of IS_OPEN

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;
```

DESCRIBE_COLUMNS Procedure

Call this procedure to describe the columns for a cursor opened and parsed through DBMS_SQL.

The DESC_REC Type

The DBMS_SQL package declares the DESC_REC record type as follows:

```
type desc_rec is record (
        col_type            binary_integer := 0,
        col_max_len         binary_integer := 0,
        col_name            varchar2(32)   := '',
        col_name_len        binary_integer := 0,
        col_schema_name     varchar2(32)   := '',
        col_schema_name_len binary_integer := 0,
        col_precision       binary_integer := 0,
        col_scale           binary_integer := 0,
        col_charsetid       binary_integer := 0,
        col_charsetform     binary_integer := 0,
        col_null_ok         boolean        := TRUE);
```

The parameters of DESC_REC are described in Table 14–12.

Table 14–12 DESC_REC Type Parameters

Parameter	Description
col_type	The type of the column being described.
col_max_len	The maximum length of the column.
col_name	The name of the column.
col_name_len	The length of the column name.
col_schema_name	The name of the schema the column type was defined in (if an object type).
col_schema_name_len	The length of the schema.
col_precision	The column precision if a number.
col_scale	The column scale if a number.
col_charsetid	The column character set identifier.
col_charsetform	The column character set form.
col_null_ok	True if column can be null.

The DESC_TAB Type

The DESC_TAB type is a PL/SQL table of DESC_REC records:

```
type desc_tab is table of desc_rec index by binary_integer;
```

You can declare a local variable as the PL/SQL table type DESC_TAB, and then call the DESCRIBE_COLUMNS procedure to fill in the table with the description of each column. All columns are described; you cannot describe a single column.

Syntax of DESCRIBE_COLUMNS

The parameters of DESCRIBE_COLUMNS are described in Table 14–13. The syntax is:

```
DBMS_SQL.DESCRIBE_COLUMNS(
        c               IN  INTEGER,
        col_cnt         OUT INTEGER,
        desc_t          OUT DESC_TAB);
```

Table 14–13 DBMS_SQL.DESCRIBE_COLUMNS Procedure Parameters

Parameter	Mode	Description
c	IN	The ID number of the cursor for the columns being described.
col_cnt	OUT	
desc_t	OUT	

See Also: "Example 8: Describe Columns" on page 14-37 for an example of how to use DESCRIBE_COLUMNS.

CLOSE_CURSOR Procedure

Call CLOSE_CURSOR to close a given cursor.

Syntax of CLOSE_CURSOR

The parameter for the CLOSE_CURSOR procedure is described in Table 14–14. The syntax for this procedure is:

```
DBMS_SQL.CLOSE_CURSOR(
        c               IN OUT INTEGER);
```

Table 14–14 DBMS_SQL.CLOSE_CURSOR Procedure Parameters

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.

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 *Oracle Call Interface Programmer's Guide*.

```
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 of Using DBMS_SQL

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);
   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);
  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);

  -- 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, 'birthdate',
                             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;
```

Examples 3, 4, and 5: Bulk DML This series of examples shows how to use bulk array binds (table items) in the SQL DML statements DELETE, INSERT, and UPDATE.

In a DELETE statement, for example, you could bind in an array in the WHERE clause and have the statement be executed for each element in the array:

```
declare
  stmt varchar2(200);
  dept_no_array dbms_sql.Number_Table;
  c number;
  dummy number;
begin
  dept_no_array(1) := 10; dept_no_array(2) := 20;
  dept_no_array(3) := 30; dept_no_array(4) := 40;
```

```
  dept_no_array(5) := 30; dept_no_array(6) := 40;
  stmt := 'delete from emp where deptno = :dept_array';
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, stmt, dbms_sql.native);
  dbms_sql.bind_array(c, ':dept_array', dept_no_array, 1, 4);
  dummy := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);

  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

In the example above, only elements 1 through 4 will be used as specified by the bind_array call. Each element of the array will potentially delete a large number of employees from the database.

Here is an example of a bulk INSERT statement:

```
declare
  stmt varchar2(200);
  empno_array dbms_sql.Number_Table;
  empname_array dbms_sql.Varchar2_Table;
  c number;
  dummy number;
begin
  for i in 0..9 loop
    empno_array(i) := 1000 + i;
    empname_array(I) := get_name(i);
  end loop;
  stmt := 'insert into emp values(:num_array, :name_array)';
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, stmt, dbms_sql.native);
  dbms_sql.bind_array(c, ':num_array', empno_array);
  dbms_sql.bind_array(c, ':name_array', empname_array);
  dummy := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);

  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
```

```
    raise;
end;
/
```

When the execute takes place, all 10 of the employees are inserted into the table.

Finally, here is an example of an bulk UPDATE statement.

```
declare
  stmt varchar2(200);
  emp_no_array dbms_sql.Number_Table;
  emp_addr_array dbms_sql.Varchar2_Table;
  c number;
  dummy number;
begin
  for i in 0..9 loop
    emp_no_array(i) := 1000 + i;
    emp_addr_array(I) := get_new_addr(i);
  end loop;
  stmt := 'update emp set ename = :name_array
    where empno = :num_array';
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, stmt, dbms_sql.native);
  dbms_sql.bind_array(c, ':num_array', empno_array);
  dbms_sql.bind_array(c, ':name_array', empname_array);
  dummy := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);

  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

Here when the execute call happens, the addresses of all employees are updated in one shot. The two arrays are always stepped in unison. If the WHERE clause returns more than one row, all those employees will get the address the "addr_array" happens to be pointing to at the time.

Examples 6 and 7: Defining an Array The following examples show how to use the DEFINE_ARRAY procedure:

```
declare
  c       number;
```

```
  d       number;
  n_tab   dbms_sql.Number_Table;
  indx    number := -10;
begin
  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, 'select n from t order by 1', dbms_sql);

  dbms_sql.define_array(c, 1, n_tab, 10, indx);

  d := dbms_sql.execute(c);
  loop
    d := dbms_sql.fetch_rows(c);

    dbms_sql.column_value(c, 1, n_tab);

    exit when d != 10;
  end loop;

  dbms_sql.close_cursor(c);

  exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

Each time the example above does a FETCH_ROWS call, it fetches 10 rows that are kept in DBMS_SQL buffers. When the COLUMN_VALUE call is executed, those rows move into the PL/SQL table specified (in this case n_tab), at positions -10 to -1, as specified in the DEFINE statements. When the second batch is fetched in the loop, the rows go to positions 0 to 9; and so on.

A current index into each array is maintained automatically. This index is initialized to "indx" at EXECUTE and keeps getting updated every time a COLUMN_VALUE call is made. If you re-execute at any point, the current index for each DEFINE is re-initialized to "indx".

In this way the entire result of the query is fetched into the table. When FETCH_ROWS cannot fetch 10 rows, it returns the number of rows actually fetched (if no rows could be fetched it returns zero) and exits the loop.

Here is another example of using the DEFINE_ARRAY procedure:

Consider a table MULTI_TAB defined as:

```
create table multi_tab (num number,
                        dat1 date,
                        var varchar2(24),
                        dat2 date)
```

To select everything from this table and move it into four PL/SQL tables, you could use the following simple program:

```
declare
  c       number;
  d       number;
  n_tab  dbms_sql.Number_Table;
  d_tab1 dbms_sql.Date_Table;
  v_tab  dbms_sql.Varchar2_Table;
  d_tab2 dbms_sql.Date_Table;
  indx number := 10;
begin

  c := dbms_sql.open_cursor;
  dbms_sql.parse(c, 'select * from multi_tab order by 1', dbms_sql);

  dbms_sql.define_array(c, 1, n_tab,  5, indx);
  dbms_sql.define_array(c, 2, d_tab1, 5, indx);
  dbms_sql.define_array(c, 3, v_tab,  5, indx);
  dbms_sql.define_array(c, 4, d_tab2, 5, indx);

  d := dbms_sql.execute(c);

  loop
    d := dbms_sql.fetch_rows(c);

    dbms_sql.column_value(c, 1, n_tab);
    dbms_sql.column_value(c, 2, d_tab1);
    dbms_sql.column_value(c, 3, v_tab);
    dbms_sql.column_value(c, 4, d_tab2);

    exit when d != 5;
  end loop;

  dbms_sql.close_cursor(c);

/*
```

Here the four tables can be used for anything at all. One usage might be to use BIND_ARRAY to move the rows to another table by using a query such as 'INSERT into SOME_T values(:a, :b, :c, :d);

```
*/

exception when others then
    if dbms_sql.is_open(c) then
      dbms_sql.close_cursor(c);
    end if;
    raise;
end;
/
```

Example 8: Describe Columns This can be used as a substitute to the SQL*Plus DESCRIBE call by using a SELECT * query on the table that you want to describe.

```
declare
  c number;
  d number;
  col_cnt integer;
  f boolean;
  rec_tab dbms_sql.desc_tab;
  col_num number;
  procedure print_rec(rec in dbms_sql.desc_rec) is
  begin
    dbms_output.new_line;
    dbms_output.put_line('col_type            =    '
                         || rec.col_type);
    dbms_output.put_line('col_maxlen          =    '
                         || rec.col_max_len);
    dbms_output.put_line('col_name            =    '
                         || rec.col_name);
    dbms_output.put_line('col_name_len        =    '
                         || rec.col_name_len);
    dbms_output.put_line('col_schema_name     =    '
                         || rec.col_schema_name);
    dbms_output.put_line('col_schema_name_len =    '
                         || rec.col_schema_name_len);
    dbms_output.put_line('col_precision       =    '
                         || rec.col_precision);
    dbms_output.put_line('col_scale           =    '
                         || rec.col_scale);
    dbms_output.put('col_null_ok         =    ');
    if (rec.col_null_ok) then
```

```
      dbms_output.put_line('true');
    else
      dbms_output.put_line('false');
    end if;
  end;
begin
  c := dbms_sql.open_cursor;

  dbms_sql.parse(c, 'select * from scott.bonus', dbms_sql);

  d := dbms_sql.execute(c);

  dbms_sql.describe_columns(c, col_cnt, rec_tab);

/*
 * Following loop could simply be for j in 1..col_cnt loop.
 * Here we are simply illustrating some of the PL/SQL table
 * features.
 */
  col_num := rec_tab.first;
  if (col_num is not null) then
    loop
      print_rec(rec_tab(col_num));
      col_num := rec_tab.next(col_num);
      exit when (col_num is null);
    end loop;
  end if;

  dbms_sql.close_cursor(c);
end;
/
```

15

Dependencies Among Schema Objects

The definitions of certain schema objects, such as views and procedures, reference other schema objects, such as tables. Therefore, some schema objects are dependent upon the objects referenced in their definitions. This chapter discusses how to manage the dependencies among schema objects. Topics include the following:

- Dependency Issues
- Manually Recompiling
- Listing Dependency Management Information

> **See Also:** f you are using *Trusted Oracle*, also see the volume *Trusted Oracle* for information about handling dependencies among schema objects 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.

  > **See Also:** For more information on these commands, see *Oracle8 SQL Reference*.

  You can determine the dependencies among the schema objects in your database by running the SQL script UTLDTREE.SQL.

> **See Also:** 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 *Oracle8 Concepts*. 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 stand-alone 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 stand-alone 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 Also:** "Manually Recompiling" on page 15-4 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.

Remote Dependencies

Dependencies among PL/SQL library units (packages, stored procedures, and stored functions) can be handled either with timestamps or with signatures.

- In the timestamp method, the server sets a timestamp when each library unit is created or recompiled, and the compiled states of its dependent library units contain records of its timestamp. If the parent unit or a relevant schema object is altered, all of its dependent units are marked as invalid and must be recompiled before they can be executed.
- In the signature method, each compiled stored library unit is associated with a signature that identifies its name, the types and modes of its parameters, the number of parameters, and (for a function) the type of the return value. A dependent unit is marked as invalid if it calls a parent unit whose signature has been changed in an incompatible manner.

See Also: For more information, see "Remote Dependencies" on page 10-16.

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 15-6 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 (stand-alone), 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

> **See Also:** For a complete description of these data dictionary views, see *Oracle8 Reference.*

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 A temporary table used to store dependency information returned by the DEPTREE_FILL procedure.

Structure:

```
object_id NUMBER
referenced_object_id NUMBER
nest_level NUMBER
seq# NUMBER
```

View DEPTREE 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.

Column names: `nested_level, object_type, owner, object_name, seq#`

View IDEPTREE 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.

Column name: `dependencies`

Sequence DEPTREE_SEQ A sequence used to uniquely identify sets of dependency information stored in the DEPTREE_TEMPTAB.

Procedure DEPTREE_FILL 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.

Syntax:
```
DEPTREE_FILL(object_type CHAR, object_owner
CHAR, object_name CHAR)
```

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 Enterprise 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"
```

16

Signalling Database Events with Alerters

This chapter describes how to use the DBMS_ALERT package to provide notification, or “alerts”, of database events. Topics include the following:

- Overview
- Using Alerts
- Checking for Alerts
- Example of Using Alerts

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 (Romford) is active, refer to your Oracle Forms documentation.

The following procedures are callable from the DBMS_ALERT package:

Table 16–1 DBMS_ALERT Package Functions and Procedures

Function/Procedure	Description	Refer to
REGISTER	Receive messages from an alert.	page 16-5
REMOVE	Disable notification from an alert.	page 16-5
SIGNAL	Signal an alert (send message to registered sessions).	page 16-5
WAITANY	Wait TIMEOUT seconds to receive alert message from an alert registered for session.	page 16-6
WAITONE	Wait TIMEOUT seconds to receive message from named alert.	page 16-7
SET_DEFAULTS	Set the polling interval.	page 16-8

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 Also:** "Privileges Required" on page 10-59 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 16–2 shows the messages, and the procedures that can raise them.

Table 16–2 DBMS_ALERT Error Messages

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

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 Also: "Database Pipes" on page 12-2.

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 "ORA$" 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 16–3 . The syntax for this procedure is

```
DBMS_ALERT.SIGNAL(name     IN  VARCHAR2,
                  message  IN  VARCHAR2);
```

Table 16–3 DBMS_ALERT.SIGNAL Procedure Parameters

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.

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 16–4 . The syntax for this procedure is

```
DBMS_ALERT.WAITANY(name     OUT  VARCHAR2,
                   message  OUT  VARCHAR2,
                   status   OUT  INTEGER,
                   timeout   IN  NUMBER DEFAULT MAXWAIT);
```

Table 16–4 DBMS_ALERT.WAITANY Procedure Parameters

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 - time-out 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.

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.

Syntax

The parameters for the WAITONE procedure are described in Table 16–5 . The syntax for this procedure is

```
DBMS_ALERT.WAITONE(name      IN  VARCHAR2,
                   message  OUT  VARCHAR2,
                   status   OUT  INTEGER,
                   timeout   IN  NUMBER DEFAULT MAXWAIT);
```

Table 16–5 DBMS_ALERT.WAITONE Procedure Parameters

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 - time-out 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.

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.

17

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 Oracle* documentation 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 17-10 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 con-

tain 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 schema 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 17–1 summarizes the object privileges available for each type of object.

Table 17–1 Object Privileges

Object Privilege	Table	View	Sequence	Procedure (1))
ALTER	3		3	
DELETE	3	3		
EXECUTE				3
INDEX	3 (2)			
INSERT	3	3		
REFERENCES	3 (2)			
SELECT	3	3 (3)	3	
UPDATE	3	3		

- “Procedure” — refers to stand-alone stored procedures, functions, and public package constructs.
- "2" — privilege cannot be granted to a role.
- "3" — can also be granted for snapshots.
- Table 17–2lists the SQL statements permitted by the object privileges listed in Table 17–1.

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.

Table 17–2 SQL Statements Permitted by Database Object Privileges

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

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 "Explicitly Enabling Roles" on page 17-11 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 *Oracle8 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 Enterprise 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 “Restricting Application Roles from Tool Users” on page 17-5 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 *Oracle8 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 schema 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;
```

Schema 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 Schema Object Privileges

Grant schema 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 schema 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 schema object privileges in the same GRANT statement.

The GRANT OPTION A schema 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 schema object privilege to any user or any role in the database.
- The grantee can also grant the schema object privilege to other users, with or without the GRANT OPTION.
- If the grantee receives schema 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 schema object privileges with the GRANT OPTION.

> **Note:** The GRANT OPTION is not valid when granting a schema object privilege to a role. Oracle prevents the propagation of schema object privileges via roles so that grantees of a role cannot propagate object privileges received via roles.

Privileges Required to Grant Schema Object Privileges To grant a schema object privilege, the grantor must either

- be the owner of the schema object being specified, or
- have been granted the schema 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 Schema Object Privileges

Schema 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 a schema object privilege cannot be selectively revoked; the schema object privilege must be revoked and then regranted without the GRANT OPTION. A user cannot revoke schema object privileges from him or herself.

Revoking Column-Selective Schema 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 schema 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 Schema 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 Schema Object Privileges To revoke a schema 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).

Schema Object Privileges Revoking a schema object privilege can have several types of cascading effects that should be investigated before a REVOKE statement is issued:

- Schema 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.
- Schema 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 schema 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 schema 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 schema 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 schema object causes the "last DDL time" attribute of the object to change. This can invalidate any dependent schema objects, in particular PL / SQL package bodies that refer to the schema object. These then must be recompiled.

18

Oracle XA

This chapter describes how to use the Oracle XA library. The chapter includes the following topics:

- XA Library-Related Information
- Changes from Release 7.3 to Release 8.0
- General Issues and Restrictions
- Developing and Installing Applications That Use the XA Libraries
- Defining the xa_open String
- Interfacing to Precompilers and OCIs
- Transaction Control
- Migrating Precompiler or OCI Applications to TPM Applications
- XA Library Thread Safety
- Troubleshooting

XA Library-Related Information

General Information about the Oracle XA

For preliminary reading and additional reference information regarding the Oracle XA library, see the following documents:

- *Oracle Call Interface Programmer's Guide*

> **See Also:** For information on library linking filenames, see the Oracle operating system-specific documentation.

README.doc

A `README.doc` file is located in a directory specified in the Oracle operating system-specific documentation and describes changes, bugs, or restrictions in the Oracle XA library for your platform since the last version.

Changes from Release 7.3 to Release 8.0

The following changes have been made:

- Session Caching Is No Longer Needed
- Dynamic Registration Is Supported
- Loosely Coupled Transaction Branches Are Supported
- SQLLIB Is Not Needed for OCI Applications
- No Installation Script Is Needed to Run XA
- The XA Library Can Be Used with the Oracle Parallel Server Option on All Platforms
- Transaction Recovery for Oracle Parallel Server Has Been Improved
- Both Global and Local Transactions Are Possible
- The xa_open String Has Been Modified

Session Caching Is No Longer Needed

Session caching is unnecessary with the new OCI. Therefore, the old `xa_open` string parameter, `SesCacheSz`, has been eliminated. Consequently, you can also reduce the *sessions* `init.ora` parameter. Instead, set the *transactions* `init.ora` parameter to the expected number of concurrent global transactions. Because ses-

sions are not migrated when global transactions are resumed, applications must not refer to any session state beyond the scope of a service. For information on how to organize your application into services, refer to the documentation provided with the transaction processing monitor. In particular, savepoints and cursor fetch state will be cancelled when a transaction is suspended. This means that a savepoint taken by the application in a service will be invalid in another service, even though the two services may belong to the same global transaction.

Dynamic Registration Is Supported

Dynamic registration can be used if, and only if, both the XA application and the Oracle Server are Version 8.

See Also: "Extensions to the XA Interface" on page 18-13

Loosely Coupled Transaction Branches Are Supported

The Oracle8 Server supports both loosely and tightly coupled transaction branches in a single Oracle instance. The Oracle7 Server supported only tightly coupled transaction branches in a single instance, and loosely coupled transaction branches in different instances.

SQLLIB Is Not Needed for OCI Applications

OCI applications used to require the use of `SQLLIB`. This meant OCI programmers had to buy `SQLLIB`, even if they had no desire to develop Pro* applications. This is no longer the case.

No Installation Script Is Needed to Run XA

The SQL script `XAVIEW.SQL` is not needed to run XA applications in Oracle Version 8. It is, however, still necessary for Version 7.3 applications.

See Also: "Responsibilities of the DBA or System Administrator" on page 18-16.

The XA Library Can Be Used with the Oracle Parallel Server Option on All Platforms

It was not possible with Version 7 to use the Oracle XA library together with the Oracle Parallel Server option on certain platforms. Only if the platform's implementation of the distributed lock manager supported transaction-based rather than process-based locking would the two work together. This limitation is no longer the case; if you can run the Oracle Parallel Server option, then you can run the Oracle XA library.

Transaction Recovery for Oracle Parallel Server Has Been Improved

All transactions can be recovered from any instance of Oracle Parallel Server. Use the **xa_recover** call to provide a snapshot of the pending transactions.

Both Global and Local Transactions Are Possible

It is now possible to have both global and local transactions within the same XA connection. Local transactions are transactions that are completely coordinated by the Oracle Server. For example, the update below belongs to a local transaction.

```
CONNECT SCOTT/TIGER;
UPDATE EMP set sal = sal + 1; /* begin local transaction*/
COMMIT;                       /* commit local transaction*/
```

Global transactions, on the other hand, are coordinated by an external transaction manager such as a transaction processing monitor. In these transactions, the Oracle Server acts as a subordinate and processes the XA commands issued by the transaction manager. The update shown below belongs to a global transaction.

```
xa_open(oracle_xa+acc=p/SCOTT/TIGER+sestm=10", 1, TMNOFLAGS);
                            /*Transaction manager opens */
                            /* connection to the Oracle server*/
tpbegin();                  /* begin global transaction, the transaction*/
                            /*manager issues XA commands to the oracle*/
                            /*server to start a global transaction */
UPDATE EMP set sal = sal + 1;
                            /* Update is performed in the */
                            /* global transaction*/
tpcommit();                 /* commit global transaction, */
                            /* the transaction manager issues XA commands*/
                            /* to the Oracle server to commit */
                            /* the global transaction */
```

The Oracle7 Server forbids a local transaction from being started in an XA connection. The update shown below would return an `ORA-2041` error code.

```
xa_open("oracle_xa+acc=p/SCOTT/TIGER+sestm=10" , 1, TMNOFLAGS);
                            /* Transaction manager opens */
                            /*connection to the Oracle server */
UPDATE EMP set sal = sal + 1;/* Oracle 7 returns an error */
```

The Oracle8 Server, on the other hand, allows local transactions to be started in an XA connection. The only restriction is that the local transaction must be ended (committed or rolled back) before starting a global transaction in the connection.

See Also: *Oracle Call Interface Programmer's Guide* for more information on global transactions.

The xa_open String Has Been Modified

Two new parameters have been added. They are:

- `Loose_Coupling`

  This parameter has a boolean value and should not be set to true when connected to an Oracle7 Server. If set to true, it indicates that global transaction branches will be loosely coupled, that is, locks will not be shared between branches.

- `SesWt`

  This parameter's value indicates the time-out limit when waiting for a transaction branch that is being used by another session. If Oracle cannot switch to the transaction branch within `SesWt` seconds, `XA_RETRY` will be returned.

Two parameters have been made obsolete and should only be used when connected to an Oracle Server Release 7.3.

- `GPWD`

  The group password is not used by Oracle8. A session that is logged in with the same user name as the session that created a transaction branch will be allowed to switch to the transaction branch.

- `SesCacheSz`

  This parameter is not used by Oracle8 because session caching has been eliminated.

General Issues and Restrictions

Database Links

Oracle XA applications can access other Oracle Server databases through database links, with the following restrictions:

- Use the Multi-Threaded Server configuration.

  This means the transaction processing monitors (TPMs) will use shared servers to open the connection to Oracle. The O/S network connection required for the database link will be opened by the dispatcher instead of the Oracle server process. Thus, when a particular service or RPC completes, the transaction can be detached from the server so that it can be used by other services or RPCs.

- Access to the other database must use SQL*Net Version 2 or Net8.
- The other database being accessed should be another Oracle Server database.

Assuming that these restrictions are satisfied, Oracle Server will allow such links and will propagate the transaction protocol (prepare, rollback, and commit) to the other Oracle Server databases.

> **WARNING: If these restrictions are not satisfied, when you use database links within an XA transaction, it creates an O/S network connection in the Oracle Server that is connected to the TPM server process. Since this O/S network connection cannot be moved from one process to another, you cannot detach from this server. When you access a database through a database link, you will receive an ORA#24777 error.**

If using the Multi-Threaded Server configuration is not possible then, access the remote database through the Pro*C/C++ application using EXEC SQL AT syntax.

The parameter `open_links_per_instance` specifies the number of migratable open database link connections. These `dblink` connections are used by XA transactions so that the connections are cached after a transaction is committed. Another transaction is free to use the dblink connection provided the user that created the connection is the same as the user who created the transaction. This parameter is different from the `open_links` parameter, which is the number of dblink connections from a session. The `open_links` parameter is not applicable to XA applications.

Oracle Parallel Server Option

You can recover failed transactions from any instance of Oracle Parallel Server. You can also heuristically commit in-doubt transactions from any instance. An XA recover call will give a list of all prepared transactions for all instances.

SQL-based Restrictions

SQL-based restrictions are listed in this section.

Rollbacks and Commits

Because the transaction manager is responsible for coordinating and monitoring the progress of the global transaction, the application should not contain any Oracle Server-specific statement that independently rolls back or commits a global transaction. However, you can use rollbacks and commits in a local transaction.

Do not use EXEC SQL ROLLBACK WORK for precompiler applications when you are in the middle of a global transaction. Similarly, an OCI application should not execute OCITransRollback(), or the Version 7 equivalent orol(). You can roll back a global transaction by calling tx_rollback().

Similarly, a precompiler application should not have the EXEC SQL COMMIT WORK statement in the middle of a global transaction. An OCI application should not execute OCITransCommit() or the Version 7 equivalent ocom(). Instead, use tx_commit() or tx_rollback() to end a global transaction.

DDL Statements

Because a DDL SQL statement such as CREATE TABLE implies an implicit commit, the Oracle XA application cannot execute any DDL SQL statements.

Session State

Oracle does not guarantee that session state will be valid between services. For example, if a service updates a session variable (such as a global package variable), another service that executes as part of the same global transaction may not see the change. Use savepoints only within a service. The application must not refer to a savepoint that was created in another service. Similarly, an application must not attempt to fetch from a cursor that was executed in another service.

SET TRANSACTION

Do not use the SET TRANSACTION READ ONLY | READ WRITE | USE ROLLBACK SEGMENT SQL statement.

Connecting or Disconnecting with EXEC SQL

Do not use the EXEC SQL command to connect or disconnect. That is, do not use EXEC SQL COMMIT WORK RELEASE or EXEC SQL ROLLBACK WORK RELEASE.

Miscellaneous XA Issues

Note the following additional information about Oracle XA:

Transaction Branches

Oracle Server transaction branches within the same global transaction can share locks in either a tightly or loosely coupled manner. However, if the branches are on different instances when running Oracle Parallel Server, then they will be loosely coupled.

In tightly coupled transaction branches, the locks are shared between the transaction branches. This means that updates performed in one transaction branch can be seen in other branches that belong to the same global transaction before the update is committed. The Oracle Server obtains the DX lock before executing any statement in a tightly coupled branch. Hence, the advantage of using loosely coupled transaction branches is that there will be more concurrency (because a lock is not obtained before the statement is executed). The disadvantage is that all the transaction branches must go through the two phases of commit, that is, XA one phase optimization cannot be used. These trade-offs between tightly coupled branches and loosely coupled branches are illustrated in Table 18–1.

Table 18–1 Tightly and Loosely Coupled Tranaction Branches

Attribute	Tightly Coupled Branches	Loosely Coupled Branches
Two Phase Commit	Read-only Optimization [prepare for all branches, commit for last branch]	Two phases [prepare and commit for all branches]
Serialization	Database Call	None

Association Migration

The Oracle Server does not support association migration (a means whereby a transaction manager may resume a suspended branch association in another branch).

Asynchronous Calls

The optional XA feature asynchronous XA calls is not supported.

Initialization Parameters

Set the `transactions init.ora` parameter to the expected number of concurrent global transactions.

The parameter `open_links_per_instance` specifies the number of migratable open database link connections. These dblink connections are used by XA transactions so that the connections are cached after a transaction is committed.

See Also: "Database Links" on page 18-6 for further information.

Maximum Connections per Thread

The maximum number of `xa_opens` per thread is now 32. Previously, it had been 8.

Installation

No scripts need be executed to use XA. It is necessary, however, to run the xaview.sql script to run Release 7.3 applications with the Oracle8 Server. Grant the `SELECT` privilege on `SYS.DBA_PENDING_TRANSACTIONS` to all users that connect to Oracle through the XA interface.

Compatibility

The XA library supplied with Release 7.3 can be used with a Release 8.0 Oracle Server. You must use the Release 7.2 XA library with a Release 7.2 Oracle Server. You can use the 8.0 library with a Release 7.3 Oracle Server. There is only one case of backward compatibility: an XA application that uses Release 8.0 OCI will work with a Release 7.3 Oracle Server, but only if you use `sqlld2` and obtain an `lda_def` before executing SQL statements. Client applications must remember to convert the Version 7 LDA to a service handle using `OCILdaToSvcCtx`() after completing the OCI calls.

Basic Architecture

The Oracle XA library is an external interface that allows global transactions to be coordinated by a transaction manager other than the Oracle8 Server. This allows inclusion of non-Oracle8 Server entities called resource managers (RM) in distributed transactions.

The Oracle XA library conforms to the X/Open Distributed Transaction Processing (DTP) software architecture's XA interface specification.

> **See Also:** For a general overview of XA, including basic architecture, see *X/Open CAE Specification - Distributed Transaction Processing: The XA Specification*. You can obtain a copy of this document by requesting X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3 from:
>
> - X/Open Company, Ltd., 1010 El Camino Real, Suite 380, Menlo Park, CA 94025, U.S.A.

X/Open Distributed Transaction Processing(DTP)

The X/Open DTP architecture defines a standard architecture or interface that allows multiple application programs to share resources, provided by multiple, and possibly different, resource managers. It coordinates the work between application programs and resource managers into global transactions.

Figure 18–1 illustrates a possible X/Open DTP model.

A resource manager (RM) controls a shared, recoverable resource that can be returned to a consistent state after a failure. For example, Oracle8 Server is an RM and uses its redo log and undo segments to return to a consistent state after a failure. An RM provides access to shared resources such as a database, file systems, printer servers, and so forth.

A transaction manager (TM) provides an application program interface (API) for specifying the boundaries of the transaction and manages the commit and recovery procedures.

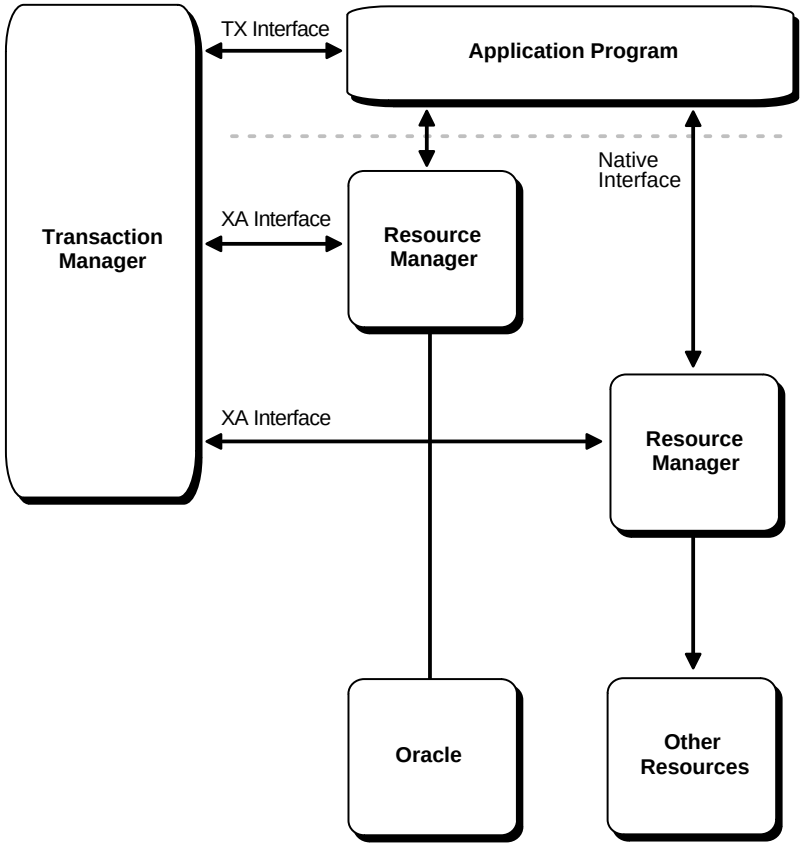
Normally, Oracle8 Server acts as its own TM and manages its own commit and recovery. However, using a standards-based TM allows Oracle8 Server to cooperate with other heterogeneous RMs in a single transaction.

A TM is usually a component provided by a transaction processing monitor (TPM) vendor. The TM assigns identifiers to transactions, and monitors and coordinates their progress. It uses Oracle XA library subroutines to tell Oracle8 Server how to

process the transaction, based on its knowledge of all RMs in the transaction. You can find a list of the XA subroutines and their descriptions later in this section.

An application program (AP) defines transaction boundaries and specifies actions that constitute a transaction. For example, an AP can be a precompiler or OCI program. The AP operates on the RM's resource through the RM's native interface, for example SQL. However, it starts and completes all transaction operations via the transaction manager through an interface called TX. The AP itself does not directly use the XA interface

Figure 18–1 One Possible DTP Model



Note: The naming conventions for the TX interface and associated subroutines are vendor-specific, and may differ from those used here. For example, you may find that the `tx_open` call is referred to as `tp_open` on your system. To check terminology, see the documentation supplied with the transaction processing monitor.

Transaction Recovery Management

The Oracle XA library interface follows the two-phase commit protocol, consisting of a prepare phase and a commit phase, to commit transactions.

In phase one, the prepare phase, the TM asks each RM to guarantee the ability to commit any part of the transaction. If this is possible, then the RM records its prepared state and replies affirmatively to the TM. If it is not possible, the RM may roll back any work, reply negatively to the TM, and forget any knowledge about the transaction. The protocol allows the application, or any RM, to roll back the transaction unilaterally until the prepare phase is complete.

In phase two, the commit phase, the TM records the commit decision. Then the TM issues a commit or rollback to all RMs which are participating in the transaction.

Note: a TM can issue a commit for an RM only if all RMs have replied affirmatively to phase one.

Oracle XA Library Interface Subroutines

The Oracle XA library subroutines allow a TM to instruct an Oracle8 Server what to do about transactions. Generally, the TM must "open" the resource (using `xa_open`). Typically, this will result from the AP's call to `tx_open`**.** Some TMs may call `xa_open` implicitly, when the application begins. Similarly, there is a close (using `xa_close`) that occurs when the application is finished with the resource. This may be when the AP calls `tx_close` or when the application terminates.

There are several other tasks the TM instructs the RMs to do. These include among others:

- starting a new transaction and associating it with an ID
- rolling back a transaction
- preparing and committing a transaction

XA Library Subroutines

The following XA Library subroutines are available:

`xa_open`	Connects to the resource manager.
`xa_close`	Disconnects from the resource manager.
`xa_start`	Starts a new transaction and associate it with the given transaction ID (XID), or associates the process with an existing transaction.
`xa_end`	Disassociates the process from the given XID.
`xa_rollback`	Rolls back the transaction associated with the given XID.
`xa_prepare`	Prepares the transaction associated with the given XID. This is the first phase of the two-phase commit protocol.
`xa_commit`	Commits the transaction associated with the given XID. This is the second phase of the two-phase commit protocol.
`xa_recover`	Retrieves a list of prepared, heuristically committed or heuristically rolled back transaction.
`xa_forget`	Forgets the heuristic transaction associated with the given XID.

In general, the AP does not need to worry about these subroutines except to understand the role played by the `xa_open` string.

Extensions to the XA Interface

Two functions have been added to the XA interface, one for returning the OCI service handle associated with an XA connection, and one for returning an XA error code.

1. `OCISvcCtx *xaoSvcCtx(text *dbname)`:

   This function returns the OCI service handle for a given XA connection. The dbname parameter must be the same as the dbname parameter passed in the `xa_open` string. OCI applications can use this routing instead of the `sqlld2` calls to obtain the connection handle. Hence, OCI applications need not link with the SQLLIB library. The service handle can be converted to the Version 7 OCI logon data area(LDA) using `OCISvcCtxToLda`() [Version 8 OCI]. Client applications must remember to convert the Version 7 LDA to a service handle using `OCILdaToSvcCtx`() after completing the OCI calls.

2. `OCIEnv *xaoEnv(text *dbname)`:

This function returns the OCI environment handle for a given XA connection. The dbname parameter must be the same as the dbname parameter passed in the `xa_open` string.

3. `int xaosterr(OCISvcCtx *SvcCtx, sb4 error)`:

   This function, only applicable to dynamic registration, converts an Oracle error code to an XA error code. The first parameter is the service handle used to execute the work in the database. The second parameter is the error code that was returned from Oracle. Use this function to determine if the error returned from an OCI command was caused because the `xa_start` failed. The function returns `XA_OK` if the error was not generated by the XA module and a valid XA error if the error was generated by the XA module.

Transaction Processing Monitors (TPMs)

A transaction processing monitor (TPM) coordinates the flow of transaction requests between the client processes that issue requests and the back-end servers that process them. Basically, it coordinates transactions that require the services of several different types of back-end processes, such as application servers and resource managers that are distributed over a network.

The TPM synchronizes any commits and rollbacks required to complete a distributed transaction. The transaction manager (TM) portion of the TPM is responsible for controlling when distributed commits and rollbacks take place. Thus, if a distributed application program is written to take advantage of a TPM, the TM portion of the TPM is responsible for controlling the two-phase commit protocol. The RMs enable the TMs to do this.

Because the TM controls distributed commits or rollbacks, it must communicate directly with Oracle (or any other resource manager) through the Oracle XA library interface.

Required Public Information

As a resource manager, Oracle is required to publish the following information.

`xa_switch_t` structures	The Oracle Server `xa_switch_t` structure name for static registration is `xaosw`. The Oracle Server `xa_switch_t` structure name for dynamic registration is `xaoswd`. These structures contain entry points and other information for the resource manager.
`xa_switch_t` resource mgr	The Oracle Server resource manager name within the `xa_switch_t` structure is `Oracle_XA`.

close string	The close string used by xa_close () is ignored and is allowed to be null.
open string	The format of the open string used by xa_open () is described in detail in “Developing and Installing Applications That Use the XA Libraries” on page 18-16.
libraries	Libraries needed to link applications using Oracle XA have operating system-specific names. It is similar to linking an ordinary precompiler or OCI program except you may have to link any TPM-specific libraries. If you are not using sqllib, make sure to link with $ORACLE_HOME/lib/xaonsl.o.
requirements	A purchased and installed distributed database option.

Registration

Dynamic and static registration are supported by the Oracle8 Server. The basic possibilities are shown in Table 18–2.

Table 18–2 XA Registration

Client	Server	XA Registration
8.0 XA application	8.0	Dynamic
8.0 XA application	7.3	Static
7.3 XA application	8.0	Static

Developing and Installing Applications That Use the XA Libraries

This section discusses developing and installing Oracle8 Server applications. It describes the responsibilities of both the DBA, or system administrator, and the application developer. It also defines how to construct the open string.

Responsibilities of the DBA or System Administrator

The responsibilities of the DBA or system administrator are

1. Define the open string with the application developer's help.

   This is described in "Defining the xa_open String" on page 18-17.

2. Make sure the DBA_PENDING_TRANSACTIONS view exists on the database.

   For Oracle Server Release 7.3:

   Make sure V$XATRANS$ exists.

   This view should have been created during the XA library installation. You can manually create the view if needed by running the SQL script XAVIEW.SQL. This SQL script should be executed as the Oracle user SYS. Grant the SELECT privilege to the V$XATRANS$ view for all Oracle Server accounts which will be used by Oracle XA library applications.

   See Also: Your Oracle operating system-specific documentation contains the location of the XAVIEW.SQL script.

   For Oracle Server Release 8.0:

   Grant the select privilege to the DBA_PENDING_TRANSACTIONS view for all Oracle Server *user*(s) specified in the xa_open string.

3. Install the resource manager, using the open string information, into the TPM configuration, following the TPM vendor instructions.

   The DBA or system administrator should be aware that a TPM system will start the process that connects to an Oracle8 Server. See your TPM documentation to determine what environment exists for the process and what user ID it will have.

   Be sure that correct values are set for ORACLE_HOME and ORACLE_SID.

   Next, grant the user ID write permission to the directory in which the XA trace file will be written. See "Defining the xa_open String" on page 18-17 for information on how to specify a *sid* or a trace directory that is different from the defaults.

Also be sure to grant the user the SELECT privilege on DBA_PENDING_TRANSACTIONS.

4. Start up the relevant databases to bring Oracle XA applications on-line.

   This should be done before starting any TPM servers.

Responsibilities of the Application Developer

The application developer's responsibilities are

1. Define the open string with the DBA or system administrator's help.

   Defining the open string is described later in this section.

2. Develop the applications.

   Observe special restrictions on transaction-oriented SQL statements for precompilers. See "Interfacing to Precompilers and OCIs" on page 18-22.

3. Link the application according to TPM vendor instructions.

Defining the xa_open String

The open string is used by the transaction monitor to open the database. The maximum number of characters in an open string is 256.

This section covers:

- Syntax of the xa_open String
- Required Fields
- Optional Fields

Syntax of the xa_open String

Oracle_XA{+*required_fields*...} [+*optional_fields*...]
where *required_fields* are:

Acc=P//

or

Acc=P/*user/password*

SesTm=*session_time_limit*

and where *optional_fields* are:

`DB`=*db_name*

`LogDir`=*log_dir*

`MaxCur`=*maximum_#_of_open_cursors*

`SqlNet`=*connect_string*

`Loose_Coupling`=*true/false*

`SesWt`=*session_wait_limit*

`Threads`=*true/false*

Note:

- You can enter the required fields and optional fields *in any order* when constructing the open string.
- All field names are case insensitive. Their values may or may not be case-sensitive depending on the platform.
- There is no way to use the "+" character as part of the actual information string.

Required Fields

Required fields for the open string are described in this section.

`Acc=P//`

or

Acc=P / *user/password*

`Acc`	Specifies user access information
`P`	Indicates that explicit user and password information is provided.
`P//`	Indicates that no explicit user or password information is provided and that the operating system authentication form will be used. For more information see *Oracle8 Administrator's Guide.*
user	A valid Oracle Server account.
password	The corresponding current password.

For example, `Acc=P/scott/tiger` indicates that user and password information is provided. In this case, the user is `scott` and the password is `tiger`.

As previously mentioned, make sure that `scott` has the `SELECT` privilege on the `DBA_PENDING_TRANSACTIONS` table.

`Acc=P//` indicates that no user or password information is provided, thus defaulting to operating system authentication.

`SesTm=`*session_time_limit*

`SesTm`	Specifies the maximum length of time a transaction can be inactive before it is automatically aborted by the system.
session_time_limit	This value should be the maximum time allowed in a transaction between one service and the next, or a service and the commit or rollback of the transaction. For example, if the TPM uses remote procedure calls between the client and the servers, then SesTM applies to the time between the completion of one RPC and the initiation of the next RPC, or the `tx_commit`, or the `tx_rollback`. The unit for this time limit is in seconds. The value of 0 indicates no limit, but entering a value of 0 is strongly discouraged. For example, `SesTM=15` indicates that the session idle time limit is 15 seconds.

Optional Fields

Optional fields are described below.

DB=*db_name*

DB	Specifies the database name.
db_name	Indicates the name used by Oracle precompilers to identify the database. Application programs that use only the default database for the Oracle precompiler (that is, they do not use the AT clause in their SQL statements) should omit the DB=*db_name* clause in the open string. Applications that use explicitly named databases should indicate that database name in their DB=*db_name* field. Version 7 OCI programs need to call the sqlld2() function to obtain the correct lda_def, which is the equivalent of a service context. Version 8 OCI programs need to call the xaoSvcCtx function to get the OCISvcCtx service context. The *db_name* is not the *sid* and is not used to locate the database to be opened. Rather, it correlates the database opened by this open string with the name used in the application program to execute SQL statements. The *sid* is set from either the environment variable ORACLE_SID of the TPM application server or the *sid* given in the Net8 (formerly, SQL*Net) clause in the open string. The Net8 clause is described later in this section. Some TPM vendors provide a way to name a group of servers that use the same open string. The DBA may find it convenient to choose the same name both for that purpose and for *db_name*.

For example, DB=payroll indicates that the database name is “payroll”, and that the application server program will use that name in AT clauses.

`LogDir=`*log_dir*

LogDir	Specifies the directory on a local machine where the Oracle XA library error and tracing information may be logged.
log_dir	Indicates the pathname of the directory where the tracing information should be stored. The default is `$ORACLE_HOME/rdbms/log` if ORACLE_HOME is set, otherwise it is the current directory.

For example, `LogDir=/xa_trace` indicates that the error and tracing information is located under the `/xa_trace` directory.

Note: Ensure that the directory you specify for logging exists and the application server can write to it.

`MaxCur=`*maximum_#_of_open_cursors*

MaxCur	Specifies the number of cursors to be allocated when the database is opened. It serves the same purpose as the precompiler option `maxopencursors`.
maximum_#_of_open_cursors	Indicates the number of open cursors to be cached.

For example, `MaxCur=5` indicates that the precompiler should try to keep five open cursors cached.

Note: This parameter overrides the precompiler option `maxopencursors` that you might have specified in your source code or at compile time.

See Also: Chapter 8 in *Pro*C/C++ Precompiler Programmer's Guide*. for more information on `maxopencursors`.

SqlNet=*db_link*

SqlNet	Specifies the Net8 (formerly, SQL*Net) database link.
db_link	Indicates the string to use to log on to the system. The syntax for this string is the same as that used to set the TWO-TASK environment variable.

For example, `SqlNet=hqfin@NEWDB` indicates the database with *sid*=`NEWDB` accessed at host `hqfin` by TCP/IP.

The `SqlNet` parameter can be used to specify the `ORACLE_SID` in cases where you cannot control the server environment variable. It must also be used when the server needs to access more than one Oracle Server database. To use the Net8 string without actually accessing a remote database, use the Pipe driver.

For example:

```
SqlNet=localsid1
```

where:

`localsid1`	is an alias defined in the Net8 `tnsnames.ora` file.

Make sure that all databases to be accessed with a Net8 database link have an entry in `/etc/oratab`.

`Loose_Coupling`=*true/false*

`Loose_Coupling`	Specifies whether locks will be shared between branches. This parameter should not be set to true when connected to an Oracle7 Server. The default value is False.
true/false	If locks are shared between branches, the setting is *false*.

`SesWt`=*session_wait_limit*

`SesWt`	Specifies the time-out limit when waiting for a transaction branch that is being used by another session. The default value is 60 seconds.
session_wait_limit	The number of seconds Oracle will wait before XA_RETRY is returned.

`Threads`=*true/false*

Threads	Specifies whether the application is multi-threaded. The default value is False.
true/false	If the application is multi-threaded, the setting is *true*.

Interfacing to Precompilers and OCIs

This section describes how to use the Oracle XA library with precompilers and Oracle Call Interfaces (OCIs).

Using Precompilers with the Oracle XA Library

When used in an Oracle XA application, cursors are valid only for the duration of the transaction. Explicit cursors should be opened after the transaction begins, and closed before the commit or rollback.

There are two options to choose from when interfacing with precompilers:

- using precompilers with the default database
- using precompilers with a named database

The following examples use the precompiler Pro*C/C++.

Using Precompilers with the Default Database

To interface to a precompiler with the default database, make certain that the DB=*db_name* field, used in the open string, is not present. The absence of this field indicates the default connection, and only one default connection is allowed per process.

The following is an example of an open string identifying a default Pro*C/C++ connection.

```
ORACLE_XA+SqlNet=host@MAIL+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/logs
```

Note that the DB=d*b_name* is absent, indicating an empty database ID string.

The syntax of a SQL statement would be:

```
EXEC SQL UPDATE EMP SET SAL = sal*1.5;
```

Using Precompilers with a Named Database

To interface to a precompiler with a named database, include the DB=*db_name field* in the open string. Any database you refer to must reference the same *db_name* you specified in the corresponding open string.

An application may include the default database, as well as one or more named databases, as shown in the following examples.

For example, suppose you want to update an employee's salary in one database, his department number (DEPTNO) in another, and his manager in a third database. You would configure the following open strings in the transaction manager:

```
ORACLE_XA+DB=MANAGERS+SqlNet=hqfin@SID1+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+DB=PAYROLL+SqlNet=SID2+ACC=P/scott/tiger
```

```
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+SqlNet=hqemp@SID3+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/xalog
```

Note that there is no DB=*db_name* field in the last open string.

In the application server program, you would enter declarations such as:

```
EXEC SQL DECLARE PAYROLL DATABASE;
EXEC SQL DECLARE MANAGERS DATABASE;
```

Again, the default connection (corresponding to the third open string that does not contain the db_name field) needs no declaration.

When doing the update, you would enter statements similar to the following:

```
EXEC SQL AT PAYROLL UPDATE EMP SET SAL=4500 WHERE EMPNO=7788;
EXEC SQL AT MANAGERS UPDATE EMP SET MGR=7566 WHERE EMPNO=7788;
EXEC SQL UPDATE EMP SET DEPTNO=30 WHERE EMPNO=7788;
```

There is no AT clause in the last statement because it is referring to the default database.

In Oracle precompilers Release 1.5.3 or later, you can use a character host variable in the AT clause, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
  DB_NAME1 CHARACTER(10);
  DB_NAME2 CHARACTER(10);
EXEC SQL END DECLARE SECTION;
    .
    .
SET DB_NAME1 = 'PAYROLL'
SET DB_NAME2 = 'MANAGERS'
    .
    .
EXEC SQL AT :DB_NAME1 UPDATE...
EXEC SQL AT :DB_NAME2 UPDATE...
```

WARNING: Oracle recommends against using XA applications to create connections. Any work performed would be outside the global transaction, and would have to be committed separately.

Using OCI with the Oracle XA Library

OCI applications that use the Oracle XA library should not call `OCISessionBegin`() (`olon`() or `orlon`() in Version 7) to log on to the resource manager. Rather, the logon should be done through the TPM. The applications can execute the function `xaoSvcCtx`() (`sqlld2`() in Version 7) to obtain the service context (`lda` in Version 7) structure they need to access the resource manager.

Because an application server can have multiple concurrent open Oracle Server resource managers, it should call the function `xaoSvcCtx`() with the correct arguments to obtain the correct service context.

For Release 7.3

If `DB=`*db_name* is not present in the open string, then execute:

```
sqlld2(lda, NULL, 0);
```

to obtain the `lda` for this resource manager.

Alternatively, if `DB=`*db_name* is present in the open string, then execute:

```
sqlld2(lda, db_name, strlen(db_name));
```

to obtain the `lda` for this resource manager.

For Release 8.0

If `DB=`*db_name* is not present in the open string, then execute:

```
xaoSvcCtx(NULL);
```

to obtain the `xaoSvcCtx` for this resource manager.

Alternatively, if `DB=`*db_name* is present in the open string, then execute:

```
xaoSvcCtx(db_name);
```

to obtain the `OCISvcCtx` for this resource manager.

> **Note Also:** *Oracle Call Interface Programmer's Guide*. for more information about using the `OCISvcCtx`

Transaction Control

This section explains how to use transaction control within the Oracle XA library environment.

When the XA library is used, transactions are not controlled by the SQL statements which commit or roll back transactions. Rather, they are controlled by an API accepted by the TM which starts and stops transactions. Most of the TMs use the TX interface for this. It includes the following functions:

`tx_open`	logs into the resource manager(s)
`tx_close`	logs out of the resource manager(s)
`tx_begin`	starts a new transaction
`tx_commit`	commits a transaction
`tx_rollback`	rolls back the transaction

Most TPM applications are written using a client-server architecture where an application client requests services and an application server provides services. The examples that follow use such a client-server model. A service is a logical unit of work, which in the case of the Oracle Server as the resource manager, comprises a set of SQL statements that perform a related unit of work.

For example, when a service named “credit” receives an account number and the amount to be credited, it will execute SQL statements to update information in certain tables in the database. In addition, a service might request other services. For example, a “transfer fund” service might request services from a “credit” and “debit” service.

Usually application clients request services from the application servers to perform tasks within a transaction. However, for some TPM systems, the application client itself can offer its own local services.

You can encode transaction control statements within either the client or the server; as shown in the examples.

To have more than one process participating in the same transaction, the TPM provides a communication API that allows transaction information to flow between the participating processes. Examples of communications APIs include RPC, pseudo-RPC functions, and send / receive functions.

Because the leading vendors support different communication functions, the examples that follow use the communication pseudo-function `tpm_service` to generalize the communications API.

X/Open has included several alternative methods for providing communication functions in their preliminary specification. At least one of these alternatives is supported by each of the leading TPM vendors.

Examples of Precompiler Applications

The following examples illustrate precompiler applications. Assume that the application servers have already logged onto the TPM system, in a TPM-specific manner.

The first example shows a transaction started by an application server, and the second example shows a transaction started by an application client.

Example 1: Transaction started by an application server

Client:

```
tpm_service("ServiceName");              /*Request Service*/
```

Server:

```
ServiceName()
{
<get service specific data>
tx_begin();                              /* Begin transaction boundary*/
EXEC SQL UPDATE ....;

/*This application server temporarily becomes*/
/*a client and requests another service.*/

tpm_service("AnotherService");
tx_commit();                              /*Commit the transaction*/
<return service status back to the client>
}
```

Example 2: Transaction started by an application client.

Client:

```
tx_begin();                             /* Begin transaction boundary */
tpm_service("Service1");
tpm_service("Service2");
tx_commit();                            /* Commit the transaction */
```

Server:

```
Service1()
```

```
{
<get service specific data>
EXEC SQL UPDATE ....;
<return service status back to the client>
}
Service2()
{
<get service specific data>
EXEC SQL UPDATE ....;
...
<return service status back to client>
}
```

Migrating Precompiler or OCI Applications to TPM Applications

To migrate existing precompiler or OCI applications to a TPM application using the Oracle XA library, you must do the following:

1. Reorganize the application into a framework of "services".

   This means that application clients request services from application servers.

   Some TPMs require the application to use the `tx_open` and `tx_close` functions, whereas other TPMs do the logon and logoff implicitly.

   If you do not specify the `sqlnet` parameter in your open string, the application will use the default Net8 driver. Thus, you need to be sure that the application server is brought up with the `ORACLE_HOME` and `ORACLE_SID` environment variables properly defined. This is accomplished in a TPM-specific fashion. See your TPM vendor documentation for instructions on how to accomplish this.

2. Ensure that the application replaces the regular connect and disconnect statements.

   For example, replace the connect statements `EXEC SQL CONNECT` (for precompilers) or `OCISessionBegin`() (for OCIs) by `tx_open`(). Replace the disconnect statements `EXEC SQL COMMIT / ROLLBACK RELEASE WORK` (for precompilers), or `OCISessionEnd`() (for OCIs) by `tx_close( )`. The V7 equivalent for `OCISessionBegin`() was `olon( )` and for `OCISessionEnd`(), `ologof`().

3. Ensure that the application replaces the regular commit / rollback statements and begins the transaction explicitly.

For example, replace the commit / rollback statements `EXEC SQL COMMIT` / `ROLLBACK WORK` (for precompilers), or `ocom()/orol()` (for OCIs) by `tx_commit()/tx_rollback()` and start the transaction by calling `tx_begin()`.

4. Ensure that the application resets the fetch state prior to ending a transaction. In general, `release_cursor=no` should be used. Use `release_cursor=yes` only when certain that a statement will be executed only once.

Table 18–3 lists the TPM functions that replace regular Oracle commands when migrating precompiler or OCI applications to TPM applications.

Table 18–3 TPM Replacement Commands

Regular Oracle Commands	TPM Functions
`CONNECT` *user/password*	`tx_open` (possibly implicit)
implicit start of transaction	`tx_begin`
`SQL`	service that executes the SQL
`COMMIT`	`tx_commit`
`ROLLBACK`	`tx_rollback`
disconnect	`tx_close` (possibly implicit)
`SET TRANSACTION READ ONLY`	illegal

XA Library Thread Safety

If you use a transaction monitor that supports threads, the Oracle XA library allows you to write applications that are thread safe. Certain issues must be kept in mind, however.

A *thread of control* (or thread) refers to the set of connections to resource managers. In an unthreaded system, each process could be considered a thread of control, since each process has its own set of connections to resource managers and each process maintains its own independent resource manager table.

In a threaded system, each thread has an autonomous set of connections to resource managers and each thread maintains a *private* resource manager table. This private resource manager table must be allocated for each new thread and de-allocated when the thread terminates, even if the termination is abnormal.

Note: In an Oracle system, once a thread has been started and establishes a connection, only that thread can use that connection. No other thread can make a call on that connection.

The Open String Specification

The `xa_open` string parameter, `xa_info`, provides the clause, Threads=, which must be specified as true to enable the use of threads by the transaction monitor. The default is false. Note that, in most cases, threads will be created by the transaction monitor and that the application will not know when a new thread is created. Therefore, it is advisable to allocate a service context (`lda` in Version 7) on the stack within each service that is written for a transaction monitor application. Before doing any Oracle-related calls in that service, the `xaoSvcCtx` (`sqlld2` for Version 7 OCI) function must be called and the service context initialized. This LDA can then be used for all OCI calls within that service.

Restrictions

The following restrictions apply when using threads:

- Any Pro* or OCI code that executes as part of the application server process on the transaction monitor cannot be threaded unless the transaction monitor is explicitly told when each new application thread is started. This is typically accomplished by using a special C compiler provided by the transaction monitor vendor.
- The Pro* statements, EXEC SQL ALLOCATE and EXEC SQL USE are not supported. Therefore, when threading is enabled, embedded SQL statements cannot be used across non-XA connections.

Troubleshooting

This section discusses how to find information in case of problems or system failure. It also discusses trace files and recovery of pending transactions.

Trace Files

The Oracle XA library logs any error and tracing information to its trace file. This information is useful in supplementing the XA error codes. For example, it can indicate whether an `xa_open` failure is caused by an incorrect open string, failure to find the Oracle Server instance, or a logon authorization failure.

The name of the trace file is:

xa_*db_namedate*.trc

where *db_name* is the database name you specified in the open string field DB=*db_name*, and *date* is the date when the information is logged to the trace file.

If you do not specify DB=*db_name* in the open string, it automatically defaults to the name "NULL".

The xa_open string DbgFl

Normally, the XA trace file is opened only if an error is detected. The xa_open string DbgFl provides a tracing facility to record additional detail about the XA library. By default, its value is zero. It can be set to any combination of the following values. Note that they are independent, so to get printout from two or more flags, each must be set.

- 0x1 Trace the entry and exit to each procedure in the XA interface. This can be useful in seeing exactly what XA calls the TP Monitor is making and what transaction identifier it is generating.
- 0x2 Trace the entry to and exit from other non-public XA library routines. This is generally of use only to Oracle developers.
- 0x4 Trace various other "interesting" calls made by the XA library, such as specific calls to the Oracle Call Interface. This is generally of use only to Oracle developers.

Trace File Locations

The trace file can be placed in one of the following locations:

- The trace file can be created in the LogDir directory as specified in the open string.
- If you do not specify LogDir in the open string, then the Oracle XA application attempts to create the trace file in the $ORACLE_HOME/rdbms/log directory, if it can determine where $ORACLE_HOME is located.
- If the Oracle XA application cannot determine where $ORACLE_HOME is located, then the trace file is created in the current working directory.

Trace File Examples

Examples of two types of trace files are discussed below:

The example, xa_NULL040292.trc, shows a trace file that was created on April 2, 1992. Its DB field was not specified in the open string when the resource manager was opened.

The example, xa_Finance121591.trc, shows a trace file was created on December 15, 1991. Its DB field was specified as "Finance" in the open string when the resource manager was opened.

Note: multiple Oracle XA library resource managers with the same DB field and LogDir field in their open strings log all trace information that occurs on the same day to the same trace file.

Each entry in the trace file contains information that looks like this:

```
1032.12345.2:  ORA-01017:  invalid username/password;  logon denied
1032.12345.2:  xaolgn:  XAER_INVAL;  logon denied
```

where "1032" is the time when the information is logged, "12345" is the process ID (PID), "2" is the resource manager ID, xaolgn is the module name, XAER_INVAL was the error returned as specified in the XA standard, and ORA-1017 is the Oracle Server information that was returned.

In-doubt or Pending Transactions

In-doubt or pending transactions are transactions that have been prepared, but not yet committed to the database.

Generally, the transaction manager provided by the TPM system should resolve any failure and recovery of in-doubt or pending transactions. However, the DBA may have to override an in-doubt transaction in certain circumstances, such as when the in-doubt transaction is:

- locking data that is required by other transactions
- not resolved in a reasonable amount of time

For more information about overriding in-doubt transactions in the circumstances described above, or about how to decide whether the in-doubt transaction should be committed or rolled back, see the TPM documentation.

Oracle Server SYS Account Tables

There are four tables under the Oracle Server SYS account that contain transactions generated by regular Oracle Server applications and Oracle XA applications. They

are DBA_PENDING_TRANSACTIONS, V$GLOBAL_TRANSACTIONS, DBA_2PC_PENDING and DBA_2PC_NEIGHBORS

For transactions generated by Oracle XA applications, the following column information applies specifically to the DBA_2PC_NEIGHBORS table.

- the DBID column is always xa_orcl
- the DBUSER_OWNER column is always *db_name*xa.oracle.com

Remember that the *db_name* is always specified as DB=*db_name* in the open string. If you do not specify this field in the open string, then the value of this column is NULLxa.oracle.com for transactions generated by Oracle XA applications.

For example, you could use the SQL statement below to obtain more information about in-doubt transactions generated by Oracle XA applications.

```
SELECT * FROM DBA_2PC_PENDING p, DBA_2PC_NEIGHBORS n
  WHERE p.LOCAL_TRAN_ID = n.LOCAL_TRAN_ID
    AND
    n.DBID = 'xa_orcl';
```

Alternatively, if you know the format ID used by the transaction processing monitor, you can use DBA_PENDING_TRANSACTIONS or V$GLOBAL_TRANSACTIONS. While DBA_PENDING_TRANSACTIONS gives a list of both active and failed prepared transactions, V$GLOBAL_TRANSACTIONS gives a list of all active global transactions.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y