Oracle® Database

Application Developer's Guide - Fundamentals

10*g* Release 1 (10.1)

Part No. B10795-01

December 2003

ORACLE®

Oracle Database Application Developer's Guide - Fundamentals, 10*g* Release 1 (10.1)

Part No. B10795-01

Primary Authors: Drew Adams, Eric Paapanen

Contributing Authors: M. Cowan, R. Moran, J. Russell, R. Strohm

Contributors: D. Alpern, G. Arora, C. Barclay, D. Bronnikov, T. Chang, M. Davidson, G. Doherty, D. Elson, A. Ganesh, M. Hartstein, J. Huang, N. Jain, R. Jenkins Jr., S. Kotsovolos, S. Kumar, C. Lei, D. Lorentz, R. Murthy, R. Pang, B. Sinha, S. Vemuri, W. Wang, D. Wong, A. Yalamanchi, Q. Yu

Graphic Designer: V. Moore

Contents

Part II Designing the Database

2 Selecting a Datatype

3 Maintaining Data Integrity Through Constraints

4 Selecting an Index Strategy

5 How Oracle Database Processes SQL Statements

6 Coding Dynamic SQL Statements

7 Using Procedures and Packages

8 Calling External Procedures

Part III The Active Database

9 Using Triggers

13 Developing Web Applications with PL/SQL

16 Using Oracle XA with Transaction Monitors

Send Us Your Comments

Oracle Database Application Developer's Guide - Fundamentals, 10*g* Release 1 (10.1)

Part No. B10795-01

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

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

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

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

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

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

Preface

The *Oracle Database Application Developer's Guide - Fundamentals* describes basic application development features of Oracle Database 10*g*. Information in this guide applies to features that work the same on all supported platforms, and does not include system-specific information.

This preface contains these topics:

Audience

Oracle Database Application Developer's Guide - Fundamentals is intended for programmers developing new applications or converting existing applications to run in the Oracle Database environment. This book 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.

Duties of an Application Developer

Activities that are typically required of an application developer include:

- Programming in SQL. Your primary source of information for this is the *Oracle Database SQL Reference*. In the *Oracle Data Warehousing Guide*,you can find information about advanced query techniques, to perform analysis and retrieve data in a single query,
- Interfacing to SQL through other languages, such as PL/SQL, Java, or C/C++. Sources of information about these other languages include:
  - *PL/SQL User's Guide and Reference*
  - *PL/SQL Packages and Types Reference*
  - *Oracle Database Java Developer's Guide*
  - *Pro*C/C++ Programmer's Guide*
  - *Oracle Call Interface Programmer's Guide* and *Oracle C++ Call Interface Programmer's Guide*
  - *Oracle Objects for OLE C++ Class Library Developer's Guide*
  - *Oracle COM Automation Feature Developer's Guide*
- Setting up interactions and mappings between multiple language environments, as described in "Calling External Procedures" on page 8-1.
- Working with schema objects. You might design part or all of a schema, and write code to fit into an existing schema. You can get full details in *Oracle Database Administrator's Guide*.
- Interfacing with the database administrator to make sure that the schema can be backed up and restored, for example after a system failure or when moving between a staging machine and a production machine.
- Building application logic into the database itself, in the form of stored procedures, constraints, and triggers, to allow multiple applications to reuse application logic and code that checks and cleans up errors. For information on these database features, see "Using Procedures and Packages" on page 7-1, "Maintaining Data Integrity Through Constraints" on page 3-1, and "Using Triggers" on page 9-1.
- Some degree of performance tuning. The database administrator might help here. You can find more information in *PL/SQL User's Guide and Reference*, *PL/SQL Packages and Types Reference*, and *Oracle Database Performance Tuning Guide*.

- Some amount of database administration, if you need to maintain your own development or test system. You can learn about administration in the *Oracle Database Administrator's Guide*.
- Debugging and interpreting error messages. See "Related Documentation" on page xxxi.
- Making your application available over the network, particularly over the Internet or company intranet. You can get an overview in "Developing Web Applications with PL/SQL" on page 13-1, and full details covering various languages and technologies in the Oracle Application Server documentation.
- Designing the class structure and choosing object-oriented methodologies, if your application is object-oriented. For more information, see *Oracle Database Application Developer's Guide - Object-Relational Features*, *PL/SQL User's Guide and Reference*, and *Oracle Database Java Developer's Guide*.

Organization

This document contains:

Part I: Introduction

This part introduces several ways that you can write Oracle Database applications. You might need to use more than one language or development environment for a single application. Some database features are only supported by, or are easier to access from, certain languages.

Chapter 1, "Programmatic Environments"

This chapter outlines the strengths of the languages, development environments, and APIs that Oracle Database provides.

Part II: Designing the Database

Before you develop an application, you need to plan the characteristics of the associated database. You must choose all the pieces that go into the database, and how they are put together. Good database design helps ensure good performance and scalability, and reduces the amount of application logic you code by making the database responsible for things like error checking and fast data access.

Chapter 2, "Selecting a Datatype"

This chapter explains how to represent your business data in the database. The datatypes include fixed- and variable-length character strings, numeric data, dates, raw binary data, and row identifiers (`ROWIDs`).

Chapter 3, "Maintaining Data Integrity Through Constraints"

This chapter explains how to use constraints to move error-checking logic out of your application and into the database.

Chapter 4, "Selecting an Index Strategy"

This chapter explains how to choose the best indexing strategy for your application.

Chapter 5, "How Oracle Database Processes SQL Statements"

This chapter explains SQL topics such as commits, cursors, and locking that you can take advantage of in your applications.

Chapter 6, "Coding Dynamic SQL Statements"

This chapter describes dynamic SQL, compares native dynamic SQL to the DBMS_SQL package, and explains when to use dynamic SQL.

Chapter 7, "Using Procedures and Packages"

This chapter explains how to store reusable procedures in the database, and how to group procedures into packages.

Chapter 8, "Calling External Procedures"

This chapter explains how to code the bodies of computation intensive procedures in languages other than PL/SQL.

Part III: The Active Database

You can include all sorts of programming logic in the database itself, making the benefits available to many applications and saving repetitious coding work.

Chapter 9, "Using Triggers"

This chapter explains how to make the database do special processing before, after, or instead of running SQL statements. You can use triggers for things like logging database access and validating or transforming data.

Chapter 10, "Working With System Events"

This chapter explains how to retrieve information, such as the user ID and database name, about the event that fires a trigger.

Chapter 11, "Using the Publish-Subscribe Model for Applications"

This chapter introduces the Oracle Database model for asynchronous communication, also known as messaging or queuing.

Part IV: Developing Specialized Applications

Chapter 12, "Using Regular Expressions With Oracle Database"

This chapter discusses regular expression support built into Oracle Database, regular expression syntax, and how to write queries using regular expressions in SQL.

Chapter 13, "Developing Web Applications with PL/SQL"

This chapter explains how to create dynamic Web pages and applications that work with the Internet, e-mail, and so on, using the PL/SQL language.

Chapter 14, "Porting Non-Oracle Applications to Oracle Database 10g"

This chapter lists features and techniques you can use to make applications run on Oracle Database 10*g* that were originally written for another, non-Oracle database.

Chapter 15, "Using Flashback Features"

This chapter describes how to use features that let you examine past data and its history, and to recover that data.

Chapter 16, "Using Oracle XA with Transaction Monitors"

This chapter describes how to connect Oracle Database with a transaction monitor.

Related Documentation

For more information, see these Oracle resources.

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

- The Oracle Call Interface (OCI) is described in *Oracle Call Interface Programmer's Guide* and *Oracle C++ Call Interface Programmer's Guide*.

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

- The *Oracle Database Security Guide* discusses security features of the database that application developers and database administrators need to be aware of.

- Oracle 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 C, C++, COBOL, or FORTRAN that incorporate embedded SQL, then refer to the corresponding precompiler manual. For example, if you program in C or C++, then refer to the *Pro*C/C++ 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. Refer to the appropriate Oracle Developer/2000 documentation if you use this product.

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

- For developing applications that manipulate XML data, see *Oracle XML Developer's Kit Programmer's Guide* and *Oracle XML DB Developer's Guide*.

- Oracle Database error message documentation is available only in HTML. If you have access only to the Oracle Documentation CD, you can browse the error messages by range. After you find the specific range, use your browser's "find in page" feature to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle Database online documentation.

- Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

- Printed documentation is available for sale in the Oracle Store at

  `http://oraclestore.oracle.com/`

- To download free release notes, installation documentation, white papers, or other collateral, visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

http://otn.oracle.com/membership/

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

http://otn.oracle.com/documentation/

Conventions

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

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

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

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executable, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
lowercase monospace (fixed-width font) italic	Lowercase monospace italic font represents placeholders or variables.	You can specify the *parallel_clause*. Run `U`*old_release*`.SQL` where *old_release* refers to the release you installed prior to upgrading.

Conventions in Code Examples

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

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*digits* `[ ,` *precision* `])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
. . .	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

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

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to

evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

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

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

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

What's New in Application Development?

The following sections give an overview of new application development features introduced in this release and some previous releases of the database. Related documentation on each feature is cross-referenced when it is available.

New Application Development Features in Oracle Database 10*g* Release 1

This section discusses new features introduced in Oracle Database 10*g* Release 1 (10.1).

- **Regular Expression Support**

  A set of SQL functions introduced in this release let you perform queries and manipulate string data using regular expressions. See Chapter 12, "Using Regular Expressions With Oracle Database" for more information.

- **Oracle Expression Filter**

  Oracle Expression Filter lets you store conditional expressions in a column that you can use in the WHERE clause of a database query. See "Representing Conditional Expressions as Data" on page 2-32 for more information.

  See Also: *Oracle Database SQL Reference*

- **Native floating-point datatypes**

  Column datatypes BINARY_FLOAT and BINARY_DOUBLE are introduced in this release. These datatypes provide an alternative to using the Oracle `NUMBER` datatype, with the following benefits:

  - More efficient use of storage resources

- Faster arithmetic operations
- Support for numerical algorithms specified in the IEEE 754 Standard

Support for native floating-point datatypes in bind and fetch operations is provided for the following client interfaces:

- SQL
- PL/SQL
- OCI
- OCCI
- Pro*C/C++
- JDBC

See Also: "Representing Numeric Data with Number and Floating-Point Datatypes" on page 2-11

- **Terabyte-Size Large Object (LOB) support**

  This release provides support for terabyte-size LOB values (from 8 to 128 terabytes) in the following programmatic environments:

  - Java (JDBC)
  - OCI
  - PL/SQL (package `DBMS_LOB`)

  You can store and manipulate LOB (`BLOB`, `CLOB`, and `NCLOB`) datatypes larger than 4GB.

  See Also: For details on terabyte-size LOB support:

  - *Oracle Database Application Developer's Guide - Large Objects*
  - *Oracle Call Interface Programmer's Guide*

- **Flashback**

  This release has new and enhanced flashback features. You can now do the following:

  - Query the transaction history of a row.

- Obtain the SQL undo syntax for a row, to perform row-level flashback operations.
- Perform remote queries of past data.

  See Also: Chapter 15, "Using Flashback Features"

- **Oracle Data Provider for .NET**

  Oracle Data Provider for .NET (ODP.NET) is a new programmatic environment that implements a data provider for Oracle Database. It uses APIs native to Oracle Database to offer fast and reliable access from any .NET application to database features and data. ODP.NET also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library.

  See Also: *Oracle Data Provider for .NET Developer's Guide*

New Application Development Features in Oracle9*i* Release 2

This section gives an overview of application development features introduced in Oracle9*i* Release 2 (9.2).

- **Enhancements to Flashback Query**

  You can perform an Oracle Flashback Query using the `AS OF` clause of the `SELECT` statement rather than going through the `DBMS_FLASHBACK` package. This technique is very flexible, allowing you to perform joins, set operations, subqueries, and views using different date/time or SCN settings for each table in the query. You can also restore or capture past data by using a Flashback Query inside an `INSERT` or `CREATE TABLE AS SELECT` statement.

  See Also: "Using Flashback Features" on page 15-1

- **Using PL/SQL Records in INSERT and UPDATE Statements**

  When you represent related data items using a PL/SQL record, you can perform insert and update operations using the entire record, instead of specifying each record field separately.

  See Also: *PL/SQL User's Guide and Reference*

- **Ability to rename constraints**

If a data management application experiences problems because it tries to create a constraint when the constraint already exists, you can rename the existing constraint to avoid the conflict. If you track down a constraint with a cryptic system-generated name, you can give it a descriptive name to make it easier to enable and disable later.

> **See Also:** "Renaming Integrity Constraints" on page 3-24

- **Enhanced support for NCHAR, NVARCHAR2, and NCLOB types**

  These globalization-support types can now be used as attributes of SQL and PL/SQL object types, and in PL/SQL collection types such as varrays and nested tables.

- **New XML programming capabilities**

  New and enhanced built-in types, such as XMLType and XDBURIType, let you delegate XML parsing, storage, and retrieval to the database.

  > **See Also:** *Oracle XML DB Developer's Guide*

- **Enhanced UTL_FILE package**

  The `UTL_FILE` package has a number of new functions for performing popular file operations. You can seek, auto-flush, read and write binary data, delete files, change file permissions, and more. Use the `CREATE DIRECTORY` statement (using double quotation marks around any lowercase names), rather than the `UTL_FILE_DIR` initialization parameter.

  > **See Also:** *PL/SQL Packages and Types Reference* for details about these enhancements

- **User-defined constructors**

  You can now override the system default constructor for an object type with your own constructor function.

  > **See Also:** *PL/SQL User's Guide and Reference*

- **Access to LOB data within triggers**

  You can access or change LOB data within `BEFORE` and `INSTEAD OF` triggers, using the `:NEW` variable.

See Also: "Example: Modifying LOB Columns with a Trigger" on page 9-18

- **Synonyms for types**

  You can now define synonyms for types.

  See Also: *Oracle Database Administrator's Guide* for details on creating synonyms

- **Scrollable cursors in Pro*C/C++ applications**

  Scrollable cursors let you move forward and backward through the result set in a Pro*C/C++ application.

  See Also: "Highlights of Pro*C/C++ Features" on page 1-21

- **Support for Connection Pooling in Pro*C/C++**

  The Connection Pool feature in Pro*C/C++ helps you optimize the performance of Pro*C/C++ applications.

  See Also: "Highlights of Pro*C/C++ Features" on page 1-21

- **Better linking in online documentation**

  Many of the cross-references from this book to other books have been made more specific, so that they link to a particular place within another book rather than to its table of contents. If you are reading a printed copy of this book, you can find the online equivalent, with full search capability, at `http://otn.oracle.com/documentation/`.

Java Features Removed from the database in Oracle9*i* Release 2

This section discusses Java features that were removed from the database in Oracle9*i* Database release 2 (version 9.2.0). Support for some of these features was moved from the database to Oracle Application Server.

The following Java features and related technologies are no longer supported as integrated components of the database:

- The J2EE stack, comprising:
  - Enterprise Java Beans (EJB) Container

- Oracle JavaServer Pages engine (OJSP)
- Oracle Servlet Engine (OSE)

- Common Object Request Broker Architecture (CORBA) framework

Oracle Application Server now includes Oracle Application Server Containers for J2EE (OC4J). Migrate any existing applications that use the following technologies in the database to OC4J: servlets, JSP pages, EJBs, and CORBA objects.

To develop new applications using EJBs or CORBA, you must use the J2EE components that are part of Oracle Application Server. EJBs and CORBA are no longer supported within the database.

You can still access the database from these components using Oracle Application Server as a middle-tier. You can still write Java stored procedures and Java methods for object types within database applications.

For more information on OC4J, visit the Oracle Application Server documentation pages at:

`http://otn.oracle.com/documentation/`

See Also:

- *Oracle Database JDBC Developer's Guide and Reference*

New Application Development Features in Oracle9*i* Release 1

This section gives an overview of application development features introduced in Oracle9*i* Release 1 (9.0.1).

- **Integration of SQL and PL/SQL parsers**

  PL/SQL now supports the complete range of syntax for SQL statements, such as `INSERT`, `UPDATE`, `DELETE`, and so on. If you received errors for valid SQL syntax in PL/SQL programs before, those statements should now work.

  Because of more consistent error-checking, you might find that some invalid code is now found at compile time instead of producing an error at runtime, or vice versa. You might need to change the source code as part of the migration procedure.

  See Also: *Oracle Database Upgrade Guide* for details on the complete migration procedure

- **Resumable Storage Allocation**

  When an application encounters some kinds of storage allocation errors, it can suspend operations and take action such as resolving the problem or notifying an operator. The operation can be resumed when storage is added or freed.

  See Also: "Resuming Execution After a Storage Error Condition" on page 5-38

- **Flashback**

  This release has new and enhanced flashback features. You can now do the following:

  - Query the transaction history of a row.
  - Obtain the SQL undo syntax for a row, to perform row-level flashback operations.
  - Perform remote queries of past data.

  See Also: "Using Flashback Features" on page 15-1

- **WITH Clause for Reusing Complex Subqueries**

  Rather than repeat a complex subquery, you can give it a name and refer to that name multiple times within the same query. This is convenient for coding; it also helps the optimizer find common code that can be optimized.

  See Also: *Oracle Database Administrator's Guide* for details on creating synonyms

- **New Date and Time Types**

  The new datatype `TIMESTAMP` records time values including fractional seconds. New datatypes `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` allow you to adjust date and time values to account for time zone differences. You can specify whether the time zone observes daylight savings time. New datatypes `INTERVAL DAY TO SECOND` and `INTERVAL YEAR TO MONTH` represent differences between two date and time values, simplifying date arithmetic.

See Also:

- "Summary of Oracle Built-In Datatypes" on page 2-2
- "Representing Date and Time Data" on page 2-20

- **Better Integration of LOB Datatypes**

  You can use character functions on `CLOB` and `NCLOB` types. You can treat `BLOB` types as `RAW`s. Conversions between LOBs and other types are much simpler now, particularly when converting from `LONG` to LOB types.

  See Also:

  - "Representing Large Amounts of Data" on page 2-34
  - "How Oracle Database Converts Datatypes" on page 2-40
  - "Representing Character Data" on page 2-8

- **Improved Globalization and National Language Support**

  Data can be stored in Unicode format using fixed-width or variable-width character sets. String handling and storage declarations can be specified using character lengths, where the number of bytes is computed for you, or explicit byte lengths. You can set up the entire database to use the same length semantics for strings, or specify the settings for individual procedures; this setting is remembered if a procedure is invalidated.

  See Also: "Representing Character Data" on page 2-8

- **Enhancements to Bulk Operations**

  You can now perform bulk SQL operations, such as bulk fetches, using native dynamic SQL (the `EXECUTE IMMEDIATE` statement). You can perform bulk insert or update operations that continue despite errors on some rows, then examine the individual row problems after the operation is complete.

  See Also: "Overview of Bulk Binds" on page 7-17

- **Improved Support for PL/SQL Web Applications**

  The `UTL_HTTP` and `UTL_SMTP` packages have a number of enhancements, such as letting you access password-protected Web pages, and sending e-mail with attachments.

See Also: Chapter 13, "Developing Web Applications with PL/SQL" on page 13-1

- **Native Compilation of PL/SQL Code**

  Improve performance by compiling Oracle-supplied and user-written stored procedures into native executables, using typical C development tools. This setting is saved, so that the procedure is compiled the same way if it is later invalidated.

  See Also: "Compiling PL/SQL Procedures for Native Execution" on page 7-21

- **Oracle C++ Call Interface (OCCI) API**

  The OCCI API lets you write fast, low-level database applications using C++. It is similar to the Oracle Call Interface (OCI) API.

  See Also: "Overview of OCI and OCCI" on page 1-25

- **Secure Application Roles**

  In Oracle9*i*, application developers no longer need to secure a role by embedding passwords inside applications. They can create application roles and specify which PL/SQL package is authorized to enable the roles. Application roles enabled by PL/SQL packages are called **secure application roles**.

- **Creating Application Contexts**

  You can create an application context by entering a command like:

  ```
  CREATE CONTEXT Order_entry USING Apps.Oe_ctx;
  ```

  Alternatively, you can use Oracle Policy Manager to create an application context.

- **Dedicated External Procedure Agents**

  You can run external procedure agents (the `EXTPROC` entry in `tnsnames.ora`) under different instances of Oracle Database or on entirely separate machines. This lets you configure external procedures more robustly, so that if one external procedure fails, other external procedures can continue running in a different agent process.

See Also:

- "Loading External C Procedures" on page 8-5
- "Publishing External Procedures" on page 8-10

Part I

Introduction to Application Development Features of Oracle Database

This part introduces application development features of Oracle Database.

It contains the following chapter:

1

Programmatic Environments

This chapter contains these topics:

Overview of Developing an Oracle Database Application

As an application developer, you have many choices when it comes to writing a program to interact with the database.

Client/Server Model

In a traditional client/server program, the code of your application runs on a machine other than the database server. Database calls are transmitted from this client machine to the database server. Data is transmitted from the client to the server for insert and update operations, and returned from the server to the client for query operations. The data is processed on the client machine. Client/server programs are typically written using precompilers, where SQL statements are embedded within the code of another language such as C, C++, or COBOL.

Server-Side Coding

You can develop application logic that resides entirely inside the database, using triggers that are executed automatically when changes occur in the database, or stored procedures that are called explicitly. Off-loading the work from your application lets you reuse code that performs verification and cleanup, and control database operations from a variety of clients. For example, by making stored procedures callable through a Web server, you can construct a Web-based user interface that performs the same functions as a client/server application.

Two-Tier Versus Three-Tier Models

Client/server computing is often referred to as a **two-tier model**: your application communicates directly with the database server. In the **three-tier model**, another server (known as the **application server**) processes the requests. The application server might be a basic Web server, or might perform advanced functions like caching and load-balancing. Increasing the processing power of this middle tier lets you lessen the resources needed by client systems, resulting in a **thin client** configuration where the client machine might need only a Web browser or other means of sending requests over the TCP/IP or HTTP protocols.

User Interface

The interface that your application displays to end users depends on the technology behind the application, as well as the needs of the users themselves. Experienced users might enter SQL commands that are passed on to the database. Novice users might be shown a graphical user interface that uses the graphics libraries of the

client system (such as Windows or X-Windows). Any of these traditional user interfaces can also be provided in a Web browser using HTML and Java.

Stateful Versus Stateless User Interfaces

In traditional client/server applications, the application can keep a record of user actions and use this information over the course of one or multiple sessions. For example, past choices can be presented in a menu so that they do not have to be entered again. When the application is able to save information like this, we refer to the application as **stateful**.

Web or thin-client applications that are **stateless** are easier to develop. This means that they gather all the required information, process it using the database, and then start over from the beginning with the next user. This is a popular way to process single-screen requests such as customer registration.

There are many ways to add stateful behavior to Web applications that are stateless by default. For example, an entry form on one Web page can pass information on to subsequent Web pages, allowing you to construct a wizard-like interface that remembers the user's choices through several different steps. Cookies can be used to store small items of information on the client machine, and retrieve them when the user returns to a Web site. Servlets can be used to keep a database session open and store variables between requests from the same client.

Overview of PL/SQL

PL/SQL is Oracle's procedural extension to SQL, the standard database access language. An advanced 4GL (fourth-generation programming language[1]), PL/SQL offers seamless SQL access, tight integration with Oracle Database and associated tools, portability, security, and modern software engineering features such as data encapsulation, overloading, exception handling, and information hiding.

With PL/SQL, you can manipulate data with SQL statements, and control program flow with procedural constructs such as IF-THEN and LOOP. You can also declare constants and variables, define procedures and functions, use collections and object types, and trap run-time errors.

Applications written using any of the Oracle programmatic interfaces can call PL/SQL stored procedures and send blocks of PL/SQL code to the server for execution. 3GL (third-generation programming language[2]) applications can access

[1] **4GL**: An "application specific" language, with built-in treatment of an application domain. PL/SQL and SQL have built-in treatment of the database domain.

PL/SQL scalar and composite datatypes through host variables and implicit datatype conversion.

Because it runs inside the database, PL/SQL code is very efficient for data-intensive operations, and minimizes network traffic in client/server applications.

PL/SQL's tight integration with Oracle Developer lets you develop the client and server components of your application in the same language, then partition the components for optimal performance and scalability. Also, Oracle's Web Forms lets you deploy your applications in a multitier Internet or intranet environment without modifying a single line of code.

See Also: *PL/SQL User's Guide and Reference*

A Simple PL/SQL Example

The procedure `debit_account` takes money from a bank account. It accepts an account number and an amount of money as parameters. It uses the account number to retrieve the account balance from the database, then computes the new balance. If this new balance is less than zero, the procedure jumps to an error routine; otherwise, it updates the bank account.

```
PROCEDURE debit_account (acct_id INTEGER, debit_amount REAL) IS
   old_balance REAL;
   new_balance REAL;
   overdrawn   EXCEPTION;
BEGIN
   SELECT bal INTO old_balance FROM accts
      WHERE acct_no = acct_id;
   new_balance := old_balance - debit_amount;
   IF new_balance < 0 THEN
      RAISE overdrawn;
   ELSE
      UPDATE accts SET bal = new_balance
         WHERE acct_no = acct_id;
   END IF;
   COMMIT;
EXCEPTION
   WHEN overdrawn THEN
      -- handle the error
END debit_account;
```

[2] **3GL:** A language designed to be easier than assembler language for a human to understand. It includes things like named variables. Unlike 4GL, it is not specific to a particular application domain.

Advantages of PL/SQL

PL/SQL is a completely portable, high-performance transaction processing language that offers the following advantages:

Full Support for SQL

PL/SQL lets you use all the SQL data manipulation, cursor control, and transaction control commands, as well as all the SQL functions, operators, and pseudocolumns. So, you can manipulate Oracle Database data flexibly and safely. PL/SQL fully supports SQL datatypes, reducing conversions as data is passed between applications and the database.

Dynamic SQL is a programming technique that lets you build and process SQL statements "on the fly" at run time. It gives PL/SQL flexibility comparable to scripting languages such as Perl, Korn shell, and Tcl.

Tight Integration with Oracle Database

PL/SQL supports all the SQL datatypes. Combined with the direct access that SQL provides, these shared datatypes integrate PL/SQL with the Oracle Database data dictionary.

The `%TYPE` and `%ROWTYPE` attributes let your code adapt as table definitions change. For example, the `%TYPE` attribute declares a variable based on the type of a database column. If the column's type changes, your variable uses the correct type at run time. This provides data independence and reduces maintenance costs.

Better Performance

If your application is database intensive, you can use PL/SQL blocks to group SQL statements before sending them to Oracle Database for execution. This can drastically reduce the communication overhead between your application and Oracle Database.

PL/SQL stored procedures are compiled once and stored in executable form, so procedure calls are quick and efficient. A single call can start a compute-intensive stored procedure, reducing network traffic and improving round-trip response times. Executable code is automatically cached and shared among users, lowering memory requirements and invocation overhead.

Higher Productivity

PL/SQL adds procedural capabilities, such as Oracle Forms and Oracle Reports. For example, you can use an entire PL/SQL block in an Oracle Forms trigger instead of multiple trigger steps, macros, or user exits.

PL/SQL is the same in all environments. As soon as you master PL/SQL with one Oracle tool, you can transfer your knowledge to others, and so multiply the productivity gains. For example, scripts written with one tool can be used by other tools.

Scalability

PL/SQL stored procedures increase scalability by centralizing application processing on the server. Automatic dependency tracking helps you develop scalable applications.

The shared memory facilities of the shared server (formerly known as Multi-Threaded Server or MTS) enable Oracle Database to support many thousands of concurrent users on a single node. For more scalability, you can use the Oracle Connection Manager to multiplex network connections.

Maintainability

Once validated, a PL/SQL stored procedure can be used with confidence in any number of applications. If its definition changes, only the procedure is affected, not the applications that call it. This simplifies maintenance and enhancement. Also, maintaining a procedure on the server is easier than maintaining copies on various client machines.

PL/SQL Support for Object-Oriented Programming

Object Types An object type is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called attributes. The functions and procedures that characterize the behavior of the object type are called methods, which you can implement in PL/SQL.

Object types are an ideal object-oriented modeling tool, which you can use to reduce the cost and time required to build complex applications. Besides allowing you to create software components that are modular, maintainable, and reusable, object types allow different teams of programmers to develop software components concurrently.

Collections A collection is an ordered group of elements, all of the same type (for example, the grades for a class of students). Each element has a unique subscript that determines its position in the collection. PL/SQL offers two kinds of collections: nested tables and varrays (short for variable-size arrays).

Collections work like the set, queue, stack, and hash table data structures found in most third-generation programming languages. Collections can store instances of an object type and can also be attributes of an object type. Collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms. You can define collection types in a PL/SQL package, then use the same types across many applications.

Portability

Applications written in PL/SQL can run on any operating system and hardware platform where Oracle Database runs. You can write portable program libraries and reuse them in different environments.

Security

PL/SQL stored procedures let you divide application logic between the client and the server, to prevent client applications from manipulating sensitive Oracle Database data. Database triggers written in PL/SQL can prevent applications from making certain updates, and can audit user queries.

You can restrict access to Oracle Database data by allowing users to manipulate it only through stored procedures that have a restricted set of privileges. For example, you can grant users access to a procedure that updates a table, but not grant them access to the table itself.

> **See Also:** *Oracle Database Security Guide* for details on database security features

Built-In Packages for Application Development

- `DBMS_PIPE` is used to communicate between sessions.
- `DBMS_ALERT` is used to broadcast alerts to users.
- `DBMS_LOCK` and `DBMS_TRANSACTION` are used for lock and transaction management.
- `DBMS_AQ` is used for Advanced Queuing.
- `DBMS_LOB` is used to manipulate large objects.

- `DBMS_ROWID` is used for employing `ROWID` values.
- `UTL_RAW` is the RAW facility.
- `UTL_REF` is for work with `REF` values.

Built-In Packages for Server Management

- DBMS_SESSION is for session management by DBAs.
- DBMS_SPACE and DBMS_SHARED_POOL provide space information and reserve shared pool resources.
- DBMS_JOB is used to schedule jobs in the server.

Built-In Packages for Distributed Database Access

These provide access to snapshots, advanced replication, conflict resolution, deferred transactions, and remote procedure calls.

Overview of Java Support Built Into the Database

This section gives an overview of built-in database features that support Java applications. The database includes the core JDK libraries such as `java.lang`, `java.io`, and so on. The database supports client-side Java standards such as JDBC and SQLJ, and provides server-side JDBC and SQLJ drivers that allow data-intensive Java code to run within the database.

See Also:

- *Oracle Database Concepts* for background information about Java and how the database supports it
- *Oracle Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database JPublisher User's Guide*

Overview of Oracle JVM

Oracle JVM, the Java Virtual Machine provided with the Oracle Database, is compliant with the J2SE version 1.4.x specification and supports the database session architecture.

Any database session can activate a dedicated JVM. All sessions share the same JVM code and statics; however, private states for any given session are held, and subsequently garbage collected, in an individual session space.

This design provides the following benefits:

- Java applications have the same session isolation and data integrity as SQL operations.
- There is no need to run Java in a separate process for data integrity.
- Oracle JVM is a robust JVM with a small memory footprint.
- The JVM has the same linear SMP scalability as the database and can support thousands of concurrent Java sessions.

Oracle JVM works consistently with every platform supported by Oracle Database. Java applications that you develop using Oracle JVM can be ported to any supported platform easily.

Oracle JVM includes a deployment-time native compiler that enables Java code to be compiled once, stored in executable form, shared among users, and invoked more quickly and efficiently.

Security features of the database are also available using Oracle JVM. Java classes must be loaded in a database schema (using Oracle JDeveloper, a third-party IDE, SQL*Plus, or the loadjava utility) before they can be invoked. Java class invocation is secured and controlled through database authentication and authorization, Java 2 security, and invoker's or definer's rights.

Overview of Oracle Extensions to JDBC

JDBC (Java Database Connectivity) is an API (Applications Programming Interface) that allows Java to send SQL statements to an object-relational database such as Oracle Database.

The JDBC standard defines four types of JDBC drivers:

- Type 1. A JDBC-ODBC bridge. Software must be installed on client systems.
- Type 2. Has Native methods (calls C or C++) and Java methods. Software must be installed on the client.
- Type 3. Pure Java. The client uses sockets to call middleware on the server.
- Type 4. The most pure Java solution. Talks directly to the database using Java sockets.

JDBC is based on the X/Open SQL Call Level Interface, and complies with the SQL92 Entry Level standard.

You can use JDBC to do dynamic SQL. Dynamic SQL means that the embedded SQL statement to be executed is not known before the application is run, and requires input to build the statement.

The drivers that are implemented by Oracle have extensions to the capabilities in the JDBC standard that was defined by Sun Microsystems. Oracle's implementations of JDBC drivers are described next. Oracle Database support of and extensions to various levels of the JDBC standard are described in "Oracle Database Extensions to JDBC Standards" on page 1-11.

JDBC Thin Driver

The JDBC thin driver is a Type 4 (100% pure Java) driver that uses Java sockets to connect directly to a database server. It has its own implementation of a Two-Task Common (TTC), a lightweight implementation of TCP/IP from Oracle Net. It is written entirely in Java and is therefore platform-independent.

The thin driver does not require Oracle software on the client side. It does need a TCP/IP listener on the server side. Use this driver in Java applets that are downloaded into a Web browser, or in applications where you do not want to install Oracle client software. The thin driver is self-contained, but it opens a Java socket, and thus can only run in a browser that supports sockets.

JDBC OCI Driver

The OCI driver is a Type 2 JDBC driver. It makes calls to the OCI (Oracle Call Interface) which is written in C, to interact with Oracle Database, thus using native and Java methods.

The OCI driver allows access to more features than the thin driver, such as Transparent Application Fail-Over, advanced security, and advanced LOB manipulation.

The OCI driver provides the highest compatibility between the different Oracle Database versions, from 7 to 9*i*. It also supports all installed Oracle Net adapters, including IPC, named pipes, TCP/IP, and IPX/SPX.

Because it uses native methods (a combination of Java and C) the OCI driver is platform-specific. It requires a client Oracle8*i* or later installation including Oracle Net (formerly known as Net8), OCI libraries, CORE libraries, and all other dependent files. The OCI driver usually executes faster than the thin driver.

The OCI driver is not appropriate for Java applets, because it uses a C library that is platform-specific and cannot be downloaded into a Web browser. It is usable in J2EE components running in middle-tier application servers, such as Oracle Application Server. Oracle Application Server provides middleware services and tools that support access between applications and browsers.

JDBC Server-Side Internal Driver

The JDBC server-side internal driver is a Type 2 driver that runs inside the database server, reducing the number of round-trips needed to access large amounts of data. The driver, the Java server VM, the database, the Java native compiler which speeds execution by as much as 10 times, and the SQL engine all run within the same address space.

This driver provides server-side support for any Java program used in the database: SQLJ stored procedures, functions, and triggers, and Java stored procedures. You can also call PL/SQL stored procedures, functions, and triggers.

The server driver fully supports the same features and extensions as the client-side drivers.

Oracle Database Extensions to JDBC Standards

Oracle Database includes the following extensions to the JDBC 1.22 standard:

- Support for Oracle datatypes
- Performance enhancement by row prefetching
- Performance enhancement by execution batching
- Specification of query column types to save round-trips
- Control of `DatabaseMetaData` calls

Oracle Database supports all APIs from the JDBC 2.0 standard, including the core APIs, optional packages, and numerous extensions. Some of the highlights include datasources, JTA and distributed transactions.

Oracle Database supports these features from the JDBC 3.0 standard:

- Support for JDK 1.4.
- Toggling between local and global transactions.
- Transaction savepoints.
- Reuse of prepared statements by connection pools.

Sample JDBC 2.0 Program

The following example shows the recommended technique for looking up a data source using JNDI in JDBC 2.0:

```
// import the JDBC packages
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.pool.*;
...
   InitialContext ictx = new InitialContext();
   DataSource ds = (DataSource)ictx.lookup("jdbc/OracleDS");
   Connection conn = ds.getConnection();
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT ename FROM emp");
   while ( rs.next() ) {
   out.println( rs.getString("ename") + "<br>");
   }
conn.close();
```

Sample Pre-2.0 JDBC Program

The following source code registers an Oracle JDBC thin driver, connects to the database, creates a `Statement` object, executes a query, and processes the result set.

The `SELECT` statement retrieves and lists the contents of the `ENAME` column of the `EMP` table.

```
import java.sql.*
import java.math.*
import java.io.*
import java.awt.*

class JdbcTest {
  public static void main (String args []) throws SQLException {
    // Load Oracle driver
    DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

    // Connect to the local database
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:thin:@myhost:1521:orcl",
                                   "scott", "tiger");

    // Query the employee names
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("SELECT ENAME FROM EMP");
```

```
    // Print the name out
    while (rset.next ())
      System.out.println (rset.getString (1));
    // Close the result set, statement, and the connection
    rset.close();
    stmt.close();
    conn.close();
  }
}
```

One Oracle Database extension to the JDBC drivers is a form of the `getConnection()` method that uses a `Properties` object. The `Properties` object lets you specify user, password, and database information as well as row prefetching and execution batching.

To use the OCI driver in this code, replace the `Connection` statement with:

```
Connection conn = DriverManager.getConnection ("jdbc:oracle:oci8:@MyHostString",
    "scott", "tiger");
```

where `MyHostString` is an entry in the `TNSNAMES.ORA` file.

If you are creating an applet, the `getConnection()` and `registerDriver()` strings will be different.

JDBC in SQLJ Applications

JDBC code and SQLJ code (see "Overview of Oracle SQLJ" on page 1-13) interoperate, allowing dynamic SQL statements in JDBC to be used with both static and dynamic SQL statements in SQLJ. A SQLJ iterator class corresponds to the JDBC result set.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for more information on JDBC

Overview of Oracle SQLJ

SQLJ is an ANSI SQL-1999 standard for embedding SQL statements in Java source code. SQLJ provides a simpler alternative to JDBC for both client-side and server-side SQL data access from Java.

A SQLJ source file contains Java source with embedded SQL statements. Oracle SQLJ supports dynamic as well as static SQL. Support for dynamic SQL is an Oracle extension to the SQLJ standard.

Note: The term "SQLJ," when used in this manual, refers to the Oracle SQLJ implementation, including Oracle SQLJ extensions.

Oracle Database provides a translator and a run time driver to support SQLJ. The SQLJ translator is 100% pure Java and is portable to any JVM that is compliant with JDK version 1.1 or higher.

The Oracle SQLJ translator performs the following tasks:

- Translates SQLJ source to Java code with calls to the SQLJ run time driver. The SQLJ translator converts the source code to pure Java source code, and can check the syntax and semantics of static SQL statements against a database schema and verify the type compatibility of host variables with SQL types.
- Compiles the generated Java code using the Java compiler.
- (Optional) Creates profiles for the target database. SQLJ generates "profile" files with customization specific to Oracle Database.

Oracle Database supports SQLJ stored procedures, functions, and triggers which execute in the Oracle JVM. SQLJ is integrated with JDeveloper. Source-level debugging support for SQLJ is available in JDeveloper.

Here is an example of a simple SQLJ executable statement, which returns one value because `empno` is unique in the `emp` table:

```
String name;
#sql  { SELECT first_name INTO :name FROM employees WHERE employee_id=112 };
System.out.println("Name is " + name + ", employee number = " + employee_id);
```

Each host variable (or qualified name or complex Java host expression) included in a SQL expression is preceded by a colon (`:`). Other SQLJ statements are declarative (they declare Java types). For example, you can declare an iterator (a construct related to a database cursor) for queries that retrieve many values, as follows:

```
#sql iterator EmpIter (String EmpNam, int EmpNumb);
```

See Also: For more examples and details on Oracle SQLJ syntax:

- *Oracle Database JPublisher User's Guide*
- Sample SQLJ code available on the Oracle Technology Network Web site: `http://otn.oracle.com/`

Benefits of SQLJ

Oracle SQLJ extensions to Java allow rapid development and easy maintenance of applications that perform database operations through embedded SQL.

In particular, Oracle SQLJ:

- Provides a concise, legible mechanism for database access from static SQL. Most SQL in applications is static. SQLJ provides more concise and less error-prone static SQL constructs than JDBC does.
- Checks static SQL at translate time.
- Provides an SQL Checker module for verification of syntax and semantics at translate-time.
- Provides flexible deployment configurations. This makes it possible to implement SQLJ on the client or database side or in the middle tier.
- Supports a software standard. SQLJ is an effort of a group of vendors and will be supported by all of them. Applications can access multiple database vendors.
- Provides source code portability. Executables can be used with all of the vendors' DBMSs if the code does not rely on any vendor-specific features.
- Enforces a uniform programming style for the clients and the servers.
- Integrates the SQLJ translator with **Oracle JDeveloper**, a graphical IDE that provides SQLJ translation, Java compilation, profile customizing, and debugging at the source code level, all in one step.
- Includes Oracle type extensions. Datatypes supported include: LOB datatypes, `ROWID`, `REF CURSOR`, `VARRAY`, nested table, user-defined object types, `RAW`, and `NUMBER`.

Comparing SQLJ with JDBC

JDBC provides a complete dynamic SQL interface from Java to databases. It gives developers full control over database operations. SQLJ simplifies Java database programming to improve development productivity.

JDBC provides fine-grained control of the execution of dynamic SQL from Java, while SQLJ provides a higher-level binding to SQL operations in a specific database schema. Here are some differences:

- SQLJ source code is more concise than equivalent JDBC source code.
- SQLJ uses database connections to type-check static SQL code. JDBC, being a completely dynamic API, does not.

- SQLJ provides strong typing of query outputs and return parameters and allows type-checking on calls. JDBC passes values to and from SQL without compile-time type checking.
- SQLJ programs allow direct embedding of Java bind expressions within SQL statements. JDBC requires a separate get or set statement for each bind variable and specifies the binding by position number.
- SQLJ provides simplified rules for calling SQL stored procedures and functions. For example, the following JDBC excerpt requires a generic call to a stored procedure or function, in this case *fun*, to have the following syntax. (This examples shows SQL92 and Oracle JDBC syntaxes. Both are allowed.)

```
prepStmt.prepareCall("{call fun(?,?)}");       //stored procedure SQL92
prepStmt.prepareCall("{? = call fun(?,?)}");   //stored function SQL92
prepStmt.prepareCall("begin fun(:1,:2);end;"); //stored procedure Oracle
prepStmt.prepareCall("begin :1 := fun(:2,:3);end;");//stored func Oracle
```

  Here is the SQLJ equivalent:

```
#sql {call fun(param_list) };  //Stored procedure
// Declare x
...
#sql x = {VALUES(fun(param_list)) };  // Stored function
// where VALUES is the SQL construct
```

The following benefits are common to SQLJ and JDBC:

- SQLJ source files can contain JDBC calls. SQLJ and JDBC are interoperable.
- Oracle JPublisher generates custom Java classes to be used in your SQLJ or JDBC application for mappings to Oracle object types and collections.
- Java and PL/SQL stored procedures can be used interchangeably.

SQLJ Stored Procedures in the Server

SQLJ applications can be stored and executed in the server. To do so, you can use the following techniques:

- Translate, compile, and customize the SQLJ source code on a client and load the generated classes and resources into the server with the `loadjava` utility. The classes are typically stored in a Java archive (`.jar`) file.
- Load the SQLJ source code into the server, also using `loadjava`, where it is translated and compiled by the server's embedded translator.

See Also: *Oracle Database JPublisher User's Guide* for more information on using stored procedures with Oracle SQLJ

Overview of Oracle JPublisher

Oracle JPublisher is a code generator that automates the process of creating database-centric Java classes by hand. Oracle JPublisher is a client-side utility and is built into the database system. You can run Oracle JPublisher from the command-line or directly from the Oracle JDeveloper IDE.

Oracle JPublisher inspects PL/SQL packages and database object types such as SQL object types, `VARRAY` types, and nested table types; then generates a Java class that is a wrapper around the PL/SQL package with corresponding fields and methods.

The generated Java class can be incorporated and used by Java clients or J2EE components to exchange and transfer object type instances to and from the database transparently.

See Also: *Oracle Database JPublisher User's Guide*

Overview of Java Stored Procedures

Java stored procedures allow you to implement programs that run in the database server, independent from programs that run in the middle tier. Structuring your applications in this way reduces complexity and increases reuse, security, performance, and scalability.

For example, you can create a Java stored procedure that performs operations that require data persistence and a separate program to perform presentation or business logic operations.

Java stored procedures interface with SQL using a similar execution model as PL/SQL.

Overview of Database Web Services

Web services represent a distributed computing paradigm for Java application development that is an alternative to earlier Java protocols such as JDBC. It allows application-to-application interaction using XML and Web protocols. The key technologies used in Web services are:

- Web Services Description Language (WSDL)—A standard format for creating an XML document that specifies the operations and parameters, including parameter types, provided by a Web service. In addition, a WSDL document

describes the location, the transport protocol, and the invocation style for the Web service.

- Simple Object Access Protocol (SOAP) messaging—An XML-based message protocol used by Web services. SOAP does not prescribe a specific transport mechanism, such as HTTP, FTP, SMTP, or JMS; however, most Web services accept messages using HTTP or HTTPS.
- Universal Description, Discovery, and Integration (UDDI) business registry—A directory that businesses use to list Web services on the internet. The UDDI registry is often compared to a telephone directory, listing unique identifiers (white pages), business categories (yellow pages), and how to bind to a service protocol (green pages).

Web services can use a variety of techniques and protocols. For example, dispatching can happen in a synchronous (typical) or asynchronous manner, invocation can be performed in RPC-style (a single operation with arguments is sent and a response returned) or in message style (a one-way SOAP document exchange), and different encoding rules can be used (literal or encoded). When calling a Web service, you may know everything about it beforehand (static invocation), or you can discover its operations and transport endpoints on the fly (dynamic invocation).

Database as a Web Service Provider

The database can function as either a Web service provider or as a Web service consumer. When used as a Web services provider, the database enables sharing and disconnected access to stored procedures, data, metadata, and other database resources such as the queuing and messaging systems.

As a Web service provider, the database provides a disconnected and heterogeneous environment that:

- Exposes PL/SQL as well as Java stored procedures
- Exposes SQL Queries and DML statements
- Exposes XML operations using existing and emerging XML APIs such as XSU, SQL/X, and XQuery.
- Exposes Advanced Queuing queues and operations
- Enables deferred invocation of database operations

Database as a Web Service Consumer

In some situations, the database functions as a Web service consumer.

For example, when business data is available dynamically from an external Web service, your database application might need to:

- Fire a trigger on a data value received from a Web service.
- Monitor and query dynamic data over time, such as stock prices, mortgage interest rates, currency exchange rates, or atmospheric data.

Overview of Writing Procedures and Functions in Java

You write these named blocks and then define them using the `loadjava` command or SQL `CREATE FUNCTION`, `CREATE PROCEDURE`, or `CREATE PACKAGE` statements. These Java methods can accept arguments and are callable from:

- SQL `CALL` statements.
- Embedded SQL `CALL` statements.
- PL/SQL blocks, subprograms and packages.
- DML statements (`INSERT`, `UPDATE`, `DELETE`, and `SELECT`).
- Oracle development tools such as OCI, Pro*C/C++ and Oracle Developer.
- Oracle Java interfaces such as JDBC, SQLJ statements, CORBA, and Enterprise Java Beans.
- Method calls from object types.

Overview of Writing Database Triggers in Java

A database trigger is a stored procedure that Oracle Database invokes ("fires") automatically when certain events occur, for example, when a DML operation modifies a certain table. Triggers enforce business rules, prevent incorrect values from being stored, and reduce the need to perform checking and cleanup operations in each application.

Why Use Java for Stored Procedures and Triggers?

- Stored procedures and triggers are compiled once, are easy to use and maintain, and require less memory and computing overhead.
- Network bottlenecks are avoided, and response time is improved. Distributed applications are easier to build and use.
- Computation-bound procedures run faster in the server.

- Data access can be controlled by letting users have only stored procedures and triggers that execute with their definer's privileges instead of invoker's rights.
- PL/SQL and Java stored procedures can call each other.
- Java in the server follows the Java language specification and can use the SQLJ standard, so that databases other than Oracle Database are also supported.
- Stored procedures and triggers can be reused in different applications as well as different geographic sites.

Overview of Pro*C/C++

The Pro*C/C++ precompiler is a software tool that allows the programmer to embed SQL statements in a C or C++ source file. Pro*C/C++ reads the source file as input and outputs a C or C++ source file that replaces the embedded SQL statements with Oracle runtime library calls, and is then compiled by the C or C++ compiler.

When there are errors found during the precompilation or the subsequent compilation, modify your precompiler input file and re-run the two steps.

How You Implement a Pro*C/C++ Application

Here is a simple code fragment from a C source file that queries the table `EMP` which is in the schema `SCOTT`:

```
...
#define  UNAME_LEN   10
...
int   emp_number;
/* Define a host structure for the output values of a SELECT statement. */
/* No declare section needed if precompiler option MODE=ORACLE          */
struct {
    VARCHAR  emp_name[UNAME_LEN];
    float    salary;
    float    commission;
} emprec;
/* Define an indicator structure to correspond to the host output structure. */
struct {
    short emp_name_ind;
    short sal_ind;
    short comm_ind;
} emprec_ind;
...
```

```
/* Select columns ename, sal, and comm given the user's input for empno. */
    EXEC SQL SELECT ename, sal, comm
        INTO :emprec INDICATOR :emprec_ind
        FROM emp
        WHERE empno = :emp_number;
...
```

The embedded SELECT statement is only slightly different from an interactive (SQL*Plus) SELECT statement. Every embedded SQL statement begins with EXEC SQL. The colon (:), precedes every host (C) variable. The returned values of data and indicators (set when the data value is NULL or character columns have been truncated) can be stored in structs (such as in the preceding code fragment), in arrays, or in arrays of structs. Multiple result set values are handled very simply in a manner that resembles the case shown, where there is only one result, because of the unique employee number. You use the actual names of columns and tables in embedded SQL.

Use the default precompiler option values, or you can enter values which give you control over the use of resources, how errors are reported, the formatting of output, and how cursors (which correspond to a particular connection or SQL statement) are managed. Cursors are used when there are multiple result set values.

Enter the options either in a configuration file, on the command line, or in-line inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can then compile, link, and execute the output source file, like any other C program that you write.

Use the precompiler to create server database access from clients that can be on many different platforms. Pro*C/C++ allows you the freedom to design your own user interfaces and to add database access to existing applications.

Before writing your embedded SQL statements, you may want to test interactive versions of the SQL in SQL*Plus. You then make only minor changes to start testing your embedded SQL application.

Highlights of Pro*C/C++ Features

The following is a short subset of the capabilities of Pro*C/C++. For complete details, see the *Pro*C/C++ Precompiler Programmer's Guide*.

- You can write your application in either C or C++.
- You can write multithreaded programs if your platform supports a threads package. Concurrent connections are supported in either single-threaded or multithreaded applications.

- You can improve performance by embedding PL/SQL blocks. These blocks can call functions or procedures in Java or PL/SQL that are written by you or provided in Oracle Database packages.
- Using precompiler options, you can check the syntax and semantics of your SQL or PL/SQL statements during precompilation, as well as at runtime.
- You can call stored PL/SQL and Java subprograms. Modules written in COBOL or in C can be called from Pro*C/C++. External C procedures in shared libraries are callable by your program.
- You can conditionally precompile sections of your code so that they can execute in different environments.
- You can use arrays, or structures, or arrays of structures as host and indicator variables in your code to improve performance.
- You can deal with errors and warnings so that data integrity is guaranteed. As a programmer, you control how errors are handled.
- Your program can convert between internal datatypes and C language datatypes.
- The Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI), lower-level C and C++ interfaces, are available for use in your precompiler source.
- Pro*C/C++ supports dynamic SQL, a technique that allows users to input variable values and statement syntax.
- Pro*C/C++ can use special SQL statements to manipulate tables containing user-defined object types. An Object Type Translator (OTT) will map the object types and named collection types in your database to structures and headers that you will then include in your source.
- Two kinds of collection types, nested tables and `VARRAY`, are supported with a set of SQL statements that allow a high degree of control over data.
- Large Objects (LOBs: `CLOB`, `NCLOB`, and `BFILE` datatypes) are accessed by another set of SQL statements.
- A new ANSI SQL standard for dynamic SQL is supported for new applications, so that you can execute SQL statements with a varying number of host variables. An older technique for dynamic SQL is still usable by pre-existing applications.
- Globalization support lets you use multibyte characters and UCS2 Unicode data.

- Using scrollable cursors, you can move backward and forward through a result set. For example, you can fetch the last row of the result set, or jump forward or backward to an absolute or relative position within the result set.
- A connection pool is a group of physical connections to a database that can be shared by several named connections. Enabling the connection pool option can help to optimize the performance of Pro*C/C++ application. The connection pool option is not enabled by default.

Overview of Pro*COBOL

The Pro*COBOL precompiler is a software tool that allows the programmer to embed SQL statements in a COBOL source code file. Pro*COBOL reads the source file as input and outputs a COBOL source file that replaces the embedded SQL statements with Oracle Database runtime library calls, and is then compiled by the COBOL compiler.

When there are errors found during the precompilation or the subsequent compilation, modify your precompiler input file and re-run the two steps.

How You Implement a Pro*COBOL Application

Here is a simple code fragment from a source file that queries the table `EMP` which is in the schema `SCOTT`:

```
...
 WORKING-STORAGE SECTION.
*
* DEFINE HOST INPUT AND OUTPUT HOST AND INDICATOR VARIABLES.
* NO DECLARE SECTION NEEDED IF MODE=ORACLE.
*
 01  EMP-REC-VARS.
     05  EMP-NAME    PIC X(10) VARYING.
     05  EMP-NUMBER  PIC S9(4) COMP VALUE ZERO.
     05  SALARY      PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMMISSION  PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMM-IND    PIC S9(4) COMP VALUE ZERO.
...
 PROCEDURE DIVISION.
...
     EXEC SQL
         SELECT ENAME, SAL, COMM
         INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
         FROM EMP
```

```
            WHERE EMPNO = :EMP_NUMBE
        END-EXEC.
...
```

The embedded `SELECT` statement is only slightly different from an interactive (SQL*Plus) `SELECT` statement. Every embedded SQL statement begins with EXEC SQL. The colon (:) precedes every host (COBOL) variable. The SQL statement is terminated by END-EXEC. The returned values of data and indicators (set when the data value is `NULL` or character columns have been truncated) can be stored in group items (such as in the preceding code fragment), in tables, or in tables of group items. Multiple result set values are handled very simply in a manner that resembles the case shown, where there is only one result, given the unique employee number. You use the actual names of columns and tables in embedded SQL.

Use the default precompiler option values, or enter values that give you control over the use of resources, how errors are reported, the formatting of output, and how cursors are managed (cursors correspond to a particular connection or SQL statement).

Enter the options in a configuration file, on the command line, or in-line inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can then compile, link, and execute the output source file, like any other COBOL program that you write.

Use the precompiler to create server database access from clients that can be on many different platforms. Pro*COBOL allows you the freedom to design your own user interfaces and to add database access to existing COBOL applications.

The embedded SQL statements available conform to an ANSI standard, so that you can access data from many databases in a program, including remote servers networked through Oracle Net.

Before writing your embedded SQL statements, you may want to test interactive versions of the SQL in SQL*Plus. You then make only minor changes to start testing your embedded SQL application.

Highlights of Pro*COBOL Features

The following is a short subset of the capabilities of Pro*COBOL.

- You can call stored PL/SQL or Java subprograms. You can improve performance by embedding PL/SQL blocks. These blocks can call PL/SQL functions or procedures written by you or provided in Oracle Database packages.

- Precompiler options allow you to define how cursors, errors, syntax-checking, file formats, and so on, are handled.
- Using precompiler options, you can check the syntax and semantics of your SQL or PL/SQL statements during precompilation, as well as at runtime.
- You can conditionally precompile sections of your code so that they can execute in different environments.
- Use tables, or group items, or tables of group items as host and indicator variables in your code to improve performance.
- You can program how errors and warnings are handled, so that data integrity is guaranteed.
- Pro*COBOL supports dynamic SQL, a technique that allows users to input variable values and statement syntax.

See Also: *Pro*COBOL Programmer's Guide* for complete details

Overview of OCI and OCCI

The Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI) are application programming interfaces (APIs) that allow you to create applications that use native procedures or function calls of a third-generation language to access Oracle Database and control all phases of SQL statement execution. These APIs provide:

- Improved performance and scalability through the efficient use of system memory and network connectivity
- Consistent interfaces for dynamic session and transaction management in a two-tier client/server or multitier environment
- N-tiered authentication
- Comprehensive support for application development using Oracle objects
- Access to external databases
- Ability to develop applications that service an increasing number of users and requests without additional hardware investments

OCI lets you manipulate data and schemas in a database using a host programming language, such as C. OCCI is an object-oriented interface suitable for use with C++. These APIs provide a library of standard database access and retrieval functions in the form of a dynamic runtime library (OCILIB) that can be linked in an application

at runtime. This eliminates the need to embed SQL or PL/SQL within 3GL programs.

See Also: For more information about OCI and OCCI calls:

- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Oracle Streams Advanced Queuing User's Guide and Reference*
- *Oracle Database Globalization Support Guide*
- *Oracle Data Cartridge Developer's Guide*

Advantages of OCI

OCI provides significant advantages over other methods of accessing Oracle Database:

- More fine-grained control over all aspects of the application design.
- High degree of control over program execution.
- Use of familiar 3GL programming techniques and application development tools such as browsers and debuggers.
- Support of dynamic SQL, method 4.
- Availability on the broadest range of platforms of all the Oracle programmatic interfaces.
- Dynamic bind and define using callbacks.
- Describe functionality to expose layers of server metadata.
- Asynchronous event notification for registered client applications.
- Enhanced array data manipulation language (DML) capability for array `INSERT`s, `UPDATE`s, and `DELETE`s.
- Ability to associate a commit request with an execute to reduce round-trips.
- Optimization for queries using transparent prefetch buffers to reduce round-trips.
- Thread safety, so you do not have to implement mutual exclusion (mutex) locks on OCI handles.
- The server connection in nonblocking mode means that control returns to the OCI code when a call is still executing or could not complete.

Parts of the OCI

The OCI encompasses four main sets of functionality:

- OCI *relational functions*, for managing database access and processing SQL statements
- OCI *navigational functions*, for manipulating objects retrieved from an Oracle Database
- OCI *datatype mapping and manipulation functions*, for manipulating data attributes of Oracle types
- OCI *external procedure functions*, for writing C callbacks from PL/SQL

Procedural and Non-Procedural Elements

The Oracle Call Interface (OCI) lets you develop applications that combine the non-procedural data access power of Structured Query Language (SQL) with the procedural capabilities of most programming languages, including C and C++.

- In a non-procedural language program, the set of data to be operated on is specified, but what operations will be performed and how the operations are to be carried out is not specified. The non-procedural nature of SQL makes it an easy language to learn and to use to perform database transactions. It is also the standard language used to access and manipulate data in modern relational and object-relational database systems.
- In a procedural language program, the execution of most statements depends on previous or subsequent statements and on control structures, such as loops or conditional branches, which are not available in SQL. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.

The combination of both non-procedural and procedural language elements in an OCI program provides easy access to Oracle Database in a structured programming environment.

The OCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through Oracle Database. For example, an OCI program can run a query against Oracle Database. The queries can require the program to supply data to the database using input (bind) variables, as follows:

```
SELECT name FROM employees WHERE empno = :empnumber
```

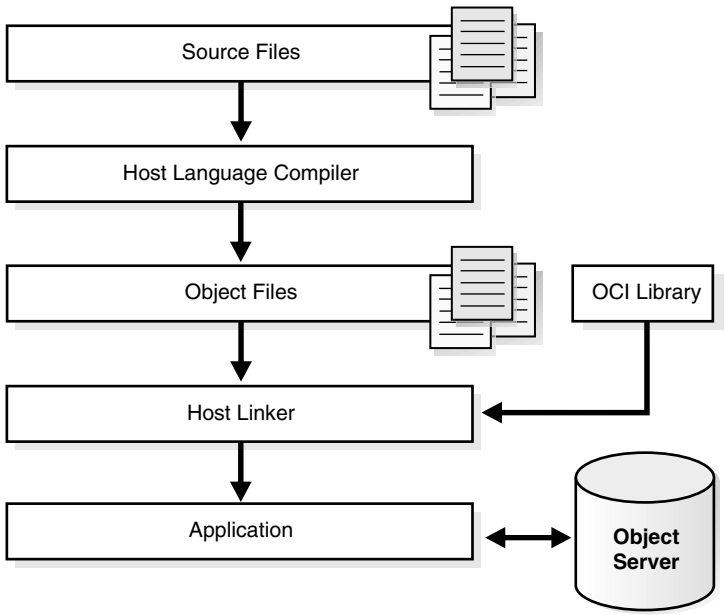
In the preceding SQL statement, `:empnumber` is a placeholder for a value that will be supplied by the application.

You can alternatively use PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. The OCI also provides facilities for accessing and manipulating objects in Oracle Database.

Building an OCI Application

As Figure 1–1 shows, you compile and link an OCI program in the same way that you compile and link a non-database application. There is no need for a separate preprocessing or precompilation step.

Figure 1–1 The OCI Development Process



Note: To properly link your OCI programs, it may be necessary on some platforms to include other libraries, in addition to the OCI library. Check your Oracle platform-specific documentation for further information about extra libraries that may be required.

Overview of Oracle Data Provider for .NET (ODP.NET)

Oracle Data Provider for .NET (ODP.NET) is an implementation of a data provider for Oracle Database.

ODP.NET uses APIs native to Oracle Database to offer fast and reliable access from any .NET application to database features and data. It also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library.

For programmers using Oracle Provider for OLE DB, ADO (ActiveX Data Objects) provides an automation layer that exposes an easy programming model. ADO.NET provides a similar programming model, but without the automation layer, for better performance. More importantly, the ADO.NET model allows native providers such as ODP.NET to expose specific features and datatypes specific to Oracle Database.

See Also: *Oracle Data Provider for .NET Developer's Guide*

Using ODP.NET in a Simple Application

The following is a simple C# application that connects to Oracle Database and displays its version number before disconnecting.

```
using System;
using Oracle.DataAccess.Client;

class Example
{
  OracleConnection con;

  void Connect()
  {
    con = new OracleConnection();
    con.ConnectionString = "User Id=scott;Password=tiger;Data Source=oracle";
    con.Open();
    Console.WriteLine("Connected to Oracle" + con.ServerVersion);
  }

  void Close()
  {
    con.Close();
    con.Dispose();
  }

  static void Main()
  {
```

```
    Example example = new Example();
    example.Connect();
    example.Close();
  }
}
```

Note: Additional samples are provided in directory *ORACLE_BASE**ORACLE_HOME*\ODP.NET\Samples.

Overview of Oracle Objects for OLE (OO4O)

Oracle Objects for OLE (OO4O) is a product designed to allow easy access to data stored in Oracle Database with any programming or scripting language that supports the Microsoft COM Automation and ActiveX technology. This includes Visual Basic, Visual C++, Visual Basic For Applications (VBA), IIS Active Server Pages (VBScript and JavaScript), and others.

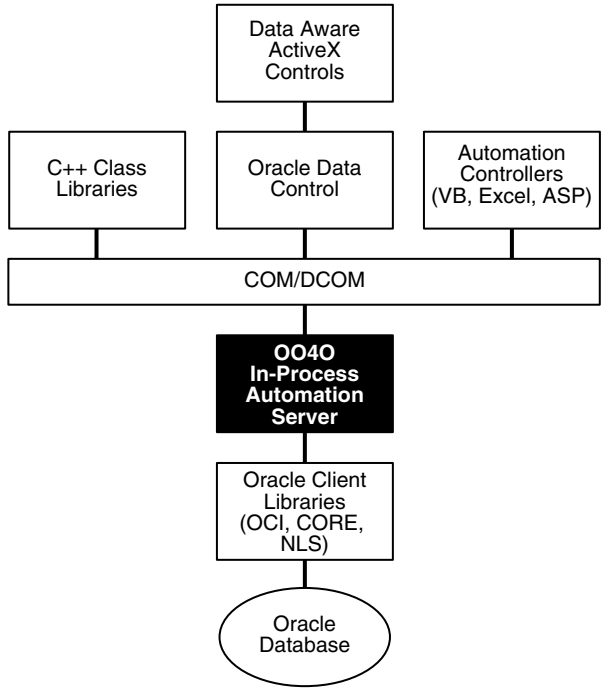
See the OO4O online help for detailed information about using OO4O.

Oracle Objects for OLE consists of the following software layers:

- OO4O "In-Process" Automation Server
- Oracle Data Control
- Oracle Objects for OLE C++ Class Library

Figure 1–2, "Software Layers" illustrates the OO4O software components.

Figure 1–2 Software Layers



OO4O Automation Server

The OO4O Automation Server is a set of COM Automation objects for connecting to Oracle Database, executing SQL statements and PL/SQL blocks, and accessing the results.

Unlike other COM-based database connectivity APIs, such as Microsoft ADO, the OO4O Automation Server has been developed and evolved specifically for use with Oracle Database.

It provides an optimized API for accessing features that are unique to Oracle Database and are otherwise cumbersome or inefficient to use from ODBC or OLE database-specific components.

OO4O provides key features for accessing Oracle Database efficiently and easily in environments ranging from the typical two-tier client/server applications, such as those developed in Visual Basic or Excel, to application servers deployed in

multitiered application server environments such as Web server applications in Microsoft Internet Information Server (IIS) or Microsoft Transaction Server (MTS).

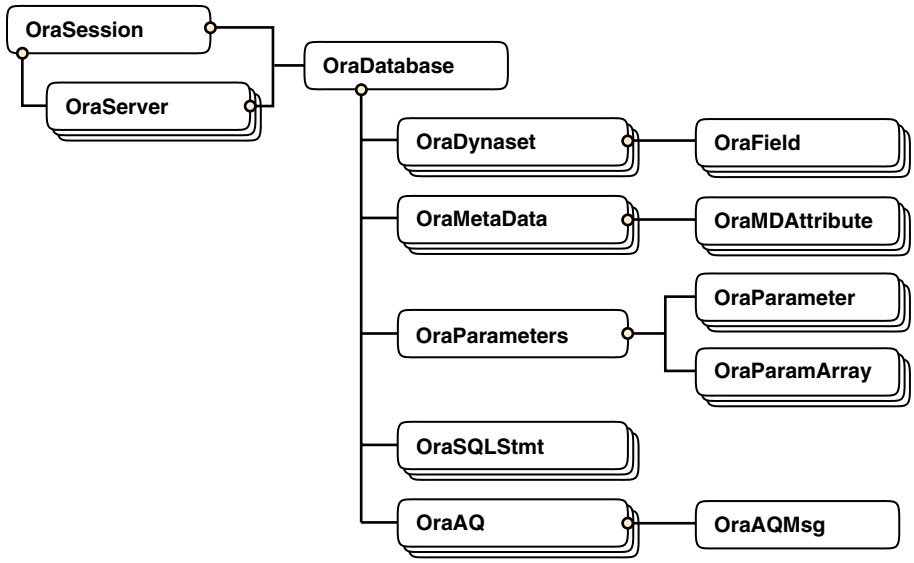
Features include:

- Support for execution of PL/SQL and Java stored procedures, and PL/SQL anonymous blocks. This includes support for Oracle datatypes used as parameters to stored procedures, including PL/SQL cursors. See "Support for Oracle LOB and Object Datatypes" on page 1-37.
- Support for scrollable and updatable cursors for easy and efficient access to result sets of queries.
- Thread-safe objects and Connection Pool Management Facility for developing efficient Web server applications.
- Full support for Oracle object-relational and LOB datatypes.
- Full support for Advanced Queuing.
- Support for array inserts and updates.
- Support for Microsoft Transaction Server (MTS).

OO4O Object Model

The Oracle Objects for OLE object model is illustrated in Figure 1–3, "Objects and Their Relations".

Figure 1–3 Objects and Their Relations



OraSession

An OraSession object manages collections of OraDatabase, OraConnection, and OraDynaset objects used within an application.

Typically, a single OraSession object is created for each application, but you can create named OraSession objects for shared use within and between applications.

The OraSession object is the top-most object for an application. It is the only object created by the CreateObject VB/VBA API and not by an Oracle Objects for OLE method. The following code fragment shows how to create an OraSession object:

```
Dim OraSession as Object
Set OraSession = CreateObject("OracleInProcServer.XOraSession")
```

OraServer

OraServer represents a physical network connection to Oracle Database.

The `OraServer` interface is introduced to expose the connection-multiplexing feature provided in the Oracle Call Interface. After an `OraServer` object is created, multiple user sessions (`OraDatabase`) can be attached to it by invoking the `OpenDatabase` method. This feature is particularly useful for application

components, such as Internet Information Server (IIS), that use Oracle Objects for OLE in n-tier distributed environments.

The use of connection multiplexing when accessing Oracle Database with a large number of user sessions active can help reduce server processing and resource requirements while improving server scalability.

OraServer is used to share a single connection across multiple OraDatabase objects (multiplexing), whereas each OraDatabase obtained from an OraSession has its own physical connection.

OraDatabase

An `OraDatabase` interface adds additional methods for controlling transactions and creating interfaces representing of Oracle object types. Attributes of schema objects can be retrieved using the `Describe` method of the `OraDatabase` interface.

In releases prior to Oracle8*i*, an `OraDatabase` object is created by invoking the `OpenDatabase` method of an `OraSession` interface. The Oracle Net alias, user name, and password are passed as arguments to this method. In Oracle8*i* and later, invocation of this method results in implicit creation of an `OraServer` object.

An `OraDatabase` object can also be created using the `OpenDatabase` method of the `OraServer` interface.

Transaction control methods are available at the `OraDatabase` (user session) level. Transactions may be started as `Read-Write` (default), `Serializable`, or `Read-only`. Transaction control methods include:

- `BeginTrans`
- `CommitTrans`
- `RollbackTrans`

For example:

```
UserSession.BeginTrans(OO4O_TXN_READ_WRITE)
UserSession.ExecuteSQL("delete emp where empno = 1234")
UserSession.CommitTrans
```

OraDynaset

An `OraDynaset` object permits browsing and updating of data created from a SQL `SELECT` statement.

The OraDynaset object can be thought of as a cursor, although in actuality several real cursors may be used to implement the semantics of OraDynaset. An OraDynaset object automatically maintains a local cache of data fetched from the server and transparently implements scrollable cursors within the browse data. Large queries may require significant local disk space; application developers are encouraged to refine queries to limit disk usage.

OraField

An OraField object represents a single column or data item within a row of a dynaset.

If the current row is being updated, then the OraField object represents the currently updated value, although the value may not yet have been committed to the database.

Assignment to the Value property of a field is permitted only if a record is being edited (using Edit) or a new record is being added (using AddNew). Other attempts to assign data to a field's Value property results in an error.

OraMetaData and OraMDAttribute

An OraMetaData object is a collection of OraMDAttribute objects that represent the description information about a particular schema object in the database.

The OraMetaData object can be visualized as a table with three columns:

- Metadata Attribute Name
- Metadata Attribute Value
- Flag specifying whether the Value is another OraMetaData object

The OraMDAttribute objects contained in the OraMetaData object can be accessed by subscripting using ordinal integers or by using the name of the property. Referencing a subscript that is not in the collection results in the return of a NULL OraMDAttribute object.

OraParameters and OraParameter

An OraParameter object represents a bind variable in a SQL statement or PL/SQL block.

OraParameter objects are created, accessed, and removed indirectly through the OraParameters collection of an OraDatabase object. Each parameter has an identifying name and an associated value. You can automatically bind a parameter

to SQL and PL/SQL statements of other objects (as noted in the object descriptions), by using the parameter name as a placeholder in the SQL or PL/SQL statement. Such use of parameters can simplify dynamic queries and increase program performance.

OraParamArray

An `OraParamArray` object represents an array-type bind variable in a SQL statement or PL/SQL block, as opposed to a scalar-type bind variable represented by the `OraParameter` object.

`OraParamArray` objects are created, accessed, and removed indirectly through the `OraParameters` collection of an `OraDatabase` object. Each `OraParamArray` object has an identifying name and an associated value.

OraSQLStmt

An `OraSQLStmt` object represents a single SQL statement. Use the `CreateSQL` method to create an `OraSQLStmt` object from an `OraDatabase` object.

During create and refresh, `OraSQLStmt` objects automatically bind all relevant, enabled input parameters to the specified SQL statement, using the parameter names as placeholders in the SQL statement. This can improve the performance of SQL statement execution without re-parsing the SQL statement.

The `OraSQLStmt` object can be used later to execute the same query using a different value for the `:SALARY` placeholder. This is done as follows (`updateStmt` is the `OraSQLStmt` object here):

```
OraDatabase.Parameters("SALARY").value = 200000
updateStmt.Parameters("ENAME").value = "KING"
updateStmt.Refresh
```

OraAQ

An `OraAQ` object is instantiated by invoking the `CreateAQ` method of the `OraDatabase` interface. It represents a queue that is present in the database.

Oracle Objects for OLE provides interfaces for accessing Oracle Advanced Queuing (AQ) feature. It makes AQ accessible from popular COM-based development environments such as Visual Basic. For a detailed description of Oracle Advanced Queuing, refer to *Oracle Streams Advanced Queuing User's Guide and Reference*.

OraAQMsg

The `OraAQMsg` object encapsulates the message to be enqueued or dequeued. The message can be of any user-defined or raw type.

For a detailed description of Oracle Advanced Queuing, refer to *Oracle Streams Advanced Queuing User's Guide and Reference.*

OraAQAgent

The `OraAQAgent` object represents a message recipient and is only valid for queues that allow multiple consumers. It is a child of `OraAQMsg`.

An `OraAQAgent` object can be instantiated by invoking the `AQAgent` method. For example:

```
Set agent = qMsg.AQAgent(name)
```

An `OraAQAgent` object can also be instantiated by invoking the `AddRecipient` method. For example:

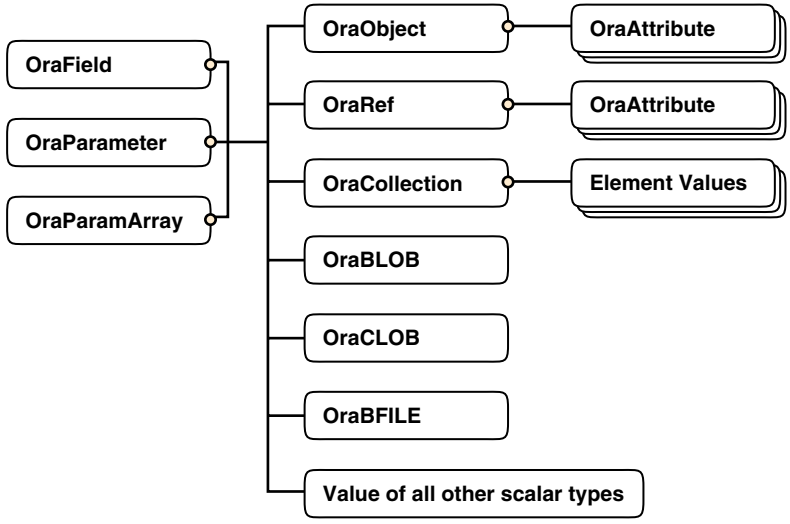
```
Set agent = qMsg.AddRecipient(name, address, protocol).
```

Support for Oracle LOB and Object Datatypes

Oracle Objects for OLE provides full support for accessing and manipulating instances of object datatypes and LOBs in Oracle Database. Figure 1–4, "Supported Oracle Datatypes" illustrates the datatypes supported by OO4O.

Instances of these types can be fetched from the database or passed as input or output variables to SQL statements and PL/SQL blocks, including stored procedures and functions. All instances are mapped to COM Automation Interfaces that provide methods for dynamic attribute access and manipulation.

Figure 1–4 Supported Oracle Datatypes



OraBLOB and OraCLOB

The `OraBlob` and `OraClob` interfaces in Oracle Objects for OLE provide methods for performing operations on large database objects of datatype `BLOB`, `CLOB`, and `NCLOB`. `BLOB`, `CLOB`, and `NCLOB` datatypes are also referred to here as **LOB** datatypes.

LOB data is accessed using `Read` and the `CopyToFile` methods.

LOB data is modified using `Write`, `Append`, `Erase`, `Trim`, `Copy`, `CopyFromFile`, and `CopyFromBFile` methods. Before modifying the content of a LOB column in a row, a row lock must be obtained. If the LOB column is a field of an `OraDynaset`, object, then the lock is obtained by invoking the `Edit` method.

OraBFILE

The `OraBFile` interface in Oracle Objects for OLE provides methods for performing operations on large database objects of datatype `BFILE`.

`BFILE` objects are large binary data objects stored in operating system files outside of the database tablespaces.

Oracle Data Control

Oracle Data Control (ODC) is an ActiveX Control that is designed to simplify the exchange of data between Oracle Database and visual controls such edit, text, list, and grid controls in Visual Basic and other development tools that support custom controls.

ODC acts as an agent to handle the flow of information from Oracle Database and a visual data-aware control, such as a grid control, that is bound to it. The data control manages various user interface (UI) tasks such as displaying and editing data. It also executes and manages the results of database queries.

Oracle Data Control is compatible with the Microsoft data control included with Visual Basic. If you are familiar with the Visual Basic data control, learning to use Oracle Data Control is quick and easy. Communication between data-aware controls and a Data Control is governed by a protocol that has been specified by Microsoft.

Oracle Objects for OLE C++ Class Library

Oracle Objects for OLE C++ Class Library is a collection of C++ classes that provide programmatic access to the Oracle Object Server. Although the class library is implemented using OLE Automation, neither the OLE development kit nor any OLE development knowledge is necessary to use it. This library helps C++ developers avoid the chore of writing COM client code for accessing the OO4O interfaces.

Additional Sources of Information

For detailed information about Oracle Objects for OLE refer to the online help provided with the OO4O product:

- Oracle Objects for OLE Help
- Oracle Objects for OLE C++ Class Library Help

To view examples of how to use Oracle Objects for OLE, see the samples located in the `ORACLE_HOME\OO4O` directory of the Oracle Database installation. Additional OO4O examples can be found in the following Oracle publications:

- *Oracle Database Application Developer's Guide - Large Objects*
- *Oracle Streams Advanced Queuing User's Guide and Reference*
- *PL/SQL Packages and Types Reference*

Choosing a Programming Environment

To choose a programming environment for a new development project:

- Review the preceding overviews and the manuals for each environment.
- Read the platform-specific manual that explains which compilers are approved for use with your platforms.
- If a particular language does not provide a feature you need, remember that PL/SQL and Java stored procedures can both be called from code written in any of the languages in this chapter. Stored procedures include triggers and object type methods.
- External procedures written in C can be called from OCI, Java, PL/SQL or SQL. The external procedure itself can call back into the database using either SQL, OCI, or Pro*C (but not C++).

The following examples illustrate easy choices:

- Pro*COBOL does not support object types or collection types, while Pro*C/C++ does.
- SQLJ does not support dynamic SQL the way that JDBC does.

Choosing Whether to Use OCI or a Precompiler

Precompiler applications typically contain less code than equivalent OCI applications, which can help productivity.

Some situations require detailed control of the database and are suited for OCI applications (either pure OCI or a precompiler application with embedded OCI calls):

- OCI provides more detailed control over multiplexing and migrating sessions.
- OCI provides dynamic bind and define using callbacks that can be used for any arbitrary structure, including lists.
- OCI has many calls to handle metadata.
- OCI allows asynchronous event notifications to be received by a client application. It provides a means for clients to generate notifications for propagation to other clients.
- OCI allows DML statements to use arrays to complete as many iterations as possible before returning any error messages.

- OCI calls for special purposes include Advanced Queuing, globalization support, Data Cartridges, and support of the date and time datatypes.
- OCI calls can be embedded in a Pro*C/C++ application.

Using Built-In Packages and Libraries

Both Java and PL/SQL have built-in packages and libraries.

PL/SQL and Java interoperate in the server. You can execute a PL/SQL package from Java or wrap a PL/SQL class with a Java wrapper so that it can be called from distributed CORBA and EJB clients. The following table shows PL/SQL packages and their Java equivalents:

Table 1–1 PL/SQL and Java Equivalent Software

PL/SQL Package	Java Equivalent
DBMS_ALERT	Call package with SQLJ or JDBC.
DBMS_DDL	JDBC has this functionality.
DBMS_JOB	Schedule a job that has a Java Stored procedure.
DBMS_LOCK	Call with SQLJ or JDBC.
DBMS_MAIL	Use JavaMail.
DBMS_OUTPUT	Use subclass `oracle.aurora.rdbms.OracleDBMSOutputStream` or Java stored procedure `DBMS_JAVA.SET_STREAMS`.
DBMS_PIPE	Call with SQLJ or JDBC.
DBMS_SESSION	Use JDBC to execute an `ALTER SESSION` statement.
DBMS_SNAPSHOT	Call with SQLJ or JDBC.
DBMS_SQL	Use JDBC.
DBMS_TRANSACTION	Use JDBC to execute an `ALTER SESSION` statement.
DBMS_UTILITY	Call with SQLJ or JDBC.
UTL_FILE	Grant the `JAVAUSERPRIV` privilege and then use Java I/O entry points.

Java Compared to PL/SQL

Both Java and PL/SQL can be used to build applications in the database. Here are some guidelines for their use:

PL/SQL Is Optimized for Database Access

PL/SQL uses the same datatypes as SQL. SQL datatypes are thus easier to use and SQL operations are faster than with Java, especially when a large amount of data is involved, when mostly database access is done, or when bulk operations are used.

PL/SQL Is Integrated with the Database

PL/SQL is an extension to SQL offering data encapsulation, information hiding, overloading, and exception-handling.

Some advanced PL/SQL capabilities are not available for Java in Oracle9*i*. Examples are autonomous transactions and the dblink facility for remote databases. Code development is usually faster in PL/SQL than in Java.

Both Java and PL/SQL Have Object-Oriented Features

Java has inheritance, polymorphism, and component models for developing distributed systems. PL/SQL has inheritance and **type evolution**, the ability to change methods and attributes of a type while preserving subtypes and table data that use the type.

Java Is Used for Open Distributed Applications

Java has a richer type system than PL/SQL and is an object-oriented language. Java can use CORBA (which can have many different computer languages in its clients) and EJB. PL/SQL packages can be called from CORBA or EJB clients.

You can run XML tools, the Internet File System, or JavaMail from Java.

Many Java-based development tools are available throughout the industry.

Part II

Designing the Database

This part contains the following chapters:

2

Selecting a Datatype

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

See Also:

- *Oracle Database Application Developer's Guide - Object-Relational Features* for information about more complex types, such as object types, varrays, and nested tables
- *Oracle Database Application Developer's Guide - Large Objects* for information about LOB datatypes
- *PL/SQL User's Guide and Reference* for information about the PL/SQL datatypes. Many SQL datatypes are the same or similar in PL/SQL.

Summary of 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 Database to treat values of one datatype differently from values of another datatype. For example, Oracle Database 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` (synonymous with `VARCHAR2`)
  - `NVARCHAR2`
  - `CLOB`
  - `NCLOB`
  - `LONG`
- Numerical datatypes:
  - `BINARY_FLOAT`
  - `BINARY_DOUBLE`
  - `NUMBER`
- Time and date datatypes:
  - `DATE`
  - `INTERVAL DAY TO SECOND`

  - `INTERVAL YEAR TO MONTH`
  - `TIMESTAMP`
  - `TIMESTAMP WITH TIME ZONE`
  - `TIMESTAMP WITH LOCAL TIME ZONE`
- Binary datatypes
  - `BLOB`
  - `BFILE`
  - `RAW`
  - `LONG RAW`
- Row address datatypes
  - `ROWID`
  - `UROWID`

See Also:

- *Oracle Database SQL Reference* for general descriptions of these datatypes
- *Oracle Database Application Developer's Guide - Large Objects* for information about the LOB datatypes

Table 2–1 summarizes the information about each Oracle built-in datatype.

Table 2–1 Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length / Default Values
`CHAR [(size [BYTE | CHAR])]`	Fixed-length character data of length *size* bytes or characters.	Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. When neither `BYTE` nor `CHAR` is specified, the setting of `NLS_LENGTH_SEMANTICS` at the time of column creation determines which is used. Consider the character set (single-byte or multibyte) before setting *size*.
`VARCHAR2 (size [BYTE | CHAR])`	Variable-length character data, with maximum length *size* bytes or characters. `BYTE` or `CHAR` indicates that the column has byte or character semantics, respectively. A *size* must be specified.	Variable for each row, up to 4000 bytes per row. When neither `BYTE` nor `CHAR` is specified, the setting of `NLS_LENGTH_SEMANTICS` at the time of column creation determines which is used. Consider the character set (single-byte or multibyte) before setting *size*.
`NCHAR [(size)]`	Fixed-length Unicode character data of length *size* characters. The number of *bytes* is twice this number for the `AL16UTF16` encoding and 3 times this number for the `UTF8` encoding.)	Fixed for every row in the table (with trailing blanks). The upper limit is 2000 bytes per row. Default *size* is 1 character.
`NVARCHAR2 (size)`	Variable-length Unicode character data of maximum length *size* characters. The number of bytes may be up to 2 times *size* for a the `AL16UTF16` encoding and 3 times this number for the `UTF8` encoding. A *size* must be specified.	Variable for each row. The upper limit is 4000 bytes per row.

Table 2–1 (Cont.) Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length / Default Values
CLOB	Single-byte or multibyte character data. Both fixed-width and variable-width character sets are supported, and both use the CHAR character set.	Up to 2^{32} - 1 bytes * (database block size), or 4 gigabytes * block size.[1] See *Oracle Database Application Developer's Guide - Large Objects*.
NCLOB	Unicode national character set (NCHAR) data. Both fixed-width and variable-width character sets are supported, and both use the NCHAR character set.	Up to 2^{32} - 1 bytes * (database block size), or 4 gigabytes * block size. [1] See *Oracle Database Application Developer's Guide - Large Objects*.
LONG	Variable-length character data. Provided for backward compatibility.	Variable for each row in the table, up to 2^{31} - 1 bytes, or 2 gigabytes, per row.
BINARY_FLOAT	32-bit floating-point number.	4 bytes.
BINARY_DOUBLE	64-bit floating-point number.	8 bytes.
NUMBER [(*prec* | *prec*, *scale*)]	Variable-length numeric data. Precision *prec* is the the total number of digits; scale *scale* is the number of digits to the right of the decimal point. Precision can range from 1 to 38. Scale can range from -84 to 127. With precision specified, this is a floating-point number; with no precision specified, it is a fixed-point number.	Variable for each row. The maximum space available for a given column is 21 bytes per row.

Table 2–1 (Cont.) Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length / Default Values
`DATE`	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 9999 C.E.	Fixed at 7 bytes for each row in the table. Default format is a string (such as `DD-MON-RR`) specified by the `NLS_DATE_FORMAT` parameter.
`INTERVAL YEAR [(`*`yr_prec`*`)] TO MONTH`	A period of time, represented as years and months. The *`yr_prec`* is the number of digits in the `YEAR` field of the date. The precision can be from 0 to 9, and defaults to 2 digits.	Fixed at 5 bytes.
`INTERVAL DAY [(`*`day_prec`*`)] TO SECOND [(`*`frac_sec_prec`*`)]`	A period of time, represented as days, hours, minutes, and seconds. The *`day_prec`* and *`frac_sec_prec`* are the number of digits in the `DAY` and the fractional `SECOND` fields of the date, respectively. These precision values can each be from 0 to 9, and they default to 2 digits for *`day_prec`* and 6 digits for *`frac_sec_prec`*.	Fixed at 11 bytes.
`TIMESTAMP [(`*`frac_sec_prec`*`)]`	A value representing a date and time, including fractional seconds. (The exact resolution depends on the operating system clock.) The *`frac_sec_prec`* specifies the number of digits in the fractional second part of the `SECOND` date field. The *`frac_sec_prec`* can be from 0 to 9, and defaults to 6 digits.	Varies from 7 to 11 bytes, depending on the precision. The default is determined by the `NLS_TIMESTAMP_FORMAT` initialization parameter.

Table 2–1 (Cont.) Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length / Default Values
`TIMESTAMP [(frac_sec_prec)] WITH TIME ZONE`	A value representing a date and time, plus an associated time zone setting. The time zone can be an offset from UTC, such as `'-5:0'`, or a region name, such as `'US/Pacific'`. The *frac_sec_prec* is as for datatype `TIMESTAMP`.	Fixed at 13 bytes. The default is determined by the `NLS_TIMESTAMP_TZ_FORMAT` initialization parameter.
`TIMESTAMP [(frac_sec_prec)] WITH LOCAL TIME ZONE`	Similar to `TIMESTAMP WITH TIME ZONE`, except that the data is normalized to the database time zone when stored, and adjusted to match the client's time zone when retrieved. The *frac_sec_prec* is as for datatype `TIMESTAMP`.	Varies from 7 to 11 bytes, depending on *frac_sec_prec*. The default is determined by the `NLS_TIMESTAMP_FORMAT` initialization parameter.
`BLOB`	Unstructured binary data.	Up to 2^{32} - 1 bytes * (database block size), or 4 gigabytes * block size.[1] See *Oracle Database Application Developer's Guide - Large Objects*.
`BFILE`	Address of a binary file stored outside the database. Enables byte-stream I/O access to external LOBs residing on the database server.	The referenced file can be up to 2^{32} - 1 bytes * (database block size), or 4 gigabytes * block size.[1] See *Oracle Database Application Developer's Guide - Large Objects*.
`RAW (size)`	Variable-length raw binary data. A *size*, which is the maximum number of bytes, must be specified. Provided for backward compatibility.	Variable for each row in the table, up to 2000 bytes per row.

Table 2–1 (Cont.) Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length / Default Values
`LONG RAW`	Variable-length raw binary data. Provided for backward compatibility.	Variable for each row in the table, up to 2^{31} - 1 bytes, or 2 gigabytes, per row.
`ROWID`	Base 64 binary data representing a row address. Used primarily for values returned by the `ROWID` pseudocolumn.	Fixed at 10 bytes (extended `ROWID`) or 6 bytes (restricted `ROWID`) for each row in the table.
`UROWID [(size)]`	Base 64 binary data representing the logical address of a row in an index-organized table. The optional *`size`* is the number of bytes in a column of type `UROWID`.	Maximum size and default are both 4000 bytes.

[1] Prior to *Oracle Database 10g*, the limit was 4 gigabytes, not 4 gigabytes*blocksize.

Representing Character Data

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.)
- `NCHAR` and `NVARCHAR2` datatypes store Unicode character data only.
- `CLOB` and `NCLOB` datatypes store single-byte and multibyte character strings of up to four gigabytes.

  See Also: *Oracle Database Application Developer's Guide - Large Objects*

- The `LONG` datatype stores variable-length character strings containing up to two gigabytes, but with many restrictions. This datatype is provided only for backward compatibility with existing applications. In general, new applications should use `CLOB` and `NCLOB` datatypes to store large amounts of character data, and `BLOB` and `BFILE` to store large amounts of binary data.

See Also:

- *Oracle Database Application Developer's Guide - Large Objects* for information on LOB datatypes (including CLOB and NCLOB datatypes) and information on migrating from LONG to LOB datatypes
- *Oracle Database SQL Reference* for details on restrictions on LONG datatypes

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 add any extra blanks.
- *Comparison semantics* – Use the CHAR datatype when you require ANSI compatibility in comparison semantics (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 by the NLS_LANGUAGE parameter from the database character set to the character set defined for the user session, if these are different.

Column Lengths for Single-Byte and Multibyte Character Sets

The lengths of CHAR and VARCHAR2 columns can be specified as either bytes or characters.

The lengths of NCHAR and NVARCHAR2 columns are always specified in characters, making them ideal for storing Unicode data, where a character might consist of multiple bytes.

```
-- ID contains only single-byte data, up to 32 bytes.
ID VARCHAR2(32 BYTE);
-- NAME contains data in the database character set. The 32 characters might
-- be stored as more than 32 bytes, if the database character set allows
-- multibyte characters.
NAME VARCHAR2(32 CHAR);
```

```
-- BIOGRAPHY can represent 2000 characters in any Unicode-representable
-- language.
-- The exact encoding depends on the national character set, but the column can
-- contain multibyte values even if the database character set is single-byte.
BIOGRAPHY NVARCHAR2(2000);
-- The representation of COMMENT, as 2000 bytes or 2000 characters, depends
-- on the initialization parameter NLS_LENGTH_SEMANTICS.
COMMENT VARCHAR2(2000);
```

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, then the number of bytes and the number of characters in a column is the same. If it is multibyte, then 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. To avoid overflowing buffers, specify data as `NCHAR` or `NVARCHAR2` if it might use a Unicode encoding that is different from the database character set.

See Also:

- *Oracle Database Globalization Support Guide*
- *Oracle Database SQL Reference*

Implicit Conversion Between CHAR/VARCHAR2 and NCHAR/NVARCHAR2

In database releases prior to Oracle9*i*, the `NCHAR` and `NVARCHAR2` types were difficult to use because they could not be interchanged with `CHAR` and `VARCHAR2`. For example, an `NVARCHAR2` literal required special notation, such as `N'string_value'`. In releases after Oracle8*i*, you can specify `NCHAR` and `NVARCHAR2` without the `N` notation, and you can mix them with `CHAR` and `VARCHAR2` values in SQL statements and functions.

Comparison Semantics

Oracle Database compares `CHAR` and `NCHAR` values using **blank-padded comparison semantics**. If two values have different character lengths, then Oracle Database adds space characters at the end of the shorter value, until the two values are the same length. Oracle Database 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 thus considered equal.

Oracle Database 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 Database 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. If one value is a prefix of the other, then it is considered less ("abc" < "abcxyz")

Because Oracle Database 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, then 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 Database 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 Database to ignore trailing blanks when comparing character values, then you must store these values in CHAR columns.

> **See Also:** *Oracle Database SQL Reference* for more information on comparison semantics for these datatypes

Representing Numeric Data with Number and Floating-Point Datatypes

The following SQL datatypes allow you to store numeric data:

- BINARY_FLOAT
- BINARY_DOUBLE
- NUMBER

The BINARY_FLOAT and BINARY_DOUBLE datatypes store floating-point data in the 32-bit IEEE 754 format and the double precision 64-bit IEEE 754 format respectively. Compared to the Oracle NUMBER datatype, arithmetic operations on floating-point data are usually faster for BINARY_FLOAT and BINARY_DOUBLE. Also, high-precision values require less space when stored as BINARY_FLOAT and BINARY_DOUBLE.

In client interfaces supported by Oracle Database, arithmetic operations on BINARY_FLOAT and BINARY_DOUBLE datatypes are performed by the native instruction set supplied by the hardware vendor. The term **native floating-point datatypes** is used here to refer to datatypes including BINARY_FLOAT and

BINARY_DOUBLE, and to all implementations of these types in supported client interfaces.

Floating-Point Number System Concepts

The floating-point number system is a common way of representing and manipulating numeric values in computer systems. A floating-point number is characterized by these components: a binary-valued *sign*, a signed *exponent*, a *significand*, and a *base*. Its value is the signed product of its significand and the base raised to the power of its exponent:

$$(-1)^{sign} \cdot significand \cdot base^{\,exponent}$$

For example, the number 4.31 can be represented as $(-1)^0 \cdot 431 \cdot 10^{-2}$, with sign 0, significand 431, base 10, and exponent -1.

A floating-point number format specifies how the components of a floating-point number are represented. The choice of representation determines the range and precision of the values the format can represent. By definition, the range is the interval bounded by the smallest and the largest values the format can represent and the precision is the number of digits in the significand.

Formats for floating-point values support neither infinite precision nor infinite range. There are a finite number of bits to represent a number and only a finite number of values that a format can represent. A floating-point number that uses more precision than available with a given format is rounded.

A floating-point number can be represented in a binary system (one that uses base 2), as in the IEEE 754 standard, or in a decimal system (one that uses base 10), such as Oracle NUMBER. The base affects many properties of the format, including how a numeric value is rounded.

For a decimal floating-point number format like Oracle NUMBER, rounding is done to the nearest decimal place (for example. 1000, 10, or 0.01). The IEEE 754 formats use a binary format for floating-point values and round numbers to the nearest binary place (for example: 1024, 512, or 1/64).

The native floating-point datatypes supported by the database round to the nearest binary place, so they are not satisfactory for applications that require decimal rounding. Use the Oracle NUMBER datatype for applications where decimal rounding is required on floating-point data.

About Floating-Point Formats

The value of a floating-point number that uses a binary format is determined by:

$(-1)^s\ 2^E\ (b_0\ b_1\ b_2\ ...\ b_{p-1})$

where

s = 0 or 1

E = any integer between E_{min} and E_{max}, inclusive (see Table 2–2)

b_i = 0 or 1; the sequence of bits represents a number in base 2

The leading bit of the significand, b_0, must be set (1), except for subnormal numbers (explained later). Consequently, the leading bit is not actually stored. Consequently, the formats provide N bits of precision, although only N-1 bits are stored.

> **Note:** The IEEE 754 specification also defines extended single-precision and extended double-precision formats, which are not supported by Oracle Database.

The parameters for these formats are listed in Table 2–2, and the storage parameters for the formats are listed in Table 2–3. The in-memory formats for single-precision and double-precision datatypes are specified by IEEE 754.

Table 2–2 Summary of Format Parameters

Parameter	Single-precision (32-bit)	Double-precision (64-bit)
p	24	53
E_{min}	-126	-1022
E_{max}	+127	+1023

Table 2–3 Summary of Storage Parameters

Datatype	Sign bits	Exponent bits	Significand bits	Total bits
single-precision	1	8	24 (23 stored)	32
double-precision	1	11	53 (52 stored)	64

A significand is **normalized** when the leading bit of the significand is set. IEEE 754 defines **denormal** or **subnormal** values as numbers that are too small to be represented with an implied leading set bit in the significand. The number is too

small because its exponent would be too large if its significand were normalized to have an implied leading bit set. IEEE 754 formats support subnormal values. Subnormal values preserve the following property:

if: x - y == 0.0 (using floating-point subtraction)

then: x == y

Table 2–4 shows the range and precision of the required formats in the IEEE 754 standard and those of Oracle `NUMBER`. Range limits are expressed here in terms of positive numbers; they also apply to the absolute value of a negative number. (The notation "*number* e *exponent*" used here stands for *number* multiplied by 10 raised to the *exponent* power: *number* · $10^{exponent}$.)

Table 2–4 Range and Precision of IEEE 754 formats

Range and Precision	Single-precision 32-bit[1]	Double-precision 64-bit[1]	Oracle NUMBER Datatype
Max positive normal number	3.40282347e+38	1.7976931348623157 e+308	< 1.0e126
Min positive normal number	1.17549435e-38	2.2250738585072014 e-308	1.0e-130
Max positive subnormal number	1.17549421e-38	2.2250738585072009 e-308	not applicable
Min positive subnormal number	1.40129846e-45	4.9406564584124654 e-324	not applicable
Precision (decimal digits)	6 - 9	15 - 17	38 - 40

[1] These numbers are quoted from the *IEEE Numerical Computation Guide*.

See Also: *Oracle Database SQL Reference*

Representing Special Values with Native Floating-Point Formats

IEEE 754 allows special values to be represented. These special values are *positive infinity* (+INF), *negative infinity* (-INF), and *not-a-number* (NaN). IEEE 754 also distinguishes between *positive zero* (+0) and *negative zero* (-0). **NaN** is used to represent results of operations that are undefined.

There are many bit patterns in IEEE 754 that represent NaN. Bit patterns can represent NaN with and without the sign bit set. IEEE 754 distinguishes between signalling NaNs and quiet NaNs. IEEE 754 specifies behavior for when exceptions

are enabled and disabled. Oracle Database does not allow exceptions to be enabled; the database behavior is that specified by IEEE 754 for when exceptions are disabled. In particular, no distinction is made between signalling NaNs and quiet NaNs. Programmers using Oracle Call Interface can retrieve NaN values from Oracle Database; whether a retrieved NaN value is signalling or quiet is dependent on the client platform and beyond the control of Oracle Database.

IEEE 754 does not define the bit pattern for either type of NaN. Positive infinity, negative infinity, positive zero, and negative zero are each represented by a specific bit pattern.

Ignoring signs, there are five classes of values: zero, subnormal, normal, infinity and NaN. The first four classes are ordered as:

zero < subnormal < normal < infinity

In IEEE 754, NaN is unordered with other classes of special values and with itself.

Behavior of Special Values for Native Floating-Point Datatypes

When used with the database, special values of native floating-point datatypes behave as follows:

- All NaNs are quiet.
- IEEE 754 exceptions are not raised.
- NaN is ordered:

  all non-NaN < NaN

  any NaN == any other NaN
- -0 is converted to +0.
- All NaNs are converted to the same bit pattern.

> **See Also:** "Comparison Operators for Native Floating-Point Datatypes" on page 2-16 for more information on NaN compared to other values

Rounding of Native Floating-Point Datatypes

IEEE 754 defines four rounding modes. The rounding modes are: *round to nearest* (default), *round to positive infinity*, *round to negative infinity*, and *round to zero*. Oracle Database supports only *round to nearest* mode.

Comparison Operators for Native Floating-Point Datatypes

Comparison operators are defined for *equal to*, *not equal to*, *greater than*, *greater than or equal to*, *less than*, *less than or equal to*, and *unordered*. There are special cases:

- Comparisons ignore the sign of zero (-0 is equal to, not less than, +0).
- In Oracle Database, NaN is equal to itself. NaN is greater than everything except itself. That is, NaN == NaN and NaN > *x*, unless *x* is NaN.

> **See Also:** "Behavior of Special Values for Native Floating-Point Datatypes" on page 2-15 for more information on comparison results, ordering, and other behaviors of special values

Arithmetic Operators for Native Floating-Point Datatypes

Arithmetic operators are defined for multiplication, division, addition, subtraction, remainder, and square root. The mode used to round the result of the operation can be defined. Exceptions can be raised when operations are performed. Exceptions can also be disabled.

Until recently, Java required floating-point arithmetic to be exactly reproducible. IEEE 754 does not require such behavior. IEEE 754 allows for the result of operations, including arithmetic, to be delivered to a destination that uses a range greater than that used by the operands to the operation. The result of a double-precision multiplication can be computed at an extended double-precision destination. When this is done, the result must be rounded as if the destination were single-precision or double-precision. However, the range of the result (the number of bits used for the exponent) can use the range supported by the wider (extended double-precision) destination. This may result in a double-rounding error in which the least significant bit of the result is incorrect.

This can only occur for double-precision multiplication and division on hardware that implements the IA-32 and IA-64 instruction set architecture. Thus, with the exception of this case, arithmetic for these datatypes will be reproducible across platforms. When the result of a computation is NaN, all platforms will produce a value for which `IS NAN` is true. However, all platforms do not have to use the same bit pattern.

Conversion Functions for Native Floating-Point Datatypes

Functions are defined that convert between floating-point and other formats, including string formats that use decimal precision. Precision may be lost during

the conversion. Exceptions can be raised during conversion. The following conversions can be done:

- float to double
- double to float
- float/double to decimal (string)
- decimal (string) to float/double
- float/double to integer valued float/double

Exceptions for Native Floating-Point Datatypes

The IEEE 754 specification defines the following exceptions that can be thrown: *invalid*, *inexact*, *divide by zero*, *underflow*, and *overflow*. Oracle Database does not raise these exceptions for native floating-point datatypes. Generally, situations that would raise an exception produce the following values:

Exception	Value
Underflow	0
Overflow	-INF, +INF
Invalid Operation	NaN
Divide by Zero	-INF, +INF, NaN
Inexact	any value – rounding was performed

Client Interfaces for Native Floating-Point Datatypes

Support for native floating-point datatypes is implemented in the following client interfaces:

- SQL
- PL/SQL
- OCI
- OCCI
- Pro*C/C++
- JDBC

SQL Native Floating-Point Datatypes

The SQL datatypes `BINARY_FLOAT` and `BINARY_DOUBLE` implement native floating-point datatypes in the SQL environment. A number of SQL functions are provided that operate on these datatypes. `BINARY_FLOAT` and `BINARY_DOUBLE` are supported wherever an expression (`expr`) appears in SQL syntax.

> **See Also:** *Oracle Database SQL Reference* for details on SQL functions and the implementation of these datatypes

OCI Native Floating-Point Datatypes SQLT_BFLOAT and SQLT_BDOUBLE

The Oracle Call Interface (OCI) application programming interface (API) implements the IEEE 754 single precision and double precision native floating-point datatypes with the datatypes `SQLT_BFLOAT` and `SQLT_BDOUBLE` respectively.

Conversions between these types and the SQL types `BINARY_FLOAT` and `BINARY_DOUBLE` are exact on platforms that implement the IEEE 754 standard for the C datatypes `float` and `double`.

> **See Also:** *Oracle Call Interface Programmer's Guide*

Native Floating-Point Datatypes Supported in Oracle OBJECT Types

The SQL datatypes `BINARY_FLOAT` and `BINARY_DOUBLE` are supported as attributes of Oracle `OBJECT` types.

Pro*C/C++ Support for Native Floating-Point Datatypes

- Pro*C/C++ supports the native `float` and native `double` datatypes using the column datatypes `BINARY_FLOAT` and `BINARY_DOUBLE`. These datatypes can be used in the same way the Oracle `NUMBER` datatype is used. You can bind the native C/C++ datatypes `float` and `double` to `BINARY_FLOAT` and `BINARY_DOUBLE` types respectively. To do so, set the Pro*C/C++ precompiler command line option `NATIVE_TYPES` to `Y` (yes) when you compile your application.

Storing Data 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 Database platforms, and offer up to 38 decimal digits of precision. You can store positive and negative numbers of magnitude 1×10^{-130} through $9.9\overline{9} \times 10^{125}$, as well as zero, in a `NUMBER` column.

You can specify that a column contains a floating-point number, for example:

```
distance NUMBER
```

Or, you can specify a precision (total number of digits) and scale (number of digits to the right of the decimal point):

```
price NUMBER (8, 2)
```

Although not required, specifying precision and scale helps to identify bad input values. If a precision is not specified, the column stores values as they are provided. Table 2–5 shows examples of how data different scale factors affect storage.

Table 2–5 How Scale Factors Affect Numeric Data Storage

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

See Also: *Oracle Database Concepts* for information about the internal format for the NUMBER datatype

Representing Date and Time Data

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.

Use the TIMESTAMP datatype to store values that are precise to fractional seconds. For example, an application that must decide which of two events occurred first might use TIMESTAMP. An application that needs to specify the time for a job to execute might use DATE.

Because TIMESTAMP WITH TIME ZONE can also store time zone information, it is particularly suited for recording date information that must be gathered or coordinated across geographic regions.

Use TIMESTAMP WITH LOCAL TIME ZONE when the time zone is not significant. For example, you might use it in an application that schedules teleconferences, where participants each see the start and end times for their own time zone.

The TIMESTAMP WITH LOCAL TIME ZONE type is appropriate for two-tier applications where you want to display dates and times using the time zone of the client system. It is generally inappropriate in three-tier applications such as those involving a Web server, because data displayed in a Web browser is formatted according to the time zone of the Web server, not the time zone of the browser. (The Web server is the database client, so its local time is used.)

Use INTERVAL DAY TO SECOND to represent the precise difference between two DATETIME values. For example, you might use this value to set a reminder for a time 36 hours in the future, or to record the time between the start and end of a race. To represent long spans of time, including multiple years, with high precision, you can use a large value for the days portion.

Use INTERVAL YEAR TO MONTH to represent the difference between two DATETIME values, where the only significant portions are the year and the month. For example, you might use this value to set a reminder for a date 18 months in the future, or check whether 6 months have elapsed since a particular date.

Oracle Database 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 Also:** *Oracle Call Interface Programmer's Guide* for a complete description of the Oracle Database internal date format

Date Format

For input and output of dates, the standard Oracle Database default date format is DD-MON-RR. For example:

```
'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. For example:

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

> **See Also:** *Oracle Database Concepts* for information about Julian dates. Oracle Database Julian dates might not be compatible with Julian dates generated by other date algorithms.

Be careful using a date format like DD-MON-YY. The YY indicates the year in the current century. For example, *31-DEC-92* is *December 31, 2092*, not 1992 as you might expect. If you want to indicate years in any century other than the current one, use a different format mask, such as the default RR.

Checking If Two DATE Values Refer to the Same Day

To compare dates that have time data, use the SQL function TRUNC to ignore the time component.

Displaying the Current Date and Time

Use the SQL function SYSDATE to return the system date and time.

Setting SYSDATE to a Constant Value

The FIXED_DATE initialization parameter lets you set SYSDATE to a constant, which can be useful for testing.

Printing a Date with BC/AD Notation

```
SQL> -- By default, the date is printed without any BC or AD qualifier.
SQL> SELECT SYSDATE FROM DUAL;
```

```
SYSDATE
---------
24-JAN-02
SQL> -- Adding BC to the format string prints the date with BC or AD
SQL> -- as appropriate.
SQL> SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY BC') FROM DUAL;

TO_CHAR(SYSDAT
--------------
24-JAN-2002 AD
```

Time Format

Time is stored in 24-hour format, `HH24:MI:SS`. By default, the time in a `DATE` column is 12:00:00 A.M. (midnight) if no time portion is entered, or if the `DATE` is truncated. 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_tab (bname, bday) VALUES
    ('ANNIE',TO_DATE('13-NOV-92 10:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

Note: You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Birthdays_tab (Bname VARCHAR2(20),Bday DATE)
```

Performing Date Arithmetic

Oracle Database provides a number of features to help with date arithmetic, so that you do not need to perform your own calculations on the number of seconds in a day, the number of days in each month, and so on.

Some useful functions include:

- `ADD_MONTHS`
- `SYSDATE`
- `SYSTIMESTAMP`
- `TRUNC`. When applied to a DATE value, it trims off the time portion so that it represents the very beginning of the day (the stroke of midnight). By truncating

two DATE values and comparing them, you can check whether they refer to the same day. You can also use TRUNC along with a GROUP BY clause to produce daily totals.

- Arithmetic operators such as + and -.
- INTERVAL datatype. To represent constants when performing date arithmetic, you can use the INTERVAL datatype rather than performing your own calculations. For example, you might add or subtract INTERVAL constants from DATE values, or subtract two DATE values and compare the result to an INTERVAL.
- Comparison operators such as >, <, =, and BETWEEN.

Converting Between Datetime Types

Some useful functions include:

- EXTRACT
- NUMTODSINTERVAL
- NUMTOYMINTERVAL
- TO_DATE (and its opposite, TO_CHAR)
- TO_DSINTERVAL
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL

> **See Also:** *Oracle Database SQL Reference* for full details about each function

Handling Time Zones

Oracle Database provides a number of functions to help with calculations involving time zones. For example, TO_DATE does not work with values of type TIMESTAMP WITH TIME ZONE; you must use TO_TIMESTAMP_TZ instead.

Some useful functions include:

- CURRENT_DATE
- CURRENT_TIMESTAMP
- DBTIMEZONE

- EXTRACT
- FROM_TZ
- LOCALTIMESTAMP
- SESSIONTIMEZONE
- SYS_EXTRACT_UTC
- SYSTIMESTAMP
- TO_TIMESTAMP_TZ

See Also: *Oracle Database SQL Reference*

Importing and Exporting Datetime Types

TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE values are always stored in normalized format, so that you can export, import, and compare them without worrying about time zone offsets. DATE and TIMESTAMP values do not store an associated time zone, and you must adjust them to account for any time zone differences between source and target databases.

Establishing Year 2000 Compliance

An application must satisfy the following criteria to meet the requirements for Year 2000 (Y2K) compliance:

- Process date information before, during, and after 1st January 2000 without error. This entails accepting date input, providing date output, storing date information and performing calculation on dates or portions of dates.
- Provide services as published in its documentation before, during and after 1st January 2000 without changes in operation resulting from the advent of the new century.
- Respond to two-digit date input in a way that resolves ambiguity as to the century in a clearly defined manner.
- Manage the leap year occurring in the year 2000 according to the quad-centennial rule.

These criteria are a superset of the Year 2000 conformance requirements set out by the British Standards Institute in *DISC PD-2000-1, A Definition of Year 2000 Conformity Requirements*.

You can warrant your application as Y2K compliant only if you have validated its conformance at all three of the following system levels:

- Hardware
- System software, including databases, transaction processors and operating systems
- Application software, from third parties or developed in-house

Oracle Server Year 2000 Compliance

The Oracle Server is Year 2000 compliant. Oracle's Development Organization has conducted tests of various Year 2000 operational scenarios to verify that there is no impact to users with respect to the year 2000. These scenarios included tests of replication, point-in-time recovery, distributed transactions. System management and networking features across time zones / datelines / centuries have also been tested.

Oracle's Year 2000 product compliance does not eliminate the need for you to test your own applications. Most importantly, your application software must be tested on Oracle Database to ensure that operations having to do with the year 2000 perform as promised. This test is critical even if the application software is certified to be Year 2000 compliant, because there are no universal protocol definitions that can guarantee conformance without such testing.

Centuries and the Year 2000

Oracle Database stores year data with the century information. For example, it 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 also have four digit years. Oracle Database utilities such as import, export, and recovery also deal properly with four-digit years.

Applications that use Oracle Database (version 7 or later) and exploit the `DATE` datatype (for dates or dates with time values) need have no concerns about their stored data and the year 2000. Beginning with Oracle Database version 7, the `DATE` datatype stores date and time data to a precision that includes a four digit year and a time component down to seconds (typically `'YYYY:MM:DD:HH24:MI:SS'`)

However, some applications might be written with an assumption about the year (such as assuming that everything is 19*xx*). Such an application might hand over a two-digit year to the database, and the procedures that Oracle Database uses for determining the century could be different from what the programmer expects (see

"Troubleshooting Y2K Problems in Applications" on page 2-29). For this reason, you should review and test your code with regard to years in different centuries.

Examples of The RR Date Format

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*. Therefore, regardless of the current century at the time the data is entered, the RR format will ensure that the year stored in the database is as follows:

- If the current year is in the second half of the century (50 - 99), and a two-digit year between 00 and 49 is entered, this will be stored as a "next century" year. For example, 02 entered in 1996 will be stored as 2002.
- If the current year is in the second half of the century (50 - 99), and a two-digit year between 50 and 99 is entered, this will be stored as a "current century" year. For example, 97 entered in 1996 will be stored as 1997.
- If the current year is in the first half of the century (00 - 49), and a two-digit year between 00 and 49 is entered, this will be stored as a "current century" year. For example, 02 entered in 2001 will be stored as 2002.
- If the current year is in the first half of the century (00 - 49), and a two-digit year between 50 and 99 is entered, this will be stored as a "previous century" year. For example, 97 entered in 2001 will be stored as 1997.

The RR date format is available for inserting and updating DATE data in the database. It is not required for retrieval or query of data already stored in the database as Oracle Database has always stored the YEAR component of a date in its four-digit form.

Here is an example of the RR usage:

```
INSERT INTO employees (employee_id, department_id, hire_date) VALUES
   (9999, 20, TO_DATE('01-jan-03', 'DD-MON-RR'));

 INSERT INTO employees (employee_id, department_id, hire_date) VALUES
    (8888, 20, TO_DATE('01-jan-67',  'DD-MON-RR'));

SELECT employee_id, department_id,
   TO_CHAR(hire_date, 'DD-MON-YYYY') hire_date
FROM employees;
```

Examples of The CC Date Format

The CC date format element of the TO_CHAR function returns the century of a given date. For example:

```
SELECT TO_CHAR(TO_DATE('01-JAN-2000','DD-MON-YYYY'),'CC') CENTURY FROM DUAL;

CENTURY
-------
20

SELECT TO_CHAR(TO_DATE('01-JAN-2001','DD-MON-YYYY'),'CC') CENTURY FROM DUAL;

CENTURY
-------
21
```

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 year 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 Hiredate is a CE date for which you want the true century, use the following expression:

```
SELECT DECODE (TO_CHAR (Hiredate, 'YY'),
    '00', TO_CHAR (Hiredate - 366, 'CC'),
    TO_CHAR (Hiredate, 'CC'))  FROM Emp_tab;
```

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

> **See Also:** *Oracle Database SQL Reference* for more information about date format codes

Storing Dates in Character Datatypes

Where applications store date values in CHAR or VARCHAR2 datatypes, and the century information is not maintained. You will need to modify the application to include routines to ensure that such dates are treated appropriately when affected by the change in century. You can do this by changing the strings to maintain

century information or, with certain constraints, by using the RR date format when interpreting the string as a date.

If you are creating a new application, or if you are modifying an application to ensure that dates stored as character strings are Year 2000 compliant, convert character datatype dates to the DATE datatype. If this is not feasible, store the dates in a form that is language- and format-independent, and that handles full years. For example, use SYYYY/MM/DD plus the time element as HH24:MI:SS if necessary. Note that dates stored in this form must be converted to the correct external format whenever they are received or displayed.

The format SYYYY/MM/DD HH24:MI:SS has the following advantages:

- It is language-independent in that the months are numeric.
- It contains the full four-digit year, so centuries are explicit.
- The time is represented fully. Since the most significant elements occur first, character-based sort operations process the dates correctly.

The S format mask prefixes BC dates with "-".

Viewing Date Settings

The following views let you verify what your date settings are:

- V$NLS_DATABASE_PARAMETERS shows instance-wide Globalization Support parameters, whether or not the values were explicitly declared in the initialization parameter file.
- NLS_SESSION_PARAMETERS shows current session values, which may have been changed by ALTER SESSION.

To see the available values for time zone region and time zone abbreviation, you can query the view V$TIMEZONE_NAMES.

A **format mask** is a character that describes the format of DATE or NUMBER data stored in a character string. You may use the format model as an argument of the TO_CHAR or TO_DATE function for one of the following:

- To specify the format for Oracle Database to use in returning a value.
- To specify the format for a value you have specified for Oracle Database to store.

> **Note:** The format mask does not change the internal representation of the value in the database.

Altering Date Settings

You may set the date format in your environment or the default date format for the entire database. If you set the format in your environment, it will override any initialization settings.

Change the NLS_DATE_FORMAT parameter settings in the following order:

1. Set the Client side, such as the Windows NT registry and Unix environment variables.
2. Set the session using ALTER SESSION SET NLS_DATE_FORMAT. To change the date format for the session, issue the following SQL command:

   ```
   ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-RR'
   ```

3. Set the Server using the NLS_DATE_FORMAT parameter in your initialization file, init.ora. To change the default date format for the entire database, edit file init.ora to include the following

   ```
   NLS_DATE_FORMAT = DD-MON-RR
   ```

The NLS_DATE_FORMAT setting relies on this order. For a client/server application, NLS_DATE_FORMAT must be set on both the server and the client.

Caution: Changing this parameter at the *database* level will change all existing date fields, as described. Make changes at the *session* level, unless all users and all currently running applications process dates in the range 1950-2049.

Troubleshooting Y2K Problems in Applications

In this section we describe some common programming problems around Y2K compliance. These problems may seem to derive from incorrect Year 2000 processing by the database engine, but on closer inspection they are seen to arise from incorrect use of Oracle Database technology.

Y2K Example: Date Columns Too Short

Your application may have defined the year of a date using a column of CHAR(2) or NUMBER(2) in order to save disk space. This can lead to unpredictable results when 20xx dates are mixed with 19xx dates. To resolve this, modify your application to use the full 4-digit year.

Y2K Example: 4-Digit Years Mixed with 2-Digit Years

Your application may be designed to store a 4-digit year, but the code may allow for the incorrect storage of 2-digit year rows with the 4-digit year rows. This will lead to unpredictable results for queries by date if the date columns contains dates earlier than 1900. To deal with this problem, have your application check for rows that contain dates earlier than 1900, and then adjust for this.

Y2K Example: Wide Range of Years Stored as Two Digits

Examine your applications to determine if it processes dates prior to 1950 or later than 2049, and stores the year as only two digits. If both conditions are met, your application should not use the `RR` format, but should instead expand the 2-digit year `YY` into a 4-digit year `YYYY`, and store the 4-digit year in the database.

Y2K Example: Handling Feb. 29, 2000

The following unusual error helps illuminate the interaction between `NLS_DATE_FORMAT` and the Oracle Database `RR` format mask. The following is a syntactically correct statement, but it contains a logical flaw:

```
SELECT TO_CHAR(TO_DATE(LAST_DAY('01-FEB-00'),'DD-MON-RR'),'MM/DD/RRRR')
FROM DUAL;
```

This query returns `02/28/2000`. This is consistent with the defined behavior of the `RR` format mask, but it is incorrect because the year 2000 is a leap year.

The problem is that the operation is using the default `NLS_DATE_FORMAT`, which is `DD-MON-YY`. If the `NLS_DATE_FORMAT` is changed to `DD-MON-RR`, then the same select returns `02/29/2000`, which is the correct value.

Let us evaluate the query asOracle Database does. The first function processed is the innermost function, `LAST_DAY`. Because `NLS_DATE_FORMAT` is `YY`, this correctly returns `2/28`, *because it is using the year 1900* to evaluate the expression. The value `2/28` is then returned to the next outer function. So, the `TO_DATE` and `TO_CHAR` functions format the value `02/28/00` using the `RR` and `RRRR` format masks, and display the result as `02/28/2000`.

If `SELECT LAST_DAY('01-FEB-00') FROM DUAL` is issued, the result changes depending on the `NLS_DATE_FORMAT`. With `YY`, the `LAST_DAY` returned is `28-Feb-00` because the year is interpreted as 1900. With `RR`, the `LAST_DAY` returned is `29-Feb-00` because the year is interpreted as 2000. The year 1900 is not a leap year, but the year 2000 is.

Y2K Example: Implicit Date Conversion within DECODE

If you use the DECODE function with a third argument that is NULL or of datatype datatype CHAR or VARCHAR2, Oracle Database converts the return value to datatype VARCHAR2. Therefore, the following statement inserts date 31.12.1900:

```
INSERT INTO destination_table (date_column)
    SELECT DECODE('31.12.2000', '00000000', NULL,
        TO_DATE('31.12.2000','DD.MM.YYYY'))
    FROM DUAL;
```

This statement inserts date 04.10.1901:

```
INSERT INTO destination_table (date_column)
    SELECT DECODE('01.11.1999', '00000000', NULL, sysdate+1000)
FROM DUAL;
```

In these examples, the third argument in the DECODE argument list is a NULL value, so Oracle Database implicitly converted the DATE value to a VARCHAR2 string using the default format mask. This is DD-MON-YY, which drops the first two digits of the year.

> **Note:** When inserting the record into a table, Oracle Database implicitly converts the string into a date, using the last two digits of the current year. To ensure the correct year is interpreted, set NLS_DATE_FORMAT using RR or YYYY.

Y2K Example: Partitioning Tables Based on DATE Columns

When you create a partitioned table using a DATE datatype column in the partition key, use a 4-digit year to specify date ranges. For example:

```
CREATE TABLE stock_xactions (stock_symbol CHAR(5),
    stock_series CHAR(1),
    num_shares NUMBER(10),
    price NUMBER(5,2),
    trade_date DATE)
    STORAGE (INITIAL 100K NEXT 50K) LOGGING
    PARTITION BY RANGE (trade_date)
      (PARTITION sx1992 VALUES LESS THAN (TO_DATE('01-JAN-1993','DD-MON-YYYY'))
    TABLESPACE ts0
        NOLOGGING,
       PARTITION sx1993 VALUES LESS THAN (TO_DATE('01-JAN-1994','DD-MON-YYYY'))
    TABLESPACE ts1,
      PARTITION sx1994 VALUES LESS THAN (TO_DATE('01-JAN-1995','DD-MON-YYYY'))
```

```
    TABLESPACE ts2);
```

Y2K Example: Views Defined Using 2-Digit Years

Oracle Database views depend on the session state. In particular, a predicate with a 2-digit year is allowed in a view. For example:

```
WHERE col > '12-MAY-99'
```

Interpretation of the full 4-digit year depends on the setting of `NLS_DATE_FORMAT`.

Representing Conditional Expressions as Data

The Oracle Expression Filter feature lets you store conditional expressions as data in the database. The Expression Filter provides a mechanism that you use to place a constraint on a `VARCHAR2` column to ensure that the values stored are valid SQL `WHERE` clause expressions. This mechanism also identifies the set of attributes that can be referenced in the conditional expressions.

For example, suppose each row of a table `Traders` holds data for a stock trading account holder. You can define a column that stores information about stocks each trader is interested in as a conditional expression. To do so, you use the following PL/SQL commands to create an attribute set `Ticker` with a list of required elementary attributes for the trading symbol, limit price, and amount of change in the stock price:

```
CREATE OR REPLACE TYPE Ticker AS OBJECT
   (Symbol VARCHAR2(20), Price NUMBER, Change NUMBER);

BEGIN
  dbms_expfil.create_attribute_set(attr_set  => 'Ticker',
                                   from_type => 'YES');
END;
```

Next, you associate the attribute set with the expression set stored in the database column `TRADER.INTEREST` as follows:

```
BEGIN
  dbms_expfil.assign_attribute_set (attr_set => 'Ticker',
                                    expr_tab => 'Traders',
                                    expr_col => 'Interest');
END;
```

This places a constraint on the INTEREST column that ensures the column stores valid conditional expressions. You can then populate the table with trader names, email addresses and conditional expressions that represents a stock the trader is interested in at a particular price:

```
INSERT INTO Traders (Name, Email, Interest)
  VALUES ('Scott', 'scott@abc.com', 'SYMBOL = ''ABC'' and PRICE > 25');
```

At this point, you can use the EVALUATE operator to identify the conditional expressions that evaluate to TRUE for a given data item. For example, the following query can be issued to return all the traders who are interested in a given stock quote (Symbol='ABC', Price=31, Change=5.2):

```
SELECT Name, Email FROM Traders
  WHERE EVALUATE (Interest,
                  'Symbol=>''ABC'', Price=>31, Change=>5.2') = 1;
```

To speed up a query like this one, you can optionally create an Oracle Expression Filter index on the INTEREST column.

> **See Also:** *Oracle Database Application Developer's Guide - Expression Filter* for details on Oracle Expression Filter

Representing Geographic Coordinate Data

To represent Geographic Information System (GIS) or spatial data in the database, you can use Oracle Spatial features, including the type MDSYS.SDO_GEOMETRY. You can store the data in the database using either an object-relational or a relational model, and manipulate and query the data using a set of PL/SQL packages.

For more information, see *Oracle Spatial User's Guide and Reference*.

Representing Image, Audio, and Video Data

Whether you store such multimedia data inside the database as BLOB or BFILE values, or store it externally on a Web server or other kind of server, you can use interMedia to access the data using either an object-relational or a relational model, and manipulate and query the data using a set of object types.

For more information, see *Oracle interMedia Reference*.

Representing Searchable Text Data

Rather than writing low-level code to do full-text searches, you can use Oracle9*i* Text, formerly known as ConText and interMedia Text. It stores the search data in a special kind of index, and lets you query the data with operators and PL/SQL packages. This makes it simple to create your own search engine using data from tables, files, or URLs, and combine the search logic with relational queries. You can also search XML data this way, using XPath notation.

For more information, see *Oracle Text Application Developer's Guide*.

Representing Large Amounts of Data

The database provides several datatypes for representing large amounts of data. These datatypes are grouped under the general category of Large Objects (LOBs); they are described in Table 2–6:

Table 2–6 Large Object Datatypes

Datatype	Description
`BLOB`	**Binary Large Object** Suitable for representing large amounts of binary data such as images, video, or other multimedia data.
`CLOB`	**Character Large Object** Suitable for representing large amounts of character data. `CLOB` types are stored using the database character set.
`NCLOB`	**National Character Set Large Object** Suitable for representing large amounts of character data in National Character Set format.
`BFILE`	Datatype for storing Large Objects in the operating system's file system, outside of the database files or tablespace. Note that the `BFILE` type is read-only; other LOB types are read/write. `BFILE` objects are also sometimes referred to as **external LOBs**.

An instance of type `BLOB`, `CLOB`, or `NCLOB` can exist as either a persistent LOB instance or a temporary LOB instance. Persistent and temporary instances differ as follows:

- A **temporary LOB** instance is declared in the scope of your application.
- A **persistent LOB** instance is created and stored in the database.

With the exception of declaring, freeing, creating, and committing, operations on persistent and temporary LOB instances are performed the same way.

For more details on using LOBs in applications, see the *Oracle Database Application Developer's Guide - Large Objects*.

Note: In earlier releases, the `LONG`, `RAW`, and `LONG RAW` datatypes were typically used to store large amounts of data. Use of these types is no longer recommended for new development. If your application still uses these types, migrate your application to use LOB types. See the *Oracle Database Application Developer's Guide - Large Objects*

Using RAW and LONG RAW Datatypes

Note: Oracle recommends that you convert `LONG RAW` columns to binary LOB (`BLOB`) columns and convert `LONG` columns to character LOB (`CLOB` or `NCLOB`) columns. LOB columns are subject to far fewer restrictions than `LONG` and `LONG RAW` columns.

See Also:

- See *Oracle Database Application Developer's Guide - Large Objects* for information about the `BLOB` and `BFILE` datatypes
- See the *Oracle Database SQL Reference* for restrictions on `LONG` and `LONG RAW` datatypes

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

Oracle Net and the Export and Import utilities do not perform character conversion when transmitting `RAW` or `LONG RAW` data. When Oracle Database 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.

> **See Also:** *Oracle Database SQL Reference* for restrictions on LONG and LONG RAW datatypes

Addressing Rows Directly with the ROWID Datatype

Every row in an Oracle Database table is assigned a ROWID that corresponds to the physical address of a row. If the row is too large to fit within a single data block, the ROWID identifies the initial row piece. Although ROWIDs are usually unique, different rows can have the same ROWID if they are in the same data block but in different clustered tables.

Each table in Oracle Database has a pseudocolumn named ROWID.

> **See Also:** *Oracle Database Concepts* for general information about the ROWID pseudocolumn and the ROWID datatype

Extended ROWID Format

Oracle Database 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 Database assigns to schema objects, such as nonpartitioned tables or partitions. For example:

```
SELECT DATA_OBJECT_ID FROM ALL_OBJECTS
       WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP_TAB';
```

This query returns the data object identifier for the EMP_TAB table in the SCOTT schema.

> **See Also:** *PL/SQL Packages and Types Reference* for information about using the DBMS_ROWID package functions to get the data object identifier in other ways

Different Forms of the ROWID

Oracle Database documentation uses the term **ROWID** in different ways, depending on context.

ROWID Pseudocolumn Each table and nonjoined view has a pseudocolumn called ROWID. For example:

```
CREATE TABLE T_tab (col1 Rowid);
INSERT INTO T_tab SELECT Rowid FROM Emp_tab WHERE Empno = 7499;
```

This command returns the ROWID pseudocolumn of the row of the EMP_TAB table that satisfies the query, and inserts it into the T1 table.

Internal ROWID The internal ROWID is an internal structure that 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.

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:** *PL/SQL Packages and Types Reference* for information about the DBMS_ROWID package

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 Oracle Database version 7 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:** *Oracle Database Administrator's Guide*

It is possible for a client of Oracle Database version 7 to access a more recent database, and vice versa. A client in this sense could be a remote database accessing a server using database links, or a client 3GL or 4GL application accessing a server.

> **See Also:** *PL/SQL Packages and Types Reference* and *Oracle Database Upgrade Guide* for more information on the ROWID_TO_EXTENDED function

Accessing Oracle Database Version 7 from an Oracle9i Client The ROWID values that are returned are always restricted ROWIDs. Also, Oracle9*i* returns restricted ROWID values to a database server for Oracle Database version 7.

The following ROWID functionality works when accessing a server for Oracle Database version 7:

- 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 Oracle9i Database from a Client of Oracle Database Version 7 Oracle9*i* 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 a client of Oracle Database version 7 to import a table from a later version that has a ROWID column (not the ROWID pseudocolumn), if any row of the table contains an extended ROWID value.

ANSI/ISO, DB2, and SQL/DS Datatypes

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

The ANSI datatype conversions are shown in Table 2–7. The ANSI/ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, s defaults to 0.

Table 2–7 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*)

Table 2–7 (Cont.) ANSI Datatype Conversions to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
INTEGER, INT, SMALLINT	NUMBER (38)
FLOAT (*p*)	FLOAT (*p*)
REAL	FLOAT (63)
DOUBLE PRECISION	FLOAT (126)
CHARACTER VARYING(*n*), CHAR VARYING(*n*)	VARCHAR2 (*n*)
TIMESTAMP	TIMESTAMP
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE

Table 2–8 shows the DB2 and SQL/DS conversions.

Table 2–8 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
TIMESTAMP	TIMESTAMP

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

How Oracle Database Converts Datatypes

In some cases, Oracle Database allows data of one datatype where it expects data of a different datatype. Generally, an expression cannot contain values with different

datatypes. However, Oracle Database can use the following functions to automatically convert data to the expected datatype:

- TO_NUMBER()
- TO_BINARY_FLOAT()
- TO_BINARY_DOUBLE()
- TO_CHAR()
- TO_NCHAR()
- TO_DATE()
- HEXTORAW()
- RAWTOHEX()
- RAWTONHEX()
- ROWIDTOCHAR()
- ROWIDTONCHAR()
- CHARTOROWID()
- TO_CLOB()
- TO_NCLOB()
- TO_BLOB()
- TO_RAW()

Implicit datatype conversions work according to the rules explained in "Datatype Conversion During Assignments".

> **See Also:** *Oracle Database SQL Reference* for details about datatype conversion

Datatype Conversion During Assignments

For assignments, Oracle Database can automatically convert the following:

- VARCHAR2, NVARCHAR2, CHAR, or NCHAR to NUMBER
- NUMBER to BINARY_FLOAT, BINARY_DOUBLE, or NVARCHAR2
- BINARY_FLOAT to NUMBER, BINARY_DOUBLE, CHAR, VARCHAR, or VARCHAR2

- BINARY_DOUBLE to NUMBER, BINARY_FLOAT, CHAR, VARCHAR, or VARCHAR2
- VARCHAR2, NVARCHAR2, CHAR, or NCHAR to DATE
- DATE to VARCHAR2 or NVARCHAR2
- VARCHAR2, NVARCHAR2, CHAR, or NCHAR to ROWID
- ROWID to VARCHAR2 or NVARCHAR2
- VARCHAR2, NVARCHAR2, CHAR, NCHAR, or LONG to CLOB
- VARCHAR2, NVARCHAR2, CHAR, NCHAR, or LONG to NCLOB
- CLOB to CHAR, NCHAR, VARCHAR2, NVARCHAR2, and LONG
- NCLOB to CHAR, NCHAR, VARCHAR2, NVARCHAR2, and LONG
- NVARCHAR2, NCHAR, or BLOB to RAW
- RAW to BLOB
- VARCHAR2 or CHAR to HEX
- HEX to VARCHAR2

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

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

```
CREATE PACKAGE Test_Pack AS var1 CHAR(5); END;
CREATE TABLE Table1_tab (col1 NUMBER);
```

- `variable := expression`

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

  ```
  VAR1 := 0;
  ```

- `INSERT INTO Table1_tab 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 `Table1_tab`. For example, Oracle Database automatically converts the data provided in the following `INSERT` statement for `Table1_tab`:

```
INSERT INTO Table1_tab VALUES ('19');
```

- `UPDATE Table1_tab SET column = expression`

  The datatype of `expression` must be either the same as, or convertible to, the datatype of `column`. For example, Oracle Database automatically converts the data provided in the following `UPDATE` statement issued against `Table1_tab`:

  ```
  UPDATE Table1_tab SET col1 = '30';
  ```

- `SELECT column INTO variable FROM Table1_tab`

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

  ```
  SELECT Col1 INTO Var1 FROM Table1_tab WHERE Col1 = 30;
  ```

Datatype Conversion During Expression Evaluation

For expression evaluation, Oracle Database 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 Database 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:

  ```
  commission + '500'
  ```

- Boolean expressions, such as:

  ```
  bonus > salary / '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 Database 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 Database first evaluates *expression* using the conversion rules for expressions; *expression* can be as simple or complex as desired. If it succeeds, then the evaluation of *expression* results in a single value and datatype. Then, Oracle Database tries to assign this value to the target variable using the conversion rules for assignments.

Representing Dynamically Typed Data

You might be familiar with features in some languages that allow datatypes to change at runtime, or let a program check the type of a variable. For example, C has the `union` keyword and the `void *` pointer, and Java has the `typeof` operator and wrapper types such as `Number`. Oracle9*i* includes features that let you create variables and columns that can hold data of any type, and test such data values to see their underlying representation. Using these features, a single table column can represent a numeric value in one row, a string value in another row, and an object in another row.

You can use the built-in type `SYS.ANYDATA` to represent values of any scalar or object type. This type is an object type with methods to bring in a scalar value of any type, and turn the value back into a scalar or object.

In the same way, you can use the built-in type `SYS.ANYDATASET` to represent values of any collection type.

To manipulate and check type information, you can use the built-in type `SYS.ANYTYPE` in combination with the `DBMS_TYPES` package. For example, the following program represents data of different underlying types in a table, then interprets the underlying type of each row and processes each value appropriately:

```
-- This example defines and executes a PL/SQL procedure that
-- uses methods built into SYS.ANYDATA to access information about
-- data stored in a SYS.ANYDATA table column.

DROP TYPE Employee FORCE;
DROP TABLE mytab;
CREATE OR REPLACE TYPE Employee AS OBJECT ( empno NUMBER,
  ename VARCHAR2(10) );
/
CREATE TABLE mytab ( id NUMBER, data SYS.ANYDATA );

INSERT INTO mytab VALUES (1, SYS.ANYDATA.ConvertNumber(5));
INSERT INTO mytab VALUES (2, SYS.ANYDATA.ConvertObject(Employee(5555, 'john')));
commit;

CREATE OR REPLACE procedure P IS
  CURSOR cur IS SELECT id, data FROM mytab;
  v_id           mytab.id%TYPE;
  v_data         mytab.data%TYPE;
  v_type         SYS.ANYTYPE;
  v_typecode     PLS_INTEGER;
  v_typename     VARCHAR2(60);
  v_dummy        PLS_INTEGER;
  v_n            NUMBER;
  v_employee     Employee;
  non_null_anytype_for_NUMBER exception;
  unknown_typename            exception;
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO v_id, v_data;
    EXIT WHEN cur%NOTFOUND;

/* The typecode is a number that signifies what type is represented by v_data.
   GetType also produces a value of type SYS.AnyType with methods you can call
   to find precision and scale of a number, length of a string, and so on. */
    v_typecode := v_data.GetType ( v_type /* OUT */ );

/* Now we compare the typecode against constants from DBMS_TYPES to see what
   kind of data we have, and decide how to display it. */
    CASE v_typecode

      WHEN Dbms_Types.Typecode_NUMBER THEN
        IF v_type IS NOT NULL
-- This condition should never happen, but we check just in case.
```

```
          THEN RAISE non_null_anytype_for_NUMBER; END IF;
-- For each type, there is a Get method.
        v_dummy := v_data.GetNUMBER ( v_n /* OUT */ );
        Dbms_Output.Put_Line (
          To_Char(v_id) || ': NUMBER = ' || To_Char(v_n) );

      WHEN Dbms_Types.Typecode_Object THEN
        v_typename := v_data.GetTypeName();
-- An object type's name is qualified with the schema name.
        IF v_typename NOT IN ( 'SCOTT.EMPLOYEE' )
-- If we encounter any object type besides EMPLOYEE, raise an exception.
          THEN RAISE unknown_typename; END IF;
        v_dummy := v_data.GetObject ( v_employee /* OUT */  );
        Dbms_Output.Put_Line (
          To_Char(v_id) || ': user-defined type = ' || v_typename ||
            ' ( ' || v_employee.empno || ', ' || v_employee.ename || ' )' );
    END CASE;
  END LOOP;
  CLOSE cur;
EXCEPTION
  WHEN non_null_anytype_for_NUMBER THEN
    RAISE_Application_Error ( -20000,
      'Paradox: the return AnyType instance FROM GetType ' ||
      'should be NULL for all but user-defined types' );
  WHEN unknown_typename THEN
    RAISE_Application_Error ( -20000, 'Unknown user-defined type ' ||
      v_typename || ' - program written to handle only SCOTT.EMPLOYEE' );
END;
/
-- The query and the procedure P in the preceding code sample
-- produce output like the following:

SQL> SELECT t.data.gettypename() FROM mytab t;

T.DATA.GETTYPENAME()
--------------------------------------------------------------------------------
SYS.NUMBER
SCOTT.EMPLOYEE

SQL> EXEC P;
1: NUMBER = 5
2: user-defined type = SCOTT.EMPLOYEE ( 5555, john )
```

You can access the same features through the OCI interface, using the OCIType, OCIAnyData, and OCIAnyDataSet interfaces.

See Also:

PL/SQL Packages and Types Reference for details about the DBMS_TYPES package

Oracle Database Application Developer's Guide - Object-Relational Features for information and examples using the ANYDATA, ANYDATASET, and ANYTYPE types

Oracle Call Interface Programmer's Guide for details about the OCI interfaces

Representing XML Data

If you have information stored as files in XML format, or if you want to take an object type and store it as XML, you can use the XMLType built-in type.

XMLType columns store their data as CLOBs. You can take an existing CLOB, VARCHAR2, or any object type, and call the XMLType constructor to turn it into an XML object.

Once an XML object is inside the database, you can use queries to traverse it (using the XML XPath notation) and extract all or part of its data.

You can also produce XML output from existing relational data, and split XML documents across relational tables and columns. You can use the DBMS_XMLQUERY, DBMS_XMLGEN, and DBMS_XMLSAVE packages, and the SYS_XMLGEN and SYS_XMLAGG functions to transfer XML data into and out of relational tables.

See Also:

- *Oracle XML DB Developer's Guide* for details about the XMLType datatype
- *Oracle XML Developer's Kit Programmer's Guide* for information about all aspects of working with XML
- *Oracle Database SQL Reference* for information about the SYS_XMLGEN and SYS_XMLAGG functions

3

Maintaining Data Integrity Through Constraints

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:

Overview of Integrity Constraints

You can define integrity constraints to enforce business rules on data in your tables. Business rules specify conditions and relationships that must always be true, or must always be false. Because each company defines its own policies about things like salaries, employee numbers, inventory tracking, and so on, you can specify a different set of rules for each database table.

When an integrity constraint applies to a table, all data in the table must conform to the corresponding rule. When you issue a SQL statement that modifies data in the table, Oracle Database ensures that the new data satisfies the integrity constraint, without the need to do any checking within your program.

When to Enforce Business Rules with Integrity Constraints

You can enforce rules by defining integrity constraints more reliably than by adding logic to your application. Oracle Database can check that all the data in a table obeys an integrity constraint faster than an application can.

Example of an Integrity Constraint for a Business Rule

To ensure that each employee works for a valid department, first create a rule that all values in the department table are *unique*:

```
ALTER TABLE Dept_tab
    ADD PRIMARY KEY (Deptno);
```

Then, create a rule that every department listed in the employee table must match one of the values in the department table:

```
ALTER TABLE Emp_tab
    ADD FOREIGN KEY (Deptno) REFERENCES Dept_tab(Deptno);
```

When you add a new employee record to the table, Oracle Database automatically checks that its department number appears in the department table.

To enforce this rule without integrity constraints, you can use a *trigger* to query the department table and test that each new employee's department is valid. But this method is less reliable than the integrity constraint. `SELECT` in Oracle Database uses "consistent read", so the query might miss uncommitted changes from other transactions.

When to Enforce Business Rules in Applications

You might enforce business rules through application logic as well as through integrity constraints, if you can filter out bad data before attempting an insert or update. This might let you provide instant feedback to the user, and reduce the load on the database. This technique is appropriate when you can determine that data values are wrong or out of range without checking against any data already in the table.

Creating Indexes for Use with Constraints

All enabled unique and primary keys require corresponding indexes. You should create these indexes by hand, rather than letting the database create them for you. Note that:

- Constraints use existing indexes where possible, rather than creating new ones.
- Unique and primary keys can use non-unique as well as unique indexes. They can even use just the first few columns of non-unique indexes.
- At most one unique or primary key can use each non-unique index.
- The column orders in the index and the constraint do not need to match.
- If you need to check whether an index is used by a constraint, for example when you want to drop the index, the object number of the index used by a unique or primary key constraint is stored in `CDEF$.ENABLED` for that constraint. It is not shown in any catalog view.

You should almost always index foreign keys; the database does not do this for you automatically.

When to Use 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, a new employee's manager or hire date might be temporarily omitted. Some employees might not have a commission. Columns like these should not have `NOT NULL` integrity constraints. However, an employee name might be required from the very beginning, and you can enforce this rule with 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' data will ever attempt to conflict with an existing row's data.

Because Oracle Database indexes do not store keys that are all null, if you want to allow index-only scans of the table or some other operation that requires indexing all rows, you must put a NOT NULL constraint on at least one indexed column.

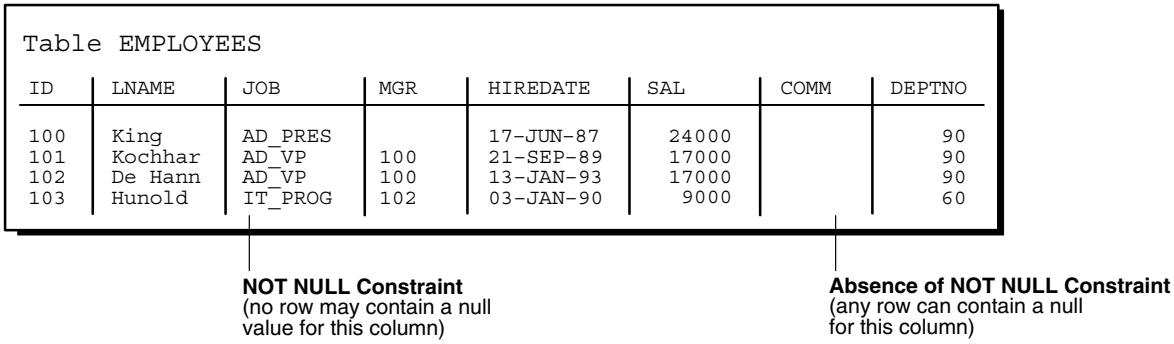
> **See Also:** "Defining Relationships Between Parent and Child Tables" on page 3-10

A NOT NULL constraint is specified like this:

```
ALTER TABLE emp MODIFY ename NOT NULL;
```

Figure 3–1shows an example of a table with NOT NULL integrity constraints.

Figure 3–1 Table with NOT NULL Integrity Constraints



When to Use Default Column Values

Assign default values to columns that contain a typical value. For example, in the DEPT_TAB table, if most departments are located at one site, then the default value for the LOC column can be set to this value (such as NEW YORK).

Default values can help avoid errors where there is a number, such as zero, that applies to a column that has no entry. For example, a default value of zero can simplify testing, by changing a test like this:

```
IF sal IS NOT NULL AND sal < 50000
```

to the simpler form:

```
IF sal < 50000
```

Depending upon your business rules, you might use default values to represent zero or false, or leave the default values as NULL to signify an unknown value.

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 through a view. The base table might also have a column named INSERTER, not included in the definition of the view, to log the user that inserts each row. To record the user name automatically, define a default value that calls the USER function:

```
CREATE TABLE audit_trail
(
    value1   NUMBER,
    value2   VARCHAR2(32),
    inserter VARCHAR2(30) DEFAULT USER
);
```

Setting Default Column Values

Default values can be defined using any literal, or almost any expression, including calls to SYSDATE, SYS_CONTEXT, USER, USERENV, and UID. Default values cannot include expressions that refer to a sequence, PL/SQL function, column, LEVEL, ROWNUM, or PRIOR. The datatype of a default literal or expression must match or be convertible to the column datatype.

Sometimes the default value is the result of a SQL function. For example, a call to SYS_CONTEXT can set a different default value depending on conditions such as the user name. To be used as a default value, a SQL function must have parameters that are all literals, cannot reference any columns, and cannot call any other functions.

If you do not explicitly define a default value for a column, the default for the column is implicitly set to NULL.

You can use the keyword DEFAULT within an INSERT statement instead of a literal value, and the corresponding default value is inserted.

Choosing a Table's Primary Key

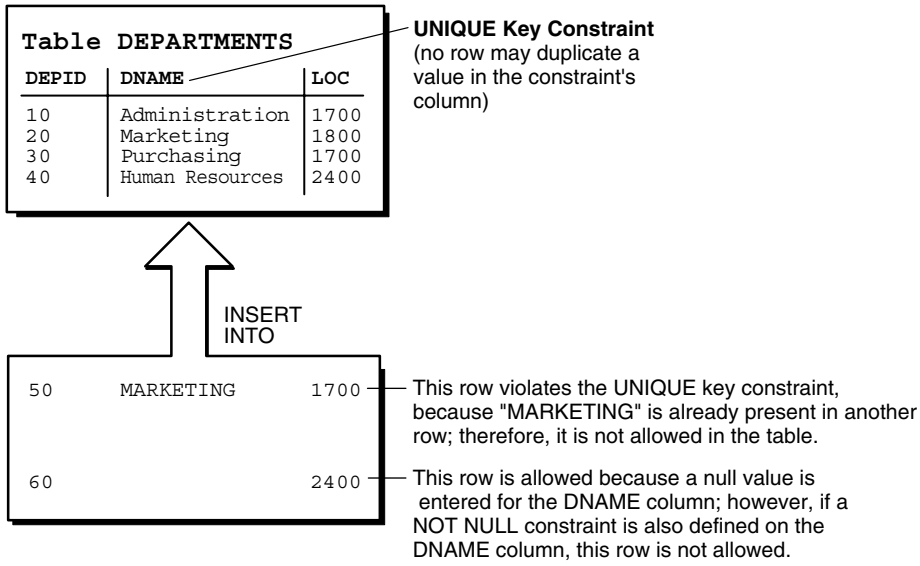
Each table can have one primary key, which uniquely identifies each row in a table and ensures that no duplicate rows exist. Use the following guidelines when selecting a primary key:

- Whenever practical, use a column containing a sequence number. It is a simple way to satisfy all the other guidelines.
- Choose a column whose data values are unique, because the purpose of a primary key is to uniquely identify each row of the table.
- Choose a column whose data values are never changed. A primary key value is only used to identify a row in the table, and its data should never be used for any other purpose. Therefore, primary key values should rarely or never be changed.
- Choose a column that does not contain any nulls. A `PRIMARY KEY` constraint, by definition, does not allow any row to contain 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.
- Minimize your use of composite primary keys. Although composite primary keys are allowed, they do not satisfy all of the other recommendations. For example, composite primary key values are long and cannot be assigned by sequence numbers.

When to Use UNIQUE Key Integrity Constraints

Choose columns for unique keys carefully. The purpose of these constraints is different from that of primary keys. Unique key constraints are appropriate for any column where duplicate values are not allowed. Primary keys identify each row of the table uniquely, and typically contain values that have no significance other than being unique. Figure 3–2 shows an example of a table with a unique key constraint.

Figure 3–2 Table with a UNIQUE Key Constraint



Note: You cannot have identical values in the non-null columns of a composite `UNIQUE` key constraint (`UNIQUE` key constraints allow `NULL` values).

Some examples of good unique keys include:

- An employee social security number (the primary key might be the employee number)
- A truck license plate number (the primary key might be the truck number)
- A customer phone number, consisting of the two columns `AREA_CODE` and `LOCAL_PHONE` (the primary key might be the customer number)
- A department name and location (the primary key might be the department number)

Constraints On Views: for Performance, Not Data Integrity

The constraints discussed throughout this chapter apply to tables, not views.

Although you can declare constraints on views, such constraints do not help maintain data integrity. Instead, they are used to enable query rewrites on queries involving views, which helps performance with materialized views and other data warehousing features. Such constraints are always declared with the `DISABLE` keyword, and you cannot use the `VALIDATE` keyword. The constraints are never enforced, and there is no associated index.

> **See Also:** *Oracle Data Warehousing Guide* for information on query rewrite, materialized views, and the performance reasons for declaring constraints on views

Enforcing Referential Integrity with Constraints

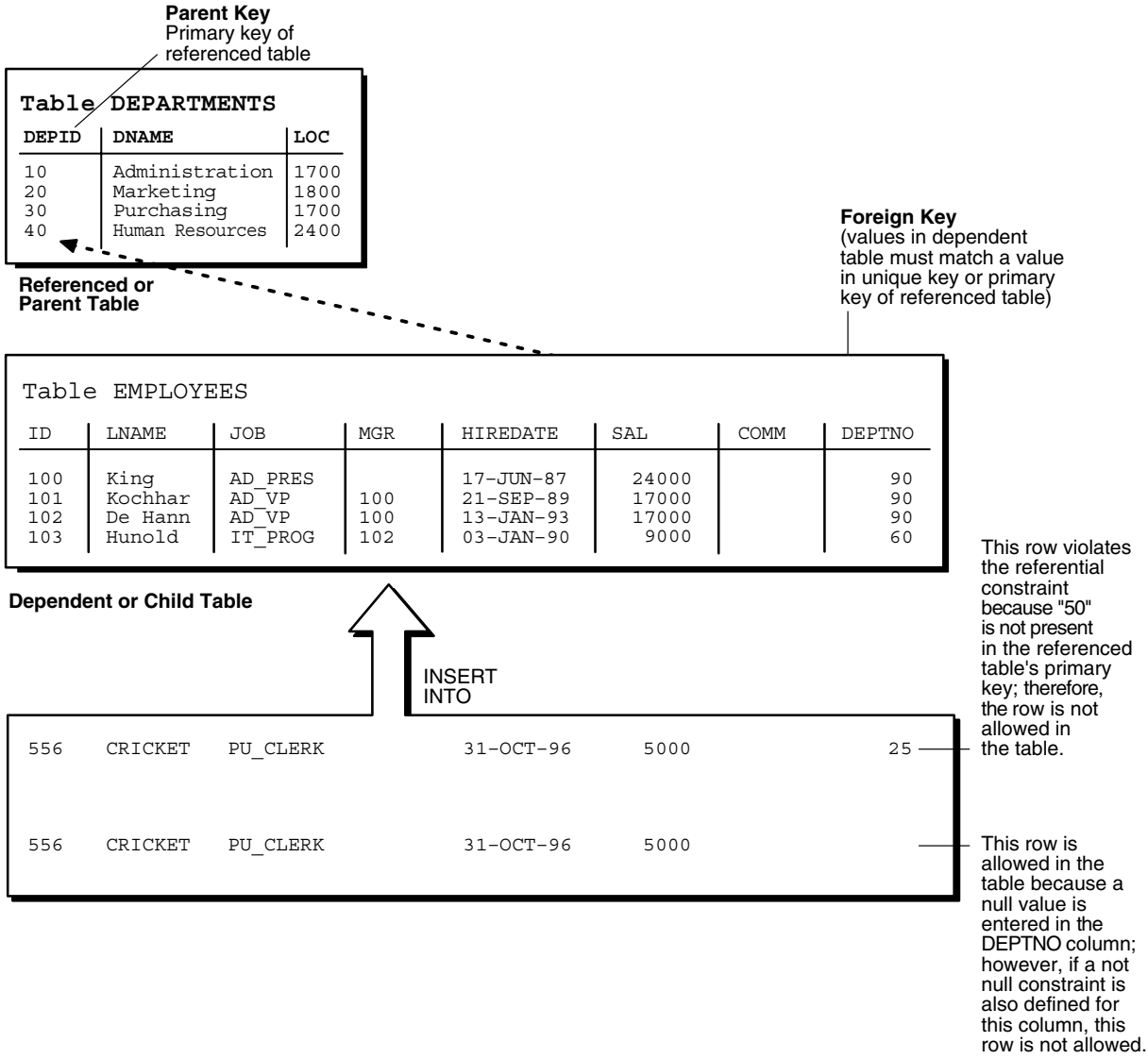
Whenever two tables contain one or more common columns, Oracle Database can enforce the relationship between the two tables through a referential integrity constraint. Define a `PRIMARY` or `UNIQUE` key constraint on the column in the parent table (the one that has the complete set of column values). Define a `FOREIGN KEY` constraint on the column in the child table (the one whose values must refer to existing values in the other table).

> **See Also:** "Defining Relationships Between Parent and Child Tables" on page 3-10 for information on defining additional integrity constraints, including the foreign key

Figure 3–3 shows a foreign key defined on the department number. It guarantees that every value in this column must match a value in the primary key of the department table. This constraint prevents erroneous department numbers from getting into the employee table.

Foreign keys can be comprised of multiple columns. Such a **composite foreign key** must reference a composite primary or unique key of the exact same structure, with the same number of columns and the same datatypes. Because composite primary and unique keys are limited to 32 columns, a composite foreign key is also limited to 32 columns.

Figure 3–3 Tables with Referential Integrity Constraints



About Nulls and Foreign Keys

Foreign keys allow key values that are all NULL, even if there are no matching PRIMARY or UNIQUE keys.

- By default (without any NOT NULL or CHECK clauses), the FOREIGN KEY constraint enforces the "match none" rule for composite foreign keys in the ANSI/ISO standard.
- To enforce the **match full** rule for NULL values 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. For example, with 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 NULL values in composite foreign keys, which requires the non-NULL portions of the key to appear in the corresponding portions in the primary or unique key of a single row in the referenced table. You can often use triggers to handle this case, as described in Chapter 9, "Using Triggers".

Defining 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 4–3 on page 8 between the employee and department tables. 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.

Any number of rows in the child table can reference the same parent key value, so 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, only one row in the child table can reference a given 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 employee table had a column named `MEMBERNO`, referring to an employee membership number in the company 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 insurance policy. The `MEMBERNO` in the employee table should be both a foreign key and a unique key:

- To enforce referential integrity rules between the `EMP_TAB` 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 given parent key value, and because `NULL` values 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 employee 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 employee table.

Rules for Multiple FOREIGN KEY Constraints

Oracle Database allows a column to be referenced by multiple `FOREIGN KEY` constraints; 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.

Deferring Constraint Checks

When Oracle Database checks a constraint, it signals an error if the constraint is not satisfied. You can use the `SET CONSTRAINTS` statement to defer checking the validity of constraints until the end of a transaction.

> **Note:** You cannot issue a `SET CONSTRAINT` statement inside a trigger.

The `SET CONSTRAINTS` setting lasts for the duration of the transaction, or until another `SET CONSTRAINTS` statement resets the mode.

> **See Also:** *Oracle Database SQL Reference* for more details on `SET CONSTRAINTS`

Guidelines for Deferring Constraint Checks

Select Appropriate Data You may wish to defer constraint checks on `UNIQUE` and `FOREIGN` keys if the data you are working with has any of the following characteristics:

- Tables are snapshots.
- Some tables contain a large amount of data being manipulated by another application, which may or may not return the data in the same order.
- Update cascade operations on foreign keys.

Ensure Constraints Are Created Deferrable After you have identified and selected the appropriate tables, make sure their `FOREIGN`, `UNIQUE` and `PRIMARY` key constraints are created deferrable. You can do so by issuing statements similar to the following:

```
CREATE TABLE dept (
     deptno NUMBER PRIMARY KEY,
     dname VARCHAR2 (30)
     );
CREATE TABLE emp (
     empno NUMBER,
     ename VARCHAR2 (30),
     deptno NUMBER REFERENCES (dept),
     CONSTRAINT epk PRIMARY KEY (empno) DEFERRABLE,
     CONSTRAINT efk FOREIGN KEY (deptno)
     REFERENCES (dept.deptno) DEFERRABLE);
```

```
INSERT INTO dept VALUES (10, 'Accounting');
INSERT INTO dept VALUES (20, 'SALES');
INSERT INTO emp VALUES (1, 'Corleone', 10);
INSERT INTO emp VALUES (2, 'Costanza', 20);
COMMIT;

SET CONSTRAINT efk DEFERRED;
UPDATE dept SET deptno = deptno + 10
     WHERE deptno = 20;

SELECT * from emp ORDER BY deptno;
EMPNO   ENAME          DEPTNO
-----   -------------- -------
   1    Corleone       10
   2    Costanza       20
UPDATE emp SET deptno = deptno + 10
     WHERE deptno = 20;
SELECT * FROM emp ORDER BY deptno;

EMPNO   ENAME          DEPTNO
-----   -------------- -------
   1    Corleone       10
   2    Costanza       30
COMMIT;
```

Set All Constraints Deferred Within the application that manipulates the data, you must set all constraints deferred before you begin processing any data. Use the following DML statement to set all constraints deferred:

```
SET CONSTRAINTS ALL DEFERRED;
```

> **Note:** The `SET CONSTRAINTS` statement applies only to the current transaction. The defaults specified when you create a constraint remain as long as the constraint exists. The `ALTER SESSION SET CONSTRAINTS` statement applies for the current session only.

Check the Commit (Optional) You can check for constraint violations before committing by issuing the `SET CONSTRAINTS ALL IMMEDIATE` statement just before issuing the `COMMIT`. If there are any problems with a constraint, this statement will fail and the constraint causing the error will be identified. If you commit while constraints are violated, the transaction will be rolled back and you will receive an error message.

Managing Constraints That Have Associated Indexes

When you create a UNIQUE or PRIMARY key, Oracle Database checks to see if an existing index can be used to enforce uniqueness for the constraint. If there is no such index, the database creates one.

Minimizing Space and Time Overhead for Indexes Associated with Constraints

When Oracle Database uses a unique index to enforce a constraint, and constraints associated with the unique index are dropped or disabled, the index is dropped. To preserve the statistics associated with the index (because, for example, it would take a long time to re-create it), you can specify the KEEP INDEX clause on the DROP command for the constraint.

While enabled foreign keys reference a PRIMARY or UNIQUE key, you cannot disable or drop the PRIMARY or UNIQUE key constraint or the index.

Note: UNIQUE and PRIMARY keys with deferrable constraints must all use non-unique indexes.

To reuse existing indexes when creating unique and primary key constraints, you can include USING INDEX in the constraint clause. For example:

```
CREATE TABLE b
(
    b1 INTEGER,
    b2 INTEGER,
    CONSTRAINT unique1 (b1, b2) USING INDEX (CREATE UNIQUE INDEX b_index on
b(b1, b2),
    CONSTRAINT unique2 (b1, b2) USING INDEX b_index
);
```

Guidelines for Indexing Foreign Keys

You should almost always index foreign keys. The only exception is when the matching unique or primary key is never updated or deleted.

See Also: *Oracle Database Concepts* for information on locking mechanisms involving indexes and keys

About Referential Integrity in a Distributed Database

The declaration of a referential integrity constraint cannot specify a foreign key that references a primary or unique key of a remote table.

However, you can maintain parent/child table relationships across nodes using triggers.

> **See Also:** Chapter 9, "Using Triggers" for more information about triggers that enforce referential integrity

---

Note: If you decide to define referential integrity across the nodes of a distributed database using triggers, be aware that network failures can make both the parent table and the child table inaccessible.

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, then some DML statements against the child table (those that insert rows or update a foreign key value) cannot proceed, because the referential integrity triggers must have access to the parent table in the `HQ` database.

When to Use CHECK Integrity Constraints

Use `CHECK` constraints when you need to enforce integrity rules based on logical expressions, such as comparisons. Never use `CHECK` constraints when any of the other types of integrity constraints can provide the necessary checking.

> **See Also:** "Choosing Between CHECK and NOT NULL Integrity Constraints" on page 3-17

Examples of `CHECK` constraints include the following:

- A `CHECK` constraint on employee salaries so that no salary value is greater than 10000.
- A `CHECK` constraint on department locations so that only the locations "`BOSTON`", "`NEW YORK`", and "`DALLAS`" are allowed.

- A CHECK constraint on the salary and commissions columns to prevent the commission from being larger than the salary.

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, then 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: *Oracle Database SQL Reference* for an explanation of these pseudocolumns

- The condition cannot contain a user-defined SQL function.

Designing CHECK Constraints

When using CHECK constraints, remember that a CHECK constraint is violated only if the condition evaluates to false; true and unknown values (such as comparisons with nulls) do not violate a check condition. Make sure that any CHECK constraint that you define is specific enough to enforce the rule.

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 employee table unless the employee salary is greater than zero or the employee commission is greater than or equal to zero." But if a row is inserted with a null salary, that row does not violate the CHECK constraint, regardless of whether or not the commission value is valid, because the entire check condition is evaluated as unknown. In this case, you can prevent 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 *Oracle Database SQL Reference*

Rules for 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.

The order in which the constraints are evaluated is not defined, so be careful not to rely on the order or to define multiple constraints that conflict with each other.

Choosing Between 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 the following:

```
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))
```

Examples of Defining Integrity Constraints

Here are some examples showing how to create simple constraints during the prototype phase of your database design.

Each constraint is given a name in these examples. Naming the constraints prevents the database from creating multiple copies of the same constraint, with different system-generated names, if the DDL is run multiple times.

See Also: *Oracle Database Administrator's Guide* for information on creating and maintaining constraints for a large production database

Example: Defining Integrity Constraints with the CREATE TABLE Command

The following examples of CREATE TABLE statements show the definition of several integrity constraints:

```
CREATE TABLE Dept_tab (
    Deptno  NUMBER(3) CONSTRAINT Dept_pkey 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_tab (
    Empno    NUMBER(5) CONSTRAINT Emp_pkey PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
    Mgr      NUMBER(5) CONSTRAINT Mgr_fkey REFERENCES Emp_tab,
    Hiredate DATE,
    Sal      NUMBER(7,2),
    Comm     NUMBER(5,2),
    Deptno   NUMBER(3) NOT NULL
             CONSTRAINT dept_fkey REFERENCES Dept_tab ON DELETE CASCADE);
```

Example: Defining Constraints with the ALTER TABLE Command

You can also define integrity constraints using the constraint clause of the ALTER TABLE command. For example:

```
CREATE UNIQUE INDEX I_dept ON Dept_tab(deptno);
ALTER TABLE Dept_tab
    ADD CONSTRAINT Dept_pkey PRIMARY KEY (deptno);

ALTER TABLE Emp_tab
    ADD CONSTRAINT Dept_fkey FOREIGN KEY (Deptno) REFERENCES Dept_tab;
ALTER TABLE Emp_tab MODIFY (Ename VARCHAR2(15) NOT NULL);
```

You cannot create a validated constraint on a table if the table already contains any rows that would violate the constraint.

Privileges Required to Create Constraints

The creator of a constraint must have the ability to create tables (the CREATE TABLE or CREATE ANY TABLE system privilege), or the ability to alter the table (the ALTER object privilege for the table or the ALTER ANY TABLE system privilege) with the constraint. Additionally, UNIQUE and PRIMARY KEY integrity constraints require that the owner of the table have either a quota for the tablespace that contains the associated index or the UNLIMITED TABLESPACE system privilege. FOREIGN KEY integrity constraints also require some additional privileges.

See Also: "Privileges Required to Create FOREIGN KEY Integrity Constraints" on page 3-27

Naming Integrity Constraints

Assign names to constraints NOT NULL, UNIQUE KEY, PRIMARY KEY, FOREIGN KEY, and CHECK 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 automatically by Oracle Database.

Picking your own name makes error messages for constraint violations more understandable, and prevents the creation of duplicate constraints with different names if the SQL statements are run more than once.

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: "Viewing Definitions of Integrity Constraints" on page 3-28 for examples of data dictionary views

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 corresponding rule is enforced on the data values in the associated columns. The definition of the constraint is stored in the data dictionary.

disabled constraint. When a constraint is disabled, the corresponding rule is not enforced. The definition of the constraint is still stored in the data dictionary.

An integrity constraint represents an assertion about the data in a database. This assertion is always true when the constraint is enabled. The assertion may or may not be true when the constraint is disabled, because data that violates the integrity constraint can be in the database.

Why Disable Constraints?

During day-to-day operations, constraints should always be enabled. In certain situations, temporarily disabling the integrity constraints of a table makes sense 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 each employee number by adding 1000 to the existing number)
- When importing or exporting one table at a time

Temporarily turning off integrity constraints can speed up these operations.

About Exceptions to Integrity Constraints

If a row of a table disobeys an integrity constraint, then this row is in violation of the constraint and is called an **exception** to the constraint. If any exceptions exist, then the constraint *cannot* be enabled. The rows that violate the constraint must be updated or deleted before the constraint can be enabled.

You can identify exceptions for a specific integrity constraint as you try to enable the constraint.

> **See Also:** "Fixing Constraint Exceptions" on page 3-23 for more information on this procedure

Enabling Constraints

When you define an integrity constraint in a `CREATE TABLE` or `ALTER TABLE` statement, Oracle Database automatically enables the constraint by default. For code clarity, you can explicitly enable the constraint by including the `ENABLE` clause in its definition.

Use this technique when creating tables that start off empty, and are populated a row at a time by individual transactions. In such cases, you want to ensure that data is consistent at all times, and the performance overhead of each DML operation is small.

The following CREATE TABLE and ALTER TABLE statements both define and enable integrity constraints:

```
CREATE TABLE Emp_tab (
    Empno NUMBER(5) PRIMARY KEY);
 ALTER TABLE Emp_tab
    ADD PRIMARY KEY (Empno);
```

An ALTER TABLE statement that tries to enable an integrity constraint will fail if any existing rows of the table violate the integrity constraint. The statement is rolled back and the constraint definition is not stored and not enabled.

> **See Also:** "Fixing Constraint Exceptions" on page 3-23 for more information about rows that violate integrity constraints

Creating Disabled Constraints

The following CREATE TABLE and ALTER TABLE statements both define and disable integrity constraints:

```
CREATE TABLE Emp_tab (
    Empno NUMBER(5) PRIMARY KEY DISABLE);

ALTER TABLE Emp_tab
    ADD PRIMARY KEY (Empno) DISABLE;
```

Use this technique when creating tables that will be loaded with large amounts of data before anybody else accesses them, particularly if you need to cleanse data after loading it, or need to fill empty columns with sequence numbers or parent/child relationships.

An ALTER TABLE statement that defines and disables an integrity constraints never fails, because its rule is not enforced.

Enabling and Disabling Existing 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 Existing Constraints

Once you have finished cleansing data and filling empty columns, you can enable constraints that were disabled during data loading.

The following statements are examples of statements that enable disabled integrity constraints:

```
ALTER TABLE Dept_tab
    ENABLE CONSTRAINT Dname_ukey;

ALTER TABLE Dept_tab
    ENABLE PRIMARY KEY
    ENABLE UNIQUE (Dname)
    ENABLE UNIQUE (Loc);
```

An `ALTER TABLE` statement that attempts to enable an integrity constraint fails if any of the table rows violate the integrity constraint. The statement is rolled back and the constraint is not enabled.

> **See Also:** "Fixing Constraint Exceptions" on page 3-23 for more information about rows that violate integrity constraints

Disabling Existing Constraints

If you need to perform a large load or update when a table already contains data, you can temporarily disable constraints to improve performance of the bulk operation.

The following statements are examples of statements that disable enabled integrity constraints:

```
ALTER TABLE Dept_tab
    DISABLE CONSTRAINT Dname_ukey;

ALTER TABLE Dept_tab
    DISABLE PRIMARY KEY
    DISABLE UNIQUE (Dname)
    DISABLE UNIQUE (Loc);
```

Tip: Using the Data Dictionary to Find Constraints

The preceding examples require that you know the relevant constraint names and which columns they affect. To find this information, you can query one of the data dictionary views defined for constraints, `USER_CONSTRAINTS` or `USER_CONS_COLUMNS`. For more information about these views, see "Viewing Definitions of Integrity Constraints" on page 3-28 and *Oracle Database Reference*.

Guidelines for Enabling and Disabling Key Integrity Constraints

When enabling or disabling UNIQUE, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. UNIQUE key and PRIMARY KEY constraints are usually managed by the database administrator.

> **See Also:** *Oracle Database Administrator's Guide* and "Managing FOREIGN KEY Integrity Constraints" on page 3-26

Fixing Constraint Exceptions

When you try to create or enable a constraint, and the statement fails because integrity constraint exceptions exist, the statement is rolled back. You cannot enable the constraint until all exceptions 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.

> **See Also:** *Oracle Database Administrator's Guide* for more information about responding to constraint exceptions

Altering Integrity Constraints

Starting with Oracle8*i*, you can alter the state of an existing constraint with the MODIFY CONSTRAINT clause.

> **See Also:** *Oracle Database SQL Reference* for information on the parameters you can modify

MODIFY CONSTRAINT Example #1

The following commands show several alternatives for whether the CHECK constraint is enforced, and when the constraint checking is done:

```
CREATE TABLE X1_tab (a1 NUMBER CONSTRAINT y CHECK (a1>3) DEFERRABLE DISABLE);

ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt RELY;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt INITIALLY DEFERRED;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE NOVALIDATE;
```

MODIFY CONSTRAINT Example #2

The following commands show several alternatives for whether the NOT NULL constraint is enforced, and when the checking is done:

```
CREATE TABLE X1_tab (A1 NUMBER CONSTRAINT Y_cnstrt
NOT NULL DEFERRABLE INITIALLY DEFERRED NORELY DISABLE);

ALTER TABLE X1_tab ADD CONSTRAINT One_cnstrt UNIQUE(A1)
DEFERRABLE INITIALLY IMMEDIATE RELY USING INDEX PCTFREE = 30
ENABLE VALIDATE;

ALTER TABLE X1_tab MODIFY UNIQUE(A1)
INITIALLY DEFERRED NORELY USING INDEX PCTFREE = 40
ENABLE NOVALIDATE;

ALTER TABLE X1_tab MODIFY CONSTRAINT One_cnstrt
INITIALLY IMMEDIATE RELY;
```

Modify Constraint Example #3

The following commands show several alternatives for whether the primary key constraint is enforced, and when the checking is done:

```
CREATE TABLE T1_tab (A1 INT, B1 INT);
ALTER TABLE T1_tab add CONSTRAINT P1_cnstrt PRIMARY KEY(a1) DISABLE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY INITIALLY IMMEDIATE
USING INDEX PCTFREE = 30 ENABLE NOVALIDATE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY
USING INDEX PCTFREE = 35 ENABLE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY ENABLE NOVALIDATE;
```

Renaming Integrity Constraints

Because constraint names must be unique, even across multiple schemas, you can encounter problems when you want to clone a table and all its constraints, because the constraint name for the new table conflicts with the one for the original table. Or, you might create a constraint with a default system-generated name, and later realize that you want to give the constraint a name that is easy to remember, so that you can easily enable and disable it.

One of the properties you can alter for a constraint is its name. The following SQL*Plus script finds the system-generated name for a constraint and changes it:

```
prompt Enter table name to find its primary key:
accept table_name
select constraint_name from user_constraints
  where table_name = upper('&table_name.')
  and constraint_type = 'P';

prompt Enter new name for its primary key:
accept new_constraint

set serveroutput on

declare
-- USER_CONSTRAINTS.CONSTRAINT_NAME is declared as VARCHAR2(30).
-- Using %TYPE here protects us if the length changes in a future release.
  constraint_name user_constraints.constraint_name%type;
begin
  select constraint_name into constraint_name from user_constraints
    where table_name = upper('&table_name.')
    and constraint_type = 'P';

  dbms_output.put_line('The primary key for ' || upper('&table_name.') || ' is:
' || constraint_name);

  execute immediate
    'alter table &table_name. rename constraint ' || constraint_name ||
    ' to &new_constraint.';
end;
/
```

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_tab
    DROP UNIQUE (Dname);
ALTER TABLE Dept_tab
    DROP UNIQUE (Loc);

ALTER TABLE Emp_tab
```

```
    DROP PRIMARY KEY,
    DROP CONSTRAINT Dept_fkey;

DROP TABLE Emp_tab CASCADE CONSTRAINTS;
```

When dropping UNIQUE, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. UNIQUE and PRIMARY KEY constraints are usually managed by the database administrator.

> **See Also:** *Oracle Database Administrator's Guide* and "Managing FOREIGN KEY Integrity Constraints" on page 3-26

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, which enforce relationships between columns in different tables.

> **Note:** FOREIGN KEY integrity constraints cannot be enabled if the constraint of the referenced primary or unique key is not present or not enabled.

Datatypes and Names for Foreign Key Columns

You must use the same datatype for corresponding columns in the dependent and referenced tables. The column names do not need to match.

Limit on Columns in 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 32 columns.

Foreign Key References Primary Key by Default

If the column list is not included in the REFERENCES option when defining a FOREIGN KEY constraint (single column or composite), then Oracle Database 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 Database automatically checks to verify that this column list references a primary or unique key of the parent table. If it does not, then an informative error is returned.

Privileges Required to Create FOREIGN KEY Integrity Constraints

To create a FOREIGN KEY constraint, the creator of the constraint must have privileged access to the parent and child tables.

- **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 through 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 which constraints are enforced and which other users can create constraints
- The owner of the parent table to explicitly decide if foreign keys can depend on the primary and unique keys in her tables

Choosing How Foreign Keys Enforce Referential Integrity

Oracle Database allows different types of referential integrity actions to be enforced, as specified with the definition of a FOREIGN KEY constraint:

- **Prevent Delete or Update of Parent Key** The default setting prevents the deletion or update of a parent key if there is a row in the child table that references the key. For example:

```
CREATE TABLE Emp_tab (
FOREIGN KEY (Deptno) REFERENCES Dept_tab);
```

- **Delete Child Rows When Parent Key Deleted** The ON DELETE CASCADE action allows parent key data that is referenced from the child table to be deleted, but not updated. When 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_tab (
    FOREIGN KEY (Deptno) REFERENCES Dept_tab
        ON DELETE CASCADE);
```

- **Set Foreign Keys to Null When Parent Key Deleted** The ON DELETE SET NULL action allows data that references the parent key to be deleted, but not updated. When referenced data in the parent key is deleted, all rows in the child table that depend on those parent key values have their foreign keys set to null. To specify this referential action, include the ON DELETE SET NULL option in the definition of the FOREIGN KEY constraint. For example:

```
CREATE TABLE Emp_tab (
    FOREIGN KEY (Deptno) REFERENCES Dept_tab
        ON DELETE SET NULL);
```

Viewing Definitions of Integrity Constraints

The data dictionary contains the following views that relate to integrity constraints:

- ALL_CONSTRAINTS
- ALL_CONS_COLUMNS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- DBA_CONSTRAINTS
- DBA_CONS_COLUMNS

You can query these views to find the names of constraints, what columns they affect, and other information to help you manage constraints.

See Also: *Oracle Database Reference* for information on each view

Examples of Defining Integrity Constraints

The following CREATE TABLE statements define a number of integrity constraints:

```
CREATE TABLE Dept_tab (
    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_tab (
    Empno    NUMBER(5) PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
    Mgr      NUMBER(5) CONSTRAINT Mgr_fkey
         REFERENCES Emp_tab ON DELETE CASCADE,
    Hiredate DATE,
    Sal      NUMBER(7,2),
    Comm     NUMBER(5,2),
    Deptno   NUMBER(3) NOT NULL
    CONSTRAINT Dept_fkey REFERENCES Dept_tab);
```

Example 1: Listing All of Your Accessible Constraints The following query lists all constraints defined on all tables accessible to 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 this is returned:

```
CONSTRAINT_NAME  C TABLE_NAME  R_CONSTRAINT_NAME
---------------  - ----------- ------------------
SYS_C00275       P DEPT_TAB
DNAME_UKEY       U DEPT_TAB
LOC_CHECK1       C DEPT_TAB
SYS_C00278       C EMP_TAB
SYS_C00279       C EMP_TAB
SYS_C00280       P EMP_TAB
MGR_FKEY         R EMP_TAB     SYS_C00280
DEPT_FKEY        R EMP_TAB     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 following table 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 using the WITH CHECK OPTION for views.

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_TAB and DEPT_TAB tables, submit the following query:

```
SELECT Constraint_name, Search_condition
    FROM User_constraints
    WHERE (Table_name = 'DEPT_TAB' OR Table_name = 'EMP_TAB') AND
        Constraint_type = 'C';
```

Considering the example CREATE TABLE statements at the beginning of this section, a list similar to this 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 that the following are explicitly listed in the SEARCH_CONDITION column:

- NOT NULL constraints
- The conditions for user-defined CHECK constraints

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 this is returned:

```
CONSTRAINT_NAME  TABLE_NAME  COLUMN_NAME
---------------  ----------- ---------------
DEPT_FKEY        EMP_TAB     DEPTNO
DNAME_UKEY       DEPT_TAB    DNAME
DNAME_UKEY       DEPT_TAB    LOC
LOC_CHECK1       DEPT_TAB    LOC
MGR_FKEY         EMP_TAB     MGR
SYS_C00275       DEPT_TAB    DEPTNO
SYS_C00278       EMP_TAB     ENAME
SYS_C00279       EMP_TAB     DEPTNO
SYS_C00280       EMP_TAB     EMPNO
```

4

Selecting an Index Strategy

This chapter discusses considerations for using the different types of indexes in an application. The topics include:

- Guidelines for Application-Specific Indexes
- Creating Indexes: Basic Examples
- When to Use Domain Indexes
- When to Use Function-Based Indexes

See Also:

- *Oracle Database Administrator's Guide* for information about creating and managing indexes
- *Oracle Database Performance Tuning Guide* for detailed information about using indexes
- *Oracle Database SQL Reference* for the syntax of commands to work with indexes
- *Oracle Database Administrator's Guide* for information on creating hash clusters to improve performance, as an alternative to indexing

Guidelines for Application-Specific Indexes

You can create indexes on columns to speed up queries. Indexes provide faster access to data for operations that return a small portion of a table's rows.

In general, you should create an index on a column in any of the following situations:

- The column is queried frequently.
- A referential integrity constraint exists on the column.
- A `UNIQUE` key integrity constraint exists on the column.

You can create an index on any column; however, if the column is not used in any of these situations, creating an index on the column does not increase performance and the index takes up resources unnecessarily.

Although the database creates an index for you on a column with an integrity constraint, explicitly creating an index on such a column is recommended.

You can use the following techniques to determine which columns are best candidates for indexing:

- Use the `EXPLAIN PLAN` feature to show a theoretical execution plan of a given query statement.
- Use the `V$SQL_PLAN` view to determine the actual execution plan used for a given query statement.

Sometimes, if an index is not being used by default and it would be most efficient to use that index, you can use a query hint so that the index is used.

> **See Also:** *Oracle Database Performance Tuning Guide* for information on using the `V$SQL_PLAN` view, the `EXPLAIN PLAN` statement, query hints, and measuring the performance benefits of indexes

The following sections explain how to create, alter, and drop indexes using SQL commands, and give guidelines for managing indexes.

Create Indexes After Inserting Table Data

Typically, you insert or load data into a table (using SQL*Loader or Import) before creating indexes. Otherwise, the overhead of updating the index slows down the insert or load operation. The *exception* to this rule is that you must create an index for a cluster before you insert any data into the cluster.

Switch Your Temporary Tablespace to Avoid Space Problems Creating Indexes

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

> **See Also:** *Oracle Database Utilities* for information on direct path load

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 about 15% of the rows in a large table. This threshold percentage varies greatly, however, 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 that are used for joins to improve join performance.
- Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key; see Chapter 3, "Maintaining Data Integrity Through Constraints" for more information.
- Small tables do not require indexes; if a query is taking too long, then 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 good candidates for indexing:

- Values are unique in the column, or there are few duplicates.
- There is a wide range of values (good for regular indexes).
- There is a small range of values (good for bitmap indexes).
- The column contains many nulls, but queries often select all rows having a value. In this case, a comparison that matches all the non-null values, such as:

  ```
  WHERE COL_X >= -9.99 *power(10,125)
  ```

  is preferable to

  ```
  WHERE COL_X IS NOT NULL
  ```

  This is 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:

- 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 for Each Table

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.

You must weigh the performance benefit of indexes for queries against the performance overhead of updates. For example, if a table is primarily read-only, you might use more indexes; but, if a table is heavily updated, you might use fewer indexes.

Choose the Order of Columns in Composite Indexes

Although you can specify columns in any order in the `CREATE INDEX` command, the order of columns in the `CREATE INDEX` statement can affect query performance. In general, you should put the column expected to be used most often first in the

index. You can create a composite index (using several columns), and the same index can be used for queries that reference all of these columns, or just some of them.

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);
```

Composite indexes speed up queries that use the *leading portion* of the index. So in this 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.

Gather Statistics to Make Index Usage More Accurate

The database can use indexes more effectively when it has statistical information about the tables involved in the queries. You can gather statistics when the indexes are created by including the keywords COMPUTE STATISTICS in the CREATE INDEX statement. As data is updated and the distribution of values changes, you or

the DBA can periodically refresh the statistics by calling procedures like DBMS_STATS.GATHER_TABLE_STATISTICS and DBMS_STATS.GATHER_SCHEMA_STATISTICS.

Drop Indexes That Are No Longer Required

You might drop an index if:

- It does not speed up queries. The table might be very small, or there might be many rows in the table but very few index entries.
- The queries in your applications do not use the index.
- The index 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, the following statement drops a specific named index:

```
DROP INDEX Emp_ename;
```

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

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

Privileges Required to Create an Index

When using indexes in an application, you might need to request that the DBA grant privileges or make changes to initialization parameters.

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.

Creating Indexes: Basic Examples

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 32 columns. A composite index key cannot exceed roughly one-half (minus some overhead) of the available space in the data block.

Oracle Database 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, instead of using the obsolete CREATE UNIQUE INDEX syntax.

Use the SQL command CREATE INDEX to create an index.

In this example, an index is created for a single column, to speed up queries that test that column:

```
CREATE INDEX emp_ename ON emp_tab(ename);
```

In this example, several storage settings are explicitly specified for the index:

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

In this example, the index applies to two columns, to speed up queries that test either the first column or both columns:

```
CREATE INDEX emp_ename ON emp_tab(ename, empno) COMPUTE STATISTICS;
```

In this example, the query is going to sort on the function UPPER(ENAME). An index on the ENAME column itself would not speed up this operation, and it might be slow to call the function for each result row. A function-based index precomputes the result of the function for each column value, speeding up queries that use the function for searching or sorting:

```
CREATE INDEX emp_upper_ename ON emp_tab(UPPER(ename)) COMPUTE STATISTICS;
```

When to Use Domain Indexes

Domain indexes are appropriate for special-purpose applications implemented using data cartridges. The domain index helps to manipulate complex data, such as spatial, audio, or video data. If you need to develop such an application, see *Oracle Data Cartridge Developer's Guide*.

Oracle Database supplies a number of specialized data cartridges to help manage these kinds of complex data. So, if you need to create a search engine, or a

geographic information system, you can do much of the work simply by creating the right kind of index.

When to Use Function-Based Indexes

A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.

Note:

- The index is more effective if you gather statistics for the table or schema, using the procedures in the `DBMS_STATS` package.
- The index cannot contain any null values. Either make sure the appropriate columns contain no null values, or use the `NVL` function in the index expression to substitute some other value for nulls.

The expression indexed by a function-based index can be an arithmetic expression or an expression that contains a PL/SQL function, package function, C callout, or SQL function. Function-based indexes also support linguistic sorts based on collation keys, efficient linguistic collation of SQL statements, and case-insensitive sorts.

Like other indexes, function-based indexes improve query performance. For example, if you need to access a computationally complex expression often, then you can store it in an index. Then when you need to access the expression, it is already computed. You can find a detailed description of the advantages of function-based indexes in "Advantages of Function-Based Indexes" on page 4-9.

Function-based indexes have all of the same properties as indexes on columns. However, unlike indexes on columns which can be used by both cost-based and rule-based optimization, function-based indexes can be used by only by cost-based optimization. Other restrictions on function-based indexes are described in "Restrictions for Function-Based Indexes" on page 4-11.

See Also:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide* for information on creating function-based indexes

Advantages of Function-Based Indexes

Function-based indexes:

- *Increase the number of situations where the optimizer can perform a range scan instead of a full table scan.* For example, consider the expression in this `WHERE` clause:

  ```
  CREATE INDEX Idx ON Example_tab(Column_a + Column_b);
  SELECT * FROM Example_tab WHERE Column_a + Column_b < 10;
  ```

  The optimizer can use a range scan for this query because the index is built on (`column_a` + `column_b`). Range scans typically produce fast response times if the predicate selects less than 15% of the rows of a large table. The optimizer can estimate how many rows are selected by expressions more accurately if the expressions are materialized in a function-based index. (Expressions of function-based indexes are represented as virtual columns and `ANALYZE` can build histograms on such columns.)

- *Precompute the value of a computationally intensive function and store it in the index.* An index can store computationally intensive expression that you access often. When you need to access a value, it is already computed, greatly improving query execution performance.

- *Create indexes on object columns and REF columns.* Methods that describe objects can be used as functions on which to build indexes. For example, you can use the `MAP` method to build indexes on an object type column.

- *Create more powerful sorts.* You can perform case-insensitive sorts with the `UPPER` and `LOWER` functions, descending order sorts with the `DESC` keyword, and linguistic-based sorts with the `NLSSORT` function.

  Note: Oracle Database sorts columns with the `DESC` keyword in descending order. Such indexes are treated as function-based indexes. Descending indexes cannot be bitmapped or reverse, and cannot be used in bitmapped optimizations. To get the `DESC` functionality prior to Oracle Database version 8, remove the `DESC` keyword from the `CREATE INDEX` statement.

Another function-based index calls the object method `distance_from_equator` for each city in the table. The method is applied to the object column `Reg_Obj`. A query could use this index to quickly find cities that are more than 1000 miles from the equator:

```
CREATE INDEX Distance_index
ON Weatherdata_tab (Distance_from_equator (Reg_obj));

SELECT * FROM Weatherdata_tab
WHERE (Distance_from_equator (Reg_Obj)) > '1000';
```

Another index stores the temperature delta and the maximum temperature. The result of the delta is sorted in descending order. A query could use this index to quickly find table rows where the temperature delta is less than 20 and the maximum temperature is greater than 75.

```
CREATE INDEX compare_index
ON Weatherdata_tab ((Maxtemp - Mintemp) DESC, Maxtemp);

SELECT * FROM Weatherdata_tab
WHERE ((Maxtemp - Mintemp) < '20' AND Maxtemp > '75');
```

Examples of Function-Based Indexes

Example: Function-Based Index for Case-Insensitive Searches

The following command allows faster case-insensitive searches in table EMP_TAB.

```
CREATE INDEX Idx ON Emp_tab (UPPER(Ename));
```

The SELECT command uses the function-based index on UPPER(e_name) to return all of the employees with name like :KEYCOL.

```
SELECT * FROM Emp_tab WHERE UPPER(Ename) like :KEYCOL;
```

Example: Precomputing Arithmetic Expressions with a Function-Based Index

The following command computes a value for each row using columns A, B, and C, and stores the results in the index.

```
CREATE INDEX Idx ON Fbi_tab (A + B * (C - 1), A, B);
```

The SELECT statement can either use index range scan (since the expression is a prefix of index IDX) or index fast full scan (which may be preferable if the index has specified a high parallel degree).

```
SELECT a FROM Fbi_tab WHERE A + B * (C - 1) < 100;
```

Example: Function-Based Index for Language-Dependent Sorting

This example demonstrates how a function-based index can be used to sort based on the collation order for a national language. The NLSSORT function returns a sort key for each name, using the collation sequence GERMAN.

```
CREATE INDEX Nls_index
    ON Nls_tab (NLSSORT(Name, 'NLS_SORT = German'));
```

The SELECT statement selects all of the contents of the table and orders it by NAME. The rows are ordered using the German collation sequence. The Globalization Support parameters are not needed in the SELECT statement, because in a German session, NLS_SORT is set to German and NLS_COMP is set to ANSI.

```
SELECT * FROM Nls_tab WHERE Name IS NOT NULL
    ORDER BY Name;
```

Restrictions for Function-Based Indexes

Note the following restrictions for function-based indexes:

- Only cost-based optimization can use function-based indexes. Remember to call DBMS_STATS.GATHER_TABLE_STATISTICS or DBMS_STATS.GATHER_SCHEMA_STATISTICS, for the function-based index to be effective.
- Any top-level or package-level PL/SQL functions that are used in the index expression must be declared as DETERMINISTIC. That is, they always return the same result given the same input, like the UPPER function. You must ensure that the subprogram really is deterministic, because Oracle Database does not check that the assertion is true.

  The following semantic rules demonstrate how to use the keyword DETERMINISTIC:

  - A top level subprogram can be declared as DETERMINISTIC.
  - A PACKAGE level subprogram can be declared as DETERMINISTIC in the PACKAGE specification but not in the PACKAGE BODY. Errors are raised if DETERMINISTIC is used inside a PACKAGE BODY.
  - A private subprogram (declared inside another subprogram or a PACKAGE BODY) cannot be declared as DETERMINISTIC.
  - A DETERMINISTIC subprogram can call another subprogram whether the called program is declared as DETERMINISTIC or not.

- Expressions used in a function-based index cannot contain any aggregate functions. The expressions should reference only columns in a row in the table.
- You must analyze the table or index before the index is used.
- Bitmap optimizations cannot use descending indexes.
- Function-based indexes are not used when OR-expansion is done.
- The index function cannot be marked `NOT NULL`. To avoid a full table scan, you must ensure that the query cannot fetch null values.
- Function-based indexes cannot use expressions that return `VARCHAR2` or `RAW` data types of unknown length from PL/SQL functions. A workaround is to limit the size of the function's output by indexing a substring of known length:

```
-- INITIALS() might return 1 letter, 2 letters, 3 letters, and so on.
-- We limit the return value to 10 characters for purposes of the index.
CREATE INDEX func_substr_index ON
  emp_tab(substr(initials(ename),1,10);

-- Call SUBSTR both when creating the index and when referencing
-- the function in queries.
SELECT SUBSTR(initials(ename),1,10) FROM emp_tab;
```

5

How Oracle Database Processes SQL Statements

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

Overview of SQL Statement Execution

Figure 5–1 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 do not need to be concerned with, or aware of, this level of detail. However, you might find this information useful when writing Oracle Database applications.

> **See Also:** *Oracle Database Concepts* for a description of the processing stages for each type of SQL statement

Identifying Extensions to SQL92 (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 Database 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, then Oracle Database flags them as errors and displays the violating syntax.

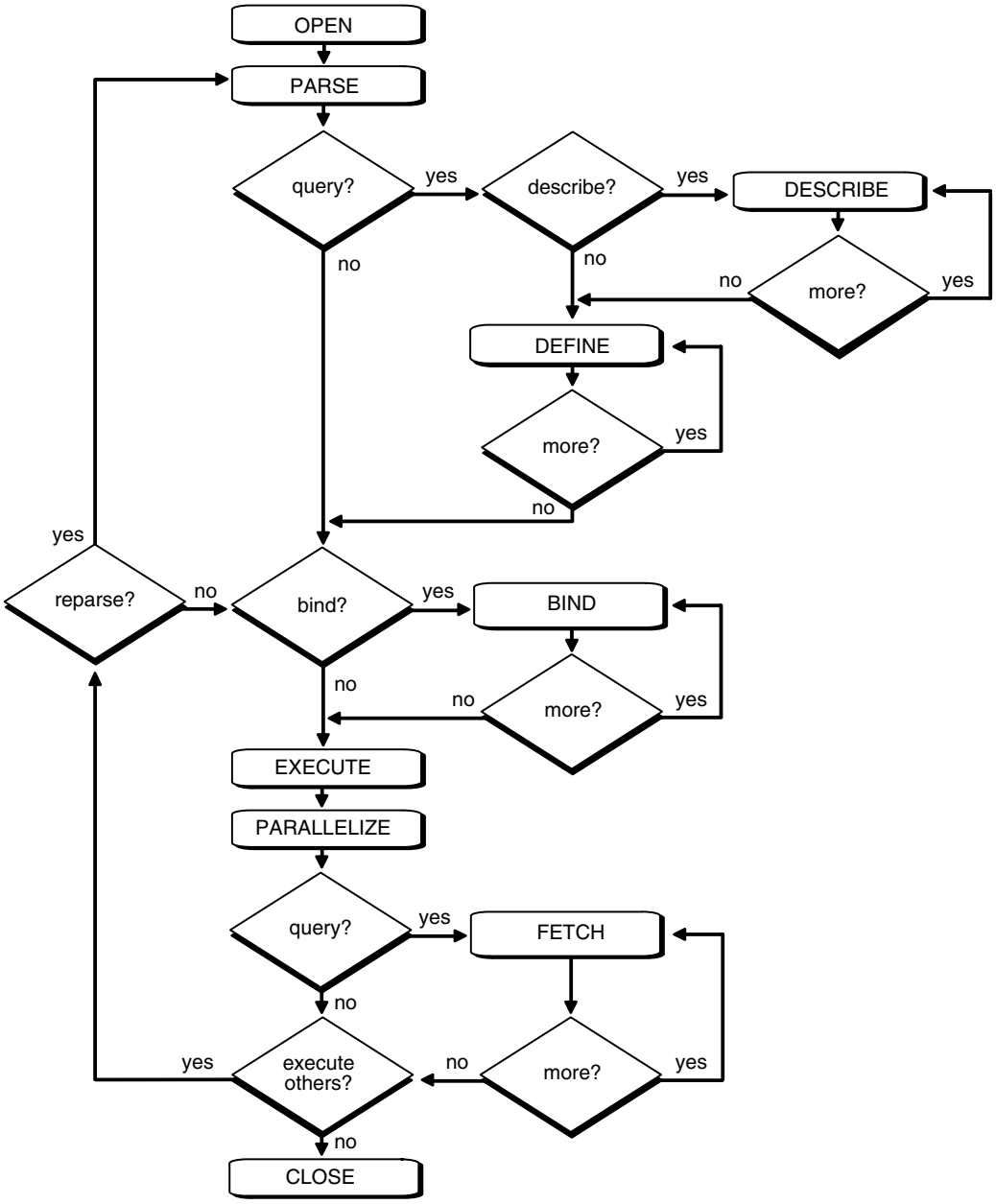
The FIPS flagging feature supports flagging through interactive SQL statements submitted using Oracle Enterprise Manager or SQL*Plus. The Oracle precompilers and SQL*Module also support FIPS flagging of embedded and module language SQL.

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

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

Where *level* can be either `ENTRY`, `INTERMEDIATE`, or `FULL`.

Figure 5–1 The Stages in Processing a SQL Statement



Grouping Operations into Transactions

In general, only application designers using the programming interfaces to Oracle Database are concerned with which types of actions should be grouped together as one transaction. Transactions must be defined properly so that 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 Transaction 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 *Oracle Database Concepts.*

- 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:

  - "How Serializable Transactions Interact" on page 5-21
  - *Oracle Database Concepts*

- Establish standards for writing SQL statements so that you can take advantage of shared SQL areas. Oracle Database recognizes identical SQL statements and allows them to share memory areas. This reduces memory usage on the database server and increases system throughput.

- Use the ANALYZE command to collect statistics that can be used by Oracle Database 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 in "Calling Stored Functions from SQL Expressions".
- 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.

See Also: "Using Cursors within Applications" on page 5-8

Committing Transactions

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 lets you 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';
```

Rolling Back Transactions

To roll back an entire transaction, or to roll back part of a transaction 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, use the TO option of the ROLLBACK command. For example, either of the following statements rolls back the current transaction to the savepoint named POINT1:

```
SAVEPOINT Point1;
...
ROLLBACK TO SAVEPOINT Point1;
ROLLBACK TO Point1;
```

Defining Transaction Savepoints

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 creating a savepoint, you can roll back to the savepoint.

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

An Example of COMMIT, SAVEPOINT, and ROLLBACK

Table 5–3 shows a series of SQL statements that illustrates the use of COMMIT, SAVEPOINT, and ROLLBACK statements within a transaction.

Table 5–1 Use of COMMIT, SAVEPOINT, and ROLLBACK

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

Table 5–1 (Cont.) Use of COMMIT, SAVEPOINT, and ROLLBACK

SQL Statement	Results
`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 were 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.

Ensuring Repeatable Reads with Read-Only Transactions

By default, the consistency model for Oracle Database guarantees statement-level read consistency, but does not guarantee transaction-level read consistency (repeatable reads). If you want transaction-level read consistency, and if your transaction does not require updates, then 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 change 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.

Long-running queries sometimes fail because undo information required for consistent read operations is no longer available. This happens when committed undo blocks are overwritten by active transactions. Automatic undo management provides a way to explicitly control when undo space can be reused; that is, how

long undo information is retained. Your database administrator can specify a retention period by using the parameter UNDO_RETENTION.

> **See Also:** *Oracle Database Administrator's Guide* for information on long-running queries and resumable space allocation

For example, if UNDO_RETENTION is set to 30 minutes, then all committed undo information in the system is retained for at least 30 minutes. This ensures that all queries running for 30 minutes or less, under usual circumstances, do not encounter the OER error, "snapshot too old."

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.

Using Cursors within Applications

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 Database users rely on the automatic cursor handling of the database 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 systemwide limit of cursors for each session is set by the initialization parameter named `OPEN_CURSORS` found in the parameter file (such as `INIT.ORA`).

> **See Also:** *Oracle Database Reference* for descriptions of parameters

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, then you can make sure you can open that many simultaneously.

Using a Cursor to Execute Statements Again

After each stage of execution, the cursor retains enough information about the SQL statement to reexecute the statement without starting over, as long as no other SQL statement has been associated with that cursor. This is illustrated in Figure 5–1. Notice that the statement can be reexecuted 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.

To understand the performance characteristics of a cursor, a DBA can retrieve the text of the query represented by the cursor using the `V$SQL` catalog view. Because the results of `EXPLAIN PLAN` on the original query might differ from the way the query is actually processed, a DBA can get more precise information by examining the `V$SQL_PLAN`, `V$SQL_PLAN_STATISTICS`, and `V$SQL_PLAN_STATISTICS_ALL` catalog views.:

- The `V$SQL_PLAN` view contains the execution plan information for each child cursor loaded in the library cache.
- The `V$SQL_PLAN_STATISTICS` view provides execution statistics at the row source level for each child cursor.

- The V$SQL_PLAN_STATISTICS_ALL view contains memory usage statistics for row sources that use SQL memory (sort or hash-join). This view concatenates information in V$SQL_PLAN with execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA.

  See Also: *Oracle Database Reference* for details on these views

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, then 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: *Oracle Call Interface Programmer's Guide* for more information about cancelling cursors

Locking Data Explicitly

Oracle Database always performs necessary locking to ensure data concurrency, integrity, and statement-level read consistency. You can override these default locking mechanisms. For example, you might want to override the default locking of Oracle Database if:

- You want transaction-level read consistency or "repeatable reads"—where transactions query a consistent set of data for the duration of the transaction, knowing that the data has not been changed by any other transactions. This level of consistency can be achieved by using explicit locking, read-only

transactions, serializable transactions, or overriding default locking for the system.

- A transaction requires 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 the transaction level. Transactions including the following SQL commands override Oracle Database's default locking:

- `LOCK TABLE`
- `SELECT`, including the `FOR UPDATE` clause
- `SET TRANSACTION` with the `READ ONLY` or `ISOLATION LEVEL SERIALIZABLE` options

Locks acquired by these statements are released after the transaction is committed or rolled back.

The following sections describe each option available for overriding the default locking of Oracle Database. The initialization parameter `DML_LOCKS` determines the maximum number of DML locks allowed.

> **See Also:** *Oracle Database Reference* for a discussion of parameters

Although the default value is usually enough, you might need to increase it if you use additional manual locks.

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

Choosing a Locking Strategy

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_TAB` and `DEPT_TAB` tables on behalf of the containing transaction:

```
LOCK TABLE Emp_tab, Dept_tab
    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 for each `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, then 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, then the transaction does not proceed until the requested table lock is acquired. If the wait for a table lock is excessive, then you might want to cancel the lock operation and retry at a later time; you can code this logic into your applications.

When to Lock with ROW SHARE and ROW EXCLUSIVE Mode

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

`ROW SHARE` and `ROW EXCLUSIVE` table locks offer the highest degree of concurrency. You might use these locks if:

- 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.

When to Lock with SHARE Mode

```
LOCK TABLE Emp_tab IN SHARE MODE;
```

`SHARE` table locks are rather restrictive data locks. You might use these locks if:

- Your transaction only queries the table, and requires a consistent set of the table data for the duration of the transaction.

- You can hold up other transactions that try to update the locked table, until all transactions that hold SHARE locks on the table either commit or roll back.
- Other transactions may acquire concurrent SHARE table locks on the same table, also allowing them the option of transaction-level read consistency.

> **Caution:** 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 are common. In this case, use share row exclusive or exclusive table locks instead.

For example, assume that two tables, EMP_TAB and BUDGET_TAB, require a consistent set of data in a third table, DEPT_TAB. 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_TAB table in SHARE MODE, as shown in the following example. Because the DEPT_TAB table is rarely updated, locking it probably does not cause many other transactions to wait long.

Note: You may need to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE dept_tab(
   deptno NUMBER(2) NOT NULL,
   dname VARCHAR2(14),
   loc VARCHAR2(13));

CREATE TABLE emp_tab (
   empno NUMBER(4) NOT NULL,
   ename VARCHAR2(10),
   job VARCHAR2(9),
   mgr NUMBER(4),
   hiredate DATE,
   sal NUMBER(7,2),
   comm NUMBER(7,2),
   deptno NUMBER(2));

CREATE TABLE Budget_tab (
   totsal NUMBER(7,2),
   deptno NUMBER(2) NOT NULL);
```

```
LOCK TABLE Dept_tab IN SHARE MODE;
UPDATE Emp_tab
    SET sal = sal * 1.1
    WHERE deptno IN
      (SELECT deptno FROM Dept_tab WHERE loc = 'DALLAS');
UPDATE Budget_tab
    SET Totsal = Totsal * 1.1
    WHERE Deptno IN
      (SELECT Deptno FROM Dept_tab WHERE Loc = 'DALLAS');

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

When to Lock with SHARE ROW EXCLUSIVE Mode

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

You might use a `SHARE ROW EXCLUSIVE` table lock if:

- Your transaction requires both transaction-level read consistency for the specified table and the ability to update the locked table.

- You do not care if other transactions acquire explicit row locks (using SELECT... FOR UPDATE), which might make UPDATE and INSERT statements in the locking transaction wait and might cause deadlocks.
- You only want a single transaction to have this behavior.

When to Lock in EXCLUSIVE Mode

```
LOCK TABLE Emp_tab IN EXCLUSIVE MODE;
```

You might use an EXCLUSIVE table if:

- Your transaction requires immediate update access to the locked table. When your transaction holds an exclusive table lock, other transactions cannot lock specific rows in the locked table.
- Your transaction also ensures 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. 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.

Letting Oracle Database Control Table Locking

Letting Oracle Database control table locking means your application needs less programming logic, but also has less control, than if you manage the table locks yourself.

Issuing the command SET TRANSACTION ISOLATION LEVEL SERIALIZABLE or ALTER SESSION ISOLATION LEVEL SERIALIZABLE preserves ANSI serializability without changing the underlying locking protocol. This technique allows concurrent access to the table while providing ANSI serializability. Getting table locks greatly reduces concurrency.

See Also:

- *Oracle Database SQL Reference* for information on the `SET TRANSACTION` statement
- *Oracle Database SQL Reference* for information on the `ALTER SESSION` statements

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

Explicitly Acquiring Row Locks

You can override default locking with a `SELECT` statement that includes the `FOR UPDATE` clause. This statement acquires exclusive row locks for selected rows (as an `UPDATE` statement does), in anticipation of updating the selected rows in a subsequent statement.

You can use a `SELECT... FOR UPDATE` statement to lock a row without actually changing it. For example, several triggers in Chapter 9, "Using Triggers", show how to implement referential integrity. In the `EMP_DEPT_CHECK` trigger (see "Foreign Key Trigger for Child Table"), 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 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 when the cursor is opened (before the first fetch) rather than being locked as they are fetched from the cursor. Locks are only released when the transaction that opened the cursor is committed or rolled back, not when the 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. If a `SELECT... FOR UPDATE` statement locks many rows in a table, and if the table experiences a lot of update activity, it might be faster to acquire an `EXCLUSIVE` table lock instead.

Note: The return set for a `SELECT... FOR UPDATE` may change while the query is running; for example, if columns selected by the query are updated or rows are deleted after the query started. When this happens, `SELECT... FOR UPDATE` acquires locks on the rows that did not change, gets a new read-consistent snapshot of the table using these locks, and then restarts the query to acquire the remaining locks.

This can cause a deadlock between sessions querying the table concurrently with DML operations when rows are locked in a non-sequential order. To prevent such deadlocks, design your application so that any concurrent DML on the table does not affect the return set of the query. If this is not feasible, you may want to serialize queries in your application.

When acquiring row locks with `SELECT... FOR UPDATE`, you can specify the `NOWAIT` option to indicate that you are not willing to wait to acquire the lock. If you cannot acquire then lock immediately, an error is returned to signal that the lock is not possible at this time. You can try to lock the row again later.

By default, the transaction waits until the requested row lock is acquired. If the wait for a row lock is too long, you can code logic into your application to cancel the lock operation and try again later.

About User Locks

You can use Oracle Lock Management services for your applications by making calls to the `DBMS_LOCK` package. 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 Database lock, it has all the features of a database lock, such as deadlock detection. Be certain that any user locks used in distributed transactions are released upon `COMMIT`, or an undetected deadlock can occur.

See Also: *PL/SQL Packages and Types Reference* for detailed information on the `DBMS_LOCK` package

When to Use User Locks

User locks can help to:

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

Example of a User Lock

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, more than $50 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

Table 5–4 describes Oracle Database facilities to display locking information for ongoing transactions within an instance.

Table 5–2 Ways to Display Locking Information

Tool	Description
Oracle Enterprise Manager 10*g* Database Control	From the Additional Monitoring Links section of the Database Performance page, click Database Locks to display user blocks, blocking locks, or the complete list of all database locks. Refer to *Oracle 2 Day DBA* for more information.
`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`.)

Using Serializable Transactions for Concurrency Control

By default, Oracle Database 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, because it provides higher concurrency and thus better performance. But some rare cases require 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. Concurrent transactions executing in serialized mode can only make

database changes that they could have made if the transactions ran one after the other.

Figure 5–2 shows a serializable transaction (B) interacting with another transaction (A).

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 5–3.

Table 5–3 Summary of 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

[1] A transaction can read uncommitted data changed by another transaction.

[2] A transaction rereads data committed by another transaction and sees the new data.

[3] A transaction can execute a query again, and discover new rows inserted by another committed transaction.

The behavior of Oracle Database with respect to these isolation levels is summarized in Table 5–4.

Table 5–4 ANSI Isolation Levels and Oracle Database

Isolation Level	Description
`READ UNCOMMITTED`	Oracle Database never permits "dirty reads." Although some other database products use this undesirable technique to improve thoughput, it is not required for high throughput with Oracle Database.

Table 5–4 (Cont.) ANSI Isolation Levels and Oracle Database

Isolation Level	Description
READ COMMITTED	Oracle Database meets the READ COMMITTED isolation standard. This is the default mode for all Oracle Database applications. Because an Oracle Database query only sees data that was committed at the beginning of the query (the snapshot time), Oracle Database actually offers more consistency than is required by the ANSI/ISO SQL92 standards for READ COMMITTED isolation.
REPEATABLE READ	Oracle Database does not normally support this isolation level, except as provided by SERIALIZABLE.
SERIALIZABLE	Oracle Database does not normally support this isolation level, except as provided by SERIALIZABLE.

How Serializable Transactions Interact

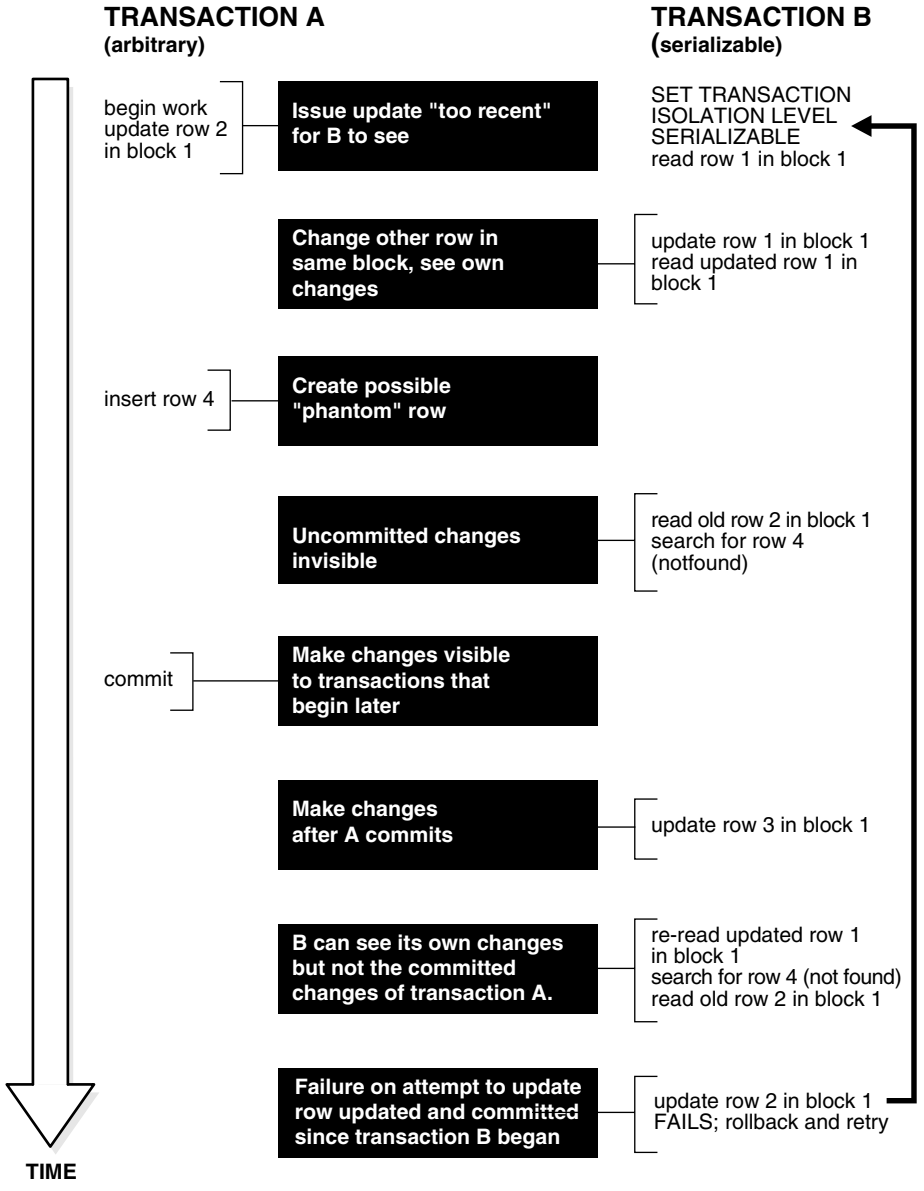
Figure 5–2 on page 5-22 shows how a serializable transaction (Transaction B) interacts with another transaction (A, which can be either SERIALIZABLE or READ COMMITTED).

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 Database 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.

Figure 5–2 Time Line for Two Transactions



Setting the Isolation Level of a Transaction

You can change the isolation level of a transaction using the `ISOLATION LEVEL` clause of the `SET TRANSACTION` command, which must be the first command issued in a transaction.

Use the `ALTER SESSION` command to set the transaction isolation level on a session-wide basis.

> **See Also:** *Oracle Database Reference* for the complete syntax of the `SET TRANSACTION` and `ALTER SESSION` commands

The INITRANS Parameter

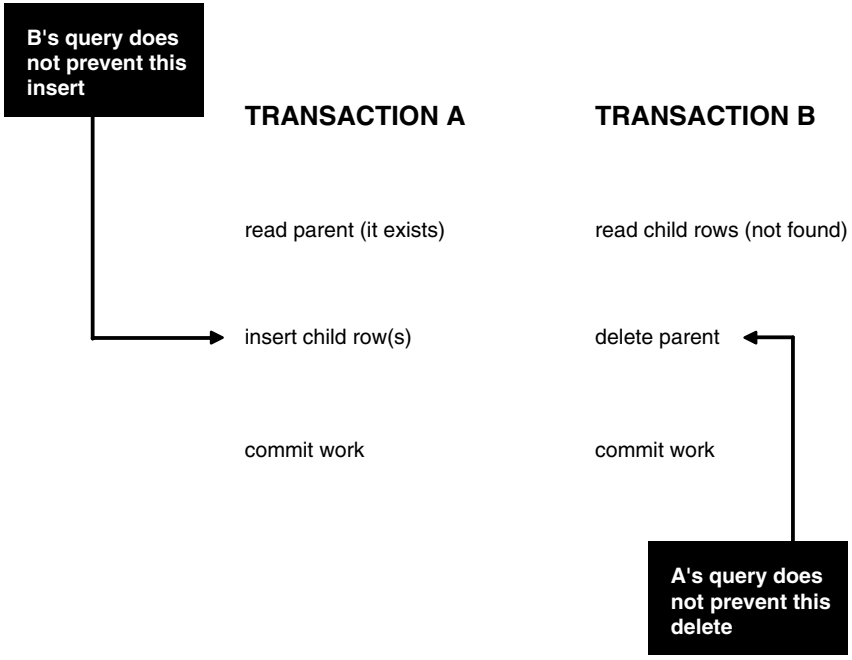
Oracle Database stores control information in each data block to manage access by concurrent transactions. Therefore, if you set the transaction isolation level to `SERIALIZABLE`, then you must use the `ALTER TABLE` command to set `INITRANS` to at least 3. This parameter causes Oracle Database to allocate 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 Database 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 5–3 on page 5-24 shows two different transactions that perform application-level checks to maintain the referential integrity parent/child relationship between two tables. One transaction checks that a row with a specific primary key value exists in the parent table before inserting corresponding child rows. The other transaction checks to see that no corresponding detail rows exist before deleting 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 5–3 Referential Integrity Check



The read issued by transaction A does not prevent transaction B from deleting the parent row, and transaction B's query for child rows does not prevent transaction A from inserting child rows. This scenario leaves a child row in the database with no corresponding parent row. This result occurs 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 shows, sometimes you must take steps to 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 Database to prevent the anomaly described:

- 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 Database using database triggers, instead of a separate query as in Transaction A. For example, an `INSERT` into the child table can fire a `BEFORE 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) remains 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 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 correctly enforces referential integrity.

READ COMMITTED and SERIALIZABLE Isolation

Oracle Database gives you 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 reduce contention, and are designed for deploying real-world applications. 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 Database is to consider:

- A collection of database tables (or any set of data)
- A sequence of reads of rows in those tables
- The set of transactions committed at any moment

An operation (a query or a transaction) is **transaction set consistent** if its read operations all return data written by the same set of committed transactions. When an operation is not transaction set consistent, some reads reflect the changes of one

set of transactions, and other reads reflect changes made by other transactions. Such an operation sees the database in a state that reflects no single set of committed transactions.

Oracle Database transactions executing in `READ COMMITTED` mode are transaction-set consistent on an individual-statement basis, because all rows read by a query must be committed before the query begins.

Oracle Database transactions executing in `SERIALIZABLE` mode are transaction set consistent on an individual-transaction basis, because all statements in a `SERIALIZABLE` transaction execute on an image of the database as of the beginning of the transaction.

In other database systems, 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. For example, 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 Database's `READ COMMITTED` mode avoids this problem, 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.

Comparison of READ COMMITTED and SERIALIZABLE Transactions

Table 5–5 summarizes key similarities and differences between `READ COMMITTED` and `SERIALIZABLE` transactions.

Table 5–5 Read Committed Versus Serializable Transaction

Operation	Read Committed	Serializable
Dirty write	Not Possible	Not Possible
Dirty read	Not Possible	Not Possible
Non-repeatable read	Possible	Not Possible
Phantoms	Possible	Not Possible

Table 5–5 (Cont.) Read Committed Versus Serializable Transaction

Operation	Read Committed	Serializable
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 for Transactions

Choose an isolation level that is appropriate to the specific application and workload. You might choose different isolation levels for different transactions. The choice depends on performance and consistency needs, and consideration of application coding requirements.

For environments with many concurrent users rapidly submitting transactions, you must assess transaction performance against the expected transaction arrival rate and response time demands, and choose an isolation level that provides the required degree of consistency while performing well. Frequently, for high performance environments, you must trade-off between consistency and concurrency (transaction throughput).

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

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

Application Tips for Transactions

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 causes an error:

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

When you get this error, roll back the current transaction and execute it again. The transaction gets a new transaction snapshot, and the operation is likely to succeed.

To minimize the performance overhead of rolling back transactions and executing them again, try to put DML statements that might conflict with other concurrent transactions near the beginning of your transaction.

Autonomous Transactions

This section gives a brief overview of autonomous transactions and what you can do with them.

> **See Also:** *PL/SQL User's Guide and Reference* and Chapter 9, "Using Triggers" for detailed information on autonomous transactions

At times, you may want to commit or roll back some changes to a table independently of a primary transaction's final outcome. For example, in a stock purchase transaction, you may want to commit a customer's information regardless of whether the overall stock purchase actually goes through. Or, while running that same transaction, you may want to log error messages to a debug table even if the overall transaction rolls back. Autonomous transactions allow you to do such tasks.

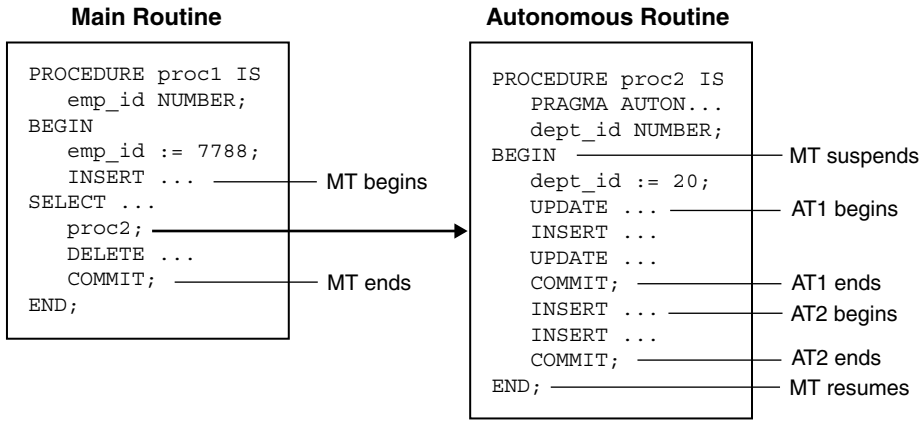
An **autonomous transaction** (AT) is an independent transaction started by another transaction, the **main transaction** (MT). It lets you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction.

An autonomous transaction executes within an **autonomous scope**. An autonomous scope is a routine you mark with the pragma (compiler directive) `AUTONOMOUS_TRANSACTION`. The pragma instructs the PL/SQL compiler to mark a routine as `autonomous` (independent). In this context, the term **routine** includes:

- Top-level (not nested) anonymous PL/SQL blocks
- Local, standalone, and packaged functions and procedures
- Methods of a SQL object type
- PL/SQL triggers

Figure 5–4 shows how control flows from the main routine (MT) to an autonomous routine (AT) and back again. As you can see, the autonomous routine can commit more than one transaction (AT1 and AT2) before control returns to the main routine.

Figure 5–4 Transaction Control Flow



When you enter the executable section of an autonomous routine, the main routine suspends. When you exit the routine, the main routine resumes. `COMMIT` and `ROLLBACK` end the active autonomous transaction but do not exit the autonomous routine. As Figure 5–4 shows, when one transaction ends, the next SQL statement begins another transaction.

A few more characteristics of autonomous transactions:

- The changes autonomous transactions effect do not depend on the state or the eventual disposition of the main transaction. For example:
  - An autonomous transaction does not see any changes made by the main transaction.
  - When an autonomous transaction commits or rolls back, it does not affect the outcome of the main transaction.
- The changes an autonomous transaction effects are visible to other transactions as soon as that autonomous transaction commits. This means that users can access the updated information without having to wait for the main transaction to commit.
- Autonomous transactions can start other autonomous transactions.

Figure 5–5 illustrates some of the possible sequences autonomous transactions can follow.

Figure 5–5 Possible Sequences of Autonomous Transactions

A main transaction scope (MT Scope) begins the main transaction, MTx. MTx invokes the first autonomous transaction scope (AT Scope1). MTx suspends. AT Scope 1 begins the transaction Tx1.1.

At Scope 1 commits or rolls back Tx1.1, than ends. MTx resumes.

MTx invokes AT Scope 2. MT suspends, passing control to AT Scope 2 which, initially, is performing queries.

AT Scope 2 then begins Tx2.1 by, say, doing an update. AT Scope 2 commits or rolls back Tx2.1.

Later, AT Scope 2 begins a second transaction, Tx2.2, then commits or rolls it back.

AT Scope 2 performs a few queries, then ends, passing control back to MTx.

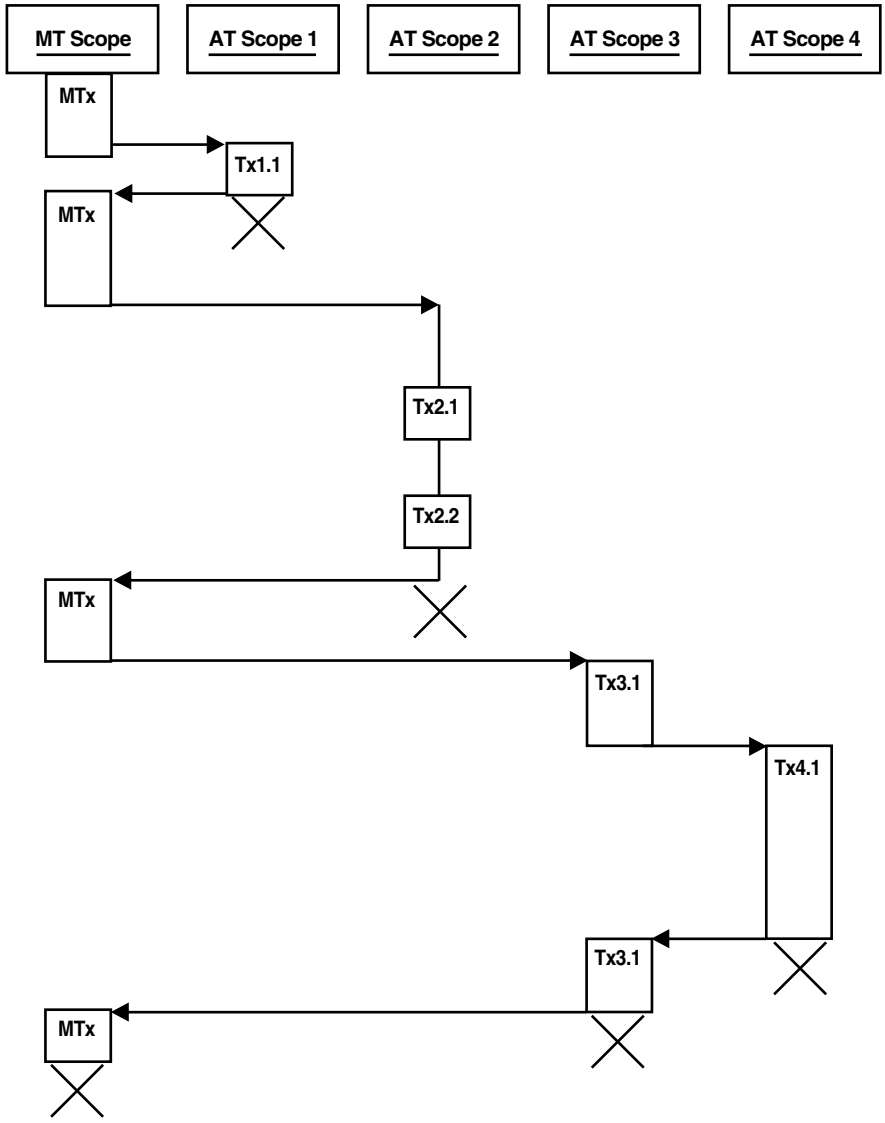
MTx invokes AT Scope 3. MTx suspends, AT Scope 3 begins.

AT Scope 3 begins Tx3.1 which, in turn, invokes AT Scope 4. Tx3.1 suspends, AT Scope 4 begins.

AT Scope 4 begins Tx4.1, commits or rolls it back, then ends. AT Scope 3 resumes.

AT Scope 3 commits or rolls back Tx3.1, then ends. MTx resumes.

Finally, MT Scope commits or rolls back MTx, then ends.



Examples of Autonomous Transactions

The two examples in this section illustrate some of the ways you can use autonomous transactions.

As these examples illustrate, there are four possible outcomes that can occur when you use autonomous and main transactions. Table 5–6 presents these possible outcomes. As you can see, there is no dependency between the outcome of an autonomous transaction and that of a main transaction.

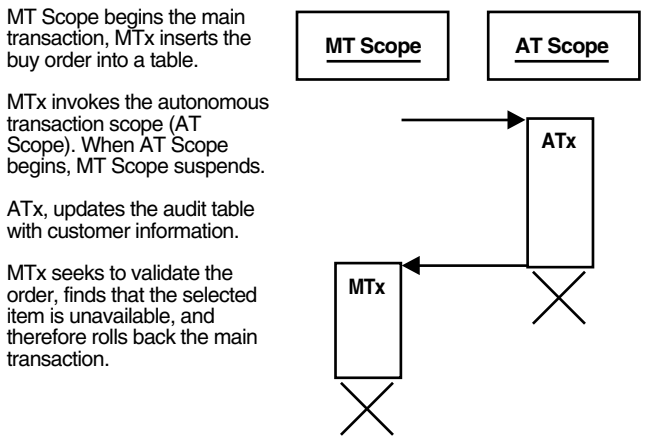
Table 5–6 Possible Transaction Outcomes

Autonomous Transaction	Main Transaction
Commits	Commits
Commits	Rolls back
Rolls back	Commits
Rolls back	Rolls back

Entering a Buy Order

In this example, illustrated by Figure 5–6, a customer enters a buy order. That customer's information (such as name, address, phone) is committed to a customer information table—even though the sale does not go through.

Figure 5–6 Example: A Buy Order



Example: Making a Bank Withdrawal

In this example, a customer tries to make a withdrawal from a bank account. In the process, a main transaction calls one of two autonomous transaction scopes (AT Scope 1, and AT Scope 2).

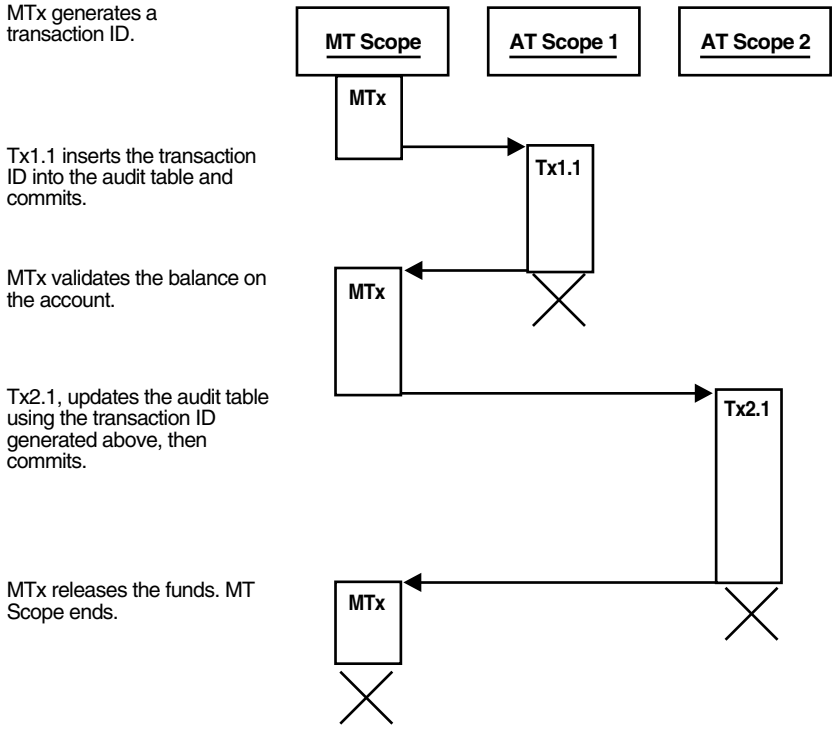
The following diagrams illustrate three possible scenarios for this transaction.

- Scenario 1: There are sufficient funds to cover the withdrawal and therefore the bank releases the funds
- Scenario 2: There are insufficient funds to cover the withdrawal, but the customer has overdraft protection. The bank therefore releases the funds.
- Scenario 3: There are insufficient funds to cover the withdrawal, the customer does not have overdraft protection, and the bank therefore withholds the requested funds.

Scenario 1

There are sufficient funds to cover the withdrawal and therefore the bank releases the funds. This is illustrated by Figure 5–7.

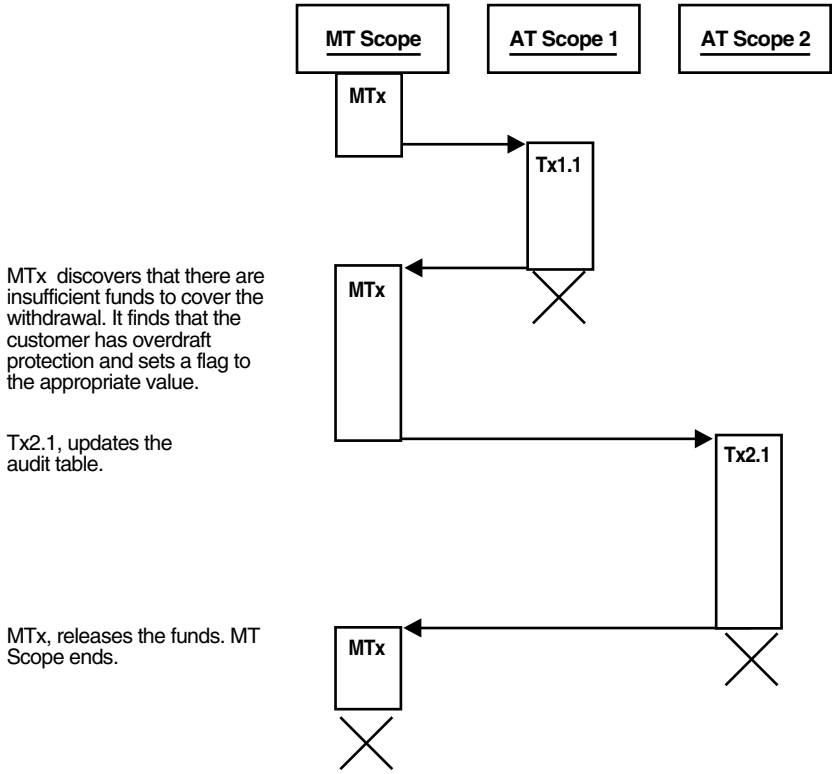
Figure 5–7 Example: Bank Withdrawal—Sufficient Funds



Scenario 2

There are insufficient funds to cover the withdrawal, but the customer has overdraft protection. The bank therefore releases the funds. This is illustrated by Figure 5–8.

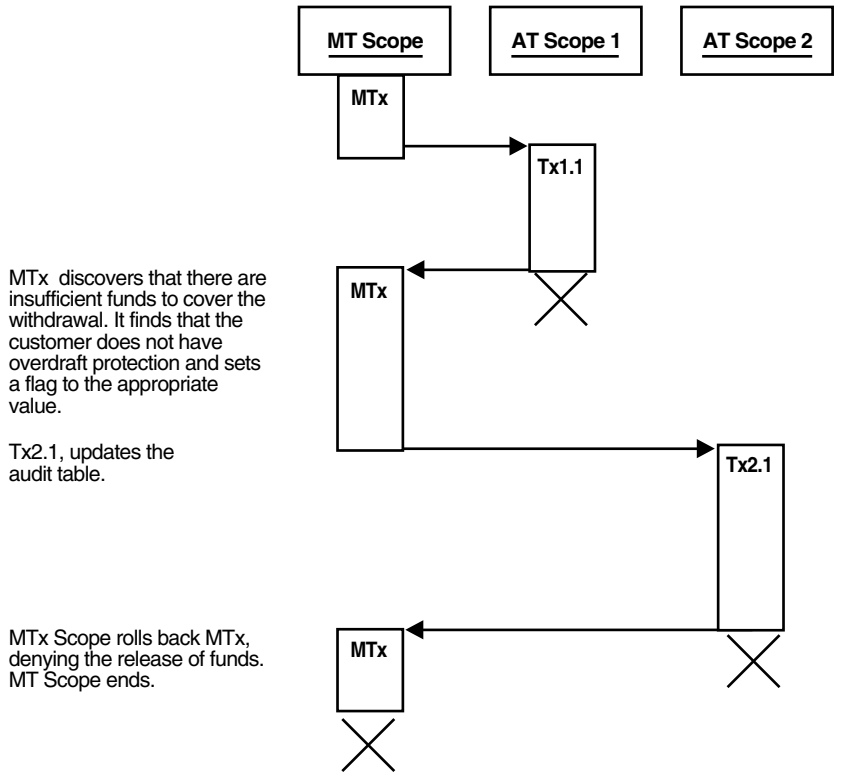
Figure 5–8 Example: Bank Withdrawal—Insufficient Funds WITH Overdraft Protection



Scenario 3

There are insufficient funds to cover the withdrawal, the customer does *not* have overdraft protection, and the bank therefore withholds the requested funds. This is illustrated by Figure 5–9.

Figure 5–9 Example: Bank Withdrawal—Insufficient Funds WITHOUT Overdraft Protection



Defining Autonomous Transactions

Note: This section is provided here to round out your *general* understanding of autonomous transactions. For a more thorough understanding of autonomous transactions, see *PL/SQL User's Guide and Reference.*

To define autonomous transactions, you use the pragma (compiler directive) AUTONOMOUS_TRANSACTION. The pragma instructs the PL/SQL compiler to mark the procedure, function, or PL/SQL block as autonomous (independent).

You can code the pragma anywhere in the declarative section of a procedure, function, or PL/SQL block. But, for readability, code the pragma at the top of the section. The syntax follows:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

In the following example, you mark a packaged function as autonomous:

```
CREATE OR REPLACE PACKAGE Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
    -- add additional functions and packages
END Banking;

CREATE OR REPLACE PACKAGE BODY Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        My_bal REAL;
    BEGIN
       --add appropriate code
    END;
    -- add additional functions and packages...
END Banking;
```

Restrictions on Autonomous Transactions

Note the following restrictions on autonomous transactions.

- You cannot use the pragma to mark all subprograms in a package (or all methods in an object type) as autonomous. Only individual routines can be marked autonomous. For example, the following pragma is illegal:

  ```
  CREATE OR REPLACE PACKAGE Banking AS
      PRAGMA AUTONOMOUS_TRANSACTION; -- illegal
      FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
      END Banking;
  ```

- You cannot execute a PIPE ROW statement in your autonomous routine while your autonomous transaction is open. You must close the autonomous transaction before executing the PIPE ROW statement. This is normally

accomplished by committing or rolling back the autonomous transaction before executing the `PIPE ROW` statement.

See Also: *PL/SQL User's Guide and Reference*

Resuming Execution After a Storage Error Condition

When a long-running transaction is interrupted by an out-of-space error condition, your application can suspend the statement that encountered the problem and resume it after the space problem is corrected. This capability is known as **resumable storage allocation**. It lets you avoid time-consuming rollbacks, without the need to split the operation into smaller pieces and write your own code to track its progress.

See Also:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*

What Operations Can Be Resumed After an Error Condition?

Queries, DML operations, and certain DDL operations can all be resumed if they encounter an out-of-space error. The capability applies if the operation is performed directly by a SQL statement, or if it is performed within a stored procedure, anonymous PL/SQL block, SQL*Loader, or an OCI call such as `OCIStmtExecute()`.

Operations can be resumed after these kinds of error conditions:

- Out of space errors, such as ORA-01653.
- Space limit errors, such as ORA-01628.
- Space quota errors, such as ORA-01536.

Limitations on Resuming Operations After an Error Condition

Certain storage errors *cannot* be handled using this technique. In dictionary-managed tablespaces, you cannot resume an operation if you run into the limit for rollback segments, or the maximum number of extents while creating an index or a table. Use locally managed tablespaces and automatic undo management in combination with this feature.

Writing an Application to Handle Suspended Storage Allocation

When an operation is suspended, your application does not receive the usual error code. Instead, perform any logging or notification by coding a trigger to detect the `AFTER SUSPEND` event and call the functions in the `DBMS_RESUMABLE` package to get information about the problem. Using this package, you can:

- Parse the error message with the `DBMS_RESUMABLE.SPACE_ERROR_INFO` function. For details about this function, see *PL/SQL Packages and Types Reference*.
- Set a new timeout value with the `SET_TIMEOUT` procedure.

Within the body of the trigger, you can perform any notifications, such as sending a mail message to alert an operator to the space problem.

Alternatively, the DBA can periodically check for suspended statements using the data dictionary views `DBA_RESUMABLE`, `USER_RESUMABLE`, and `V$_SESSION_WAIT`.

When the space condition is corrected (usually by the DBA), the suspended statement automatically resumes execution. If it is not corrected before the timeout period expires, the operation causes a `SERVERERROR` exception.

To reduce the chance of out-of-space errors within the trigger itself, you must declare it as an autonomous transaction so that it uses a rollback segment in the `SYSTEM` tablespace. If the trigger encounters a deadlock condition because of locks held by the suspended statement, the trigger is aborted and your application receives the original error condition, as if it was never suspended. If the trigger encounters an out-of-space condition, the trigger and the suspended statement are rolled back. You can prevent the rollback through an exception handler in the trigger, and just wait for the statement to be resumed.

> **See Also:** *Oracle Database Reference* for details on the `DBA_RESUMABLE`, `USER_RESUMABLE`, and `V$_SESSION_WAIT` data dictionary views

Example of Resumable Storage Allocation

This trigger handles applicable storage errors within the database. For some kinds of errors, it aborts the statement and alerts the DBA that this has happened through a mail message. For other errors that might be temporary, it specifies that the statement should wait for eight hours before resuming, with the expectation that the storage problem will be fixed by then.

```
CREATE OR REPLACE TRIGGER suspend_example
  AFTER SUSPEND
  ON DATABASE
  DECLARE
  cur_sid NUMBER;
  cur_inst NUMBER;
  err_type VARCHAR2(64);
  object_owner VARCHAR2(64);
  object_type VARCHAR2(64);
  table_space_name VARCHAR2(64);
  object_name VARCHAR2(64);
  sub_object_name VARCHAR2(64);
  msg_body VARCHAR2(64);
  ret_value boolean;
  error_txt varchar2(64);
  mail_conn utl_smtp.connection;
  BEGIN
  SELECT DISTINCT(sid) INTO cur_sid FROM v$mystat;
  cur_inst := userenv('instance');
  ret_value := dbms_resumable.space_error_info(err_type, object_owner, object_
type, table_space_name, object_name, sub_object_name);
  IF object_type = 'ROLLBACK SEGMENT' THEN
  INSERT INTO sys.rbs_error ( SELECT sql_text, error_msg, suspend_time FROM dba_
resumable WHERE session_id = cur_sid AND instance_id = cur_inst);
  SELECT error_msg into error_txt FROM dba_resumable WHERE session_id = cur_sid
AND instance_id = cur_inst;
  msg_body := 'Subject: Space error occurred: Space limit reached for rollback
segment  '|| object_name || ' on ' || to_char(SYSDATE, 'Month dd, YYYY,
HH:MIam') || '. Error message was: ' || error_txt;
  mail_conn := utl_smtp.open_connection('localhost', 25);
  utl_smtp.helo(mail_conn, 'localhost');
  utl_smtp.mail(mail_conn, 'sender@localhost');
  utl_smtp.rcpt(mail_conn, 'recipient@localhost');
  utl_smtp.data(mail_conn, msg_body);
  utl_smtp.quit(mail_conn);
  dbms_resumable.abort(cur_sid);
  ELSE
  dbms_resumable.set_timeout(3600*8);
  END IF;
  COMMIT;
  END;
```

6

Coding Dynamic SQL Statements

Dynamic SQL is a programming technique that enables you to build SQL statements dynamically at runtime. You can create more general purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation. For example, dynamic SQL lets you create a procedure that operates on a table whose name is not known until runtime.

Oracle Database includes two ways to implement dynamic SQL in a PL/SQL application:

- Native dynamic SQL, where you place dynamic SQL statements directly into PL/SQL blocks.
- Calling procedures in the `DBMS_SQL` package.

This chapter covers the following topics:

You can find details about the `DBMS_SQL` package in the *PL/SQL Packages and Types Reference*.

What Is Dynamic SQL?

Dynamic SQL enables you to write programs that reference SQL statements whose full text is not known until runtime. Before discussing dynamic SQL in detail, a clear definition of static SQL may provide a good starting point for understanding dynamic SQL. Static SQL statements do not change from execution to execution. The full text of static SQL statements are known at compilation, which provides the following benefits:

- Successful compilation verifies that the SQL statements reference valid database objects.
- Successful compilation verifies that the necessary privileges are in place to access the database objects.
- Performance of static SQL is generally better than dynamic SQL.

Because of these advantages, you should use dynamic SQL only if you cannot use static SQL to accomplish your goals, or if using static SQL is cumbersome compared to dynamic SQL. However, static SQL has limitations that can be overcome with dynamic SQL. You may not always know the full text of the SQL statements that must be executed in a PL/SQL procedure. Your program may accept user input that defines the SQL statements to execute, or your program may need to complete some processing work to determine the correct course of action. In such cases, you should use dynamic SQL.

For example, a reporting application in a data warehouse environment might not know the exact table name until runtime. These tables might be named according to the starting month and year of the quarter, for example `INV_01_1997`, `INV_04_1997`, `INV_07_1997`, `INV_10_1997`, `INV_01_1998`, and so on. You can use dynamic SQL in your reporting application to specify the table name at runtime.

You might also want to run a complex query with a user-selectable sort order. Instead of coding the query twice, with different `ORDER BY` clauses, you can construct the query dynamically to include a specified `ORDER BY` clause.

Dynamic SQL programs can handle changes in data definitions, without the need to recompile. This makes dynamic SQL much more flexible than static SQL. Dynamic SQL lets you write reusable code because the SQL can be easily adapted for different environments.

Dynamic SQL also lets you execute data definition language (DDL) statements and other SQL statements that are not supported in purely static SQL programs.

Why Use Dynamic SQL?

You should use dynamic SQL in cases where static SQL does not support the operation you want to perform, or in cases where you do not know the exact SQL statements that must be executed by a PL/SQL procedure. These SQL statements may depend on user input, or they may depend on processing work done by the program. The following sections describe typical situations where you should use dynamic SQL and typical problems that can be solved by using dynamic SQL

Executing DDL and SCL Statements in PL/SQL

In PL/SQL, you can only execute the following types of statements using dynamic SQL, rather than static SQL:

- Data definition language (DDL) statements, such as `CREATE`, `DROP`, `GRANT`, and `REVOKE`
- Session control language (SCL) statements, such as `ALTER SESSION` and `SET ROLE`

> **See Also:** *Oracle Database SQL Reference* for information about DDL and SCL statements

Also, you can only use the `TABLE` clause in the `SELECT` statement through dynamic SQL. For example, the following PL/SQL block contains a `SELECT` statement that uses the `TABLE` clause and native dynamic SQL:

```
CREATE TYPE t_emp AS OBJECT (id NUMBER, name VARCHAR2(20))
/
CREATE TYPE t_emplist AS TABLE OF t_emp
/

CREATE TABLE dept_new (id NUMBER, emps t_emplist)
    NESTED TABLE emps STORE AS emp_table;

INSERT INTO dept_new VALUES (
    10,
    t_emplist(
        t_emp(1, 'SCOTT'),
        t_emp(2, 'BRUCE')));

DECLARE
    deptid NUMBER;
    ename  VARCHAR2(20);
```

```
BEGIN
    EXECUTE IMMEDIATE 'SELECT d.id, e.name
        FROM dept_new d, TABLE(d.emps) e  -- not allowed in static SQL
                                          -- in PL/SQL
        WHERE e.id = 1'
        INTO deptid, ename;
END;
/
```

Executing Dynamic Queries

You can use dynamic SQL to create applications that execute dynamic queries, whose full text is not known until runtime. Many types of applications need to use dynamic queries, including:

- Applications that allow users to input or choose query search or sorting criteria at runtime
- Applications that allow users to input or choose optimizer hints at run time
- Applications that query a database where the data definitions of tables are constantly changing
- Applications that query a database where new tables are created often

For examples, see "Querying Using Dynamic SQL: Example" on page 6-17, and see the query examples in "A Dynamic SQL Scenario Using Native Dynamic SQL" on page 6-7.

Referencing Database Objects that Do Not Exist at Compilation

Many types of applications must interact with data that is generated periodically. For example, you might know the tables definitions at compile time, but not the names of the tables.

Dynamic SQL can solve this problem, because it lets you wait until runtime to specify the table names. For example, in the sample data warehouse application discussed in "What Is Dynamic SQL?" on page 6-2, new tables are generated every quarter, and these tables always have the same definition. You might let a user specify the name of the table at runtime with a dynamic SQL query similar to the following:

```
CREATE OR REPLACE PROCEDURE query_invoice(
       month VARCHAR2,
       year VARCHAR2) IS
    TYPE cur_typ IS REF CURSOR;
```

```
    c cur_typ;
    query_str VARCHAR2(200);
    inv_num NUMBER;
    inv_cust VARCHAR2(20);
    inv_amt NUMBER;
BEGIN
    query_str := 'SELECT num, cust, amt FROM inv_' || month ||'_'|| year
      || ' WHERE invnum = :id';
    OPEN c FOR query_str USING inv_num;
    LOOP
        FETCH c INTO inv_num, inv_cust, inv_amt;
        EXIT WHEN c%NOTFOUND;
        -- process row here
    END LOOP;
    CLOSE c;
END;
/
```

Optimizing Execution Dynamically

You can use dynamic SQL to build a SQL statement in a way that optimizes the execution by concatenating the hints into a SQL statement dynamically. This lets you change the hints based on your current database statistics, without requiring recompilation.

For example, the following procedure uses a variable called `a_hint` to allow users to pass a hint option to the `SELECT` statement:

```
CREATE OR REPLACE PROCEDURE query_emp
      (a_hint VARCHAR2) AS
   TYPE cur_typ IS REF CURSOR;
   c cur_typ;
BEGIN
   OPEN c FOR 'SELECT ' || a_hint ||
      ' empno, ename, sal, job FROM emp WHERE empno = 7566';
      -- process
END;
/
```

In this example, the user can pass any of the following values for `a_hint`:

```
a_hint = '/*+ ALL_ROWS */'
a_hint = '/*+ FIRST_ROWS */'
a_hint = '/*+ CHOOSE */'
```

or any other valid hint option.

> **See Also:** *Oracle Database Performance Tuning Guide* for more information about using hints

Executing Dynamic PL/SQL Blocks

You can use the `EXECUTE IMMEDIATE` statement to execute anonymous PL/SQL blocks. You can add flexibility by constructing the block contents at runtime.

For example, suppose you want to write an application that takes an event number and dispatches to a handler for the event. The name of the handler is in the form `EVENT_HANDLER_`*`event_num`*, where *`event_num`* is the number of the event. One approach is to implement the dispatcher as a switch statement, where the code handles each event by making a static call to its appropriate handler. This code is not very extensible because the dispatcher code must be updated whenever a handler for a new event is added.

```
CREATE OR REPLACE PROCEDURE event_handler_1(param number) AS BEGIN
   -- process event
   RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_handler_2(param number) AS BEGIN
   -- process event
   RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_handler_3(param number) AS BEGIN
   -- process event
   RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_dispatcher
   (event number, param number) IS
BEGIN
  IF (event = 1) THEN
    EVENT_HANDLER_1(param);
  ELSIF (event = 2) THEN
    EVENT_HANDLER_2(param);
  ELSIF (event = 3) THEN
    EVENT_HANDLER_3(param);
```

```
  END IF;
END;
/
```

Using native dynamic SQL, you can write a smaller, more flexible event dispatcher similar to the following:

```
CREATE OR REPLACE PROCEDURE event_dispatcher
   (event NUMBER, param NUMBER) IS
BEGIN
  EXECUTE IMMEDIATE
     'BEGIN
        EVENT_HANDLER_' || to_char(event) || '(:1);
      END;'
  USING param;
END;
/
```

Performing Dynamic Operations Using Invoker's Rights

By using the invoker's rights feature with dynamic SQL, you can build applications that issue dynamic SQL statements under the privileges and schema of the invoker. These two features—invoker's rights and dynamic SQL—enable you to build reusable application subcomponents that can operate on and access the invoker's data and modules.

> **See Also:** *PL/SQL User's Guide and Reference* for information about using invoker's rights and native dynamic SQL

A Dynamic SQL Scenario Using Native Dynamic SQL

This scenario shows you how to perform the following operations using native dynamic SQL:

- Execute DDL and DML operations
- Execute single row and multiple row queries

The database in this scenario is a company's human resources database (named `hr`) with the following data model:

A master table named `offices` contains the list of all company locations. The `offices` table has the following definition:

```
Column Name             Null?           Type
LOCATION                NOT_NULL        VARCHAR2(200)
```

Multiple `emp_location` tables contain the employee information, where *`location`* is the name of city where the office is located. For example, a table named `emp_houston` contains employee information for the company's Houston office, while a table named `emp_boston` contains employee information for the company's Boston office.

Each `emp_location` table has the following definition:

```
Column Name             Null?           Type
EMPNO                   NOT_NULL        NUMBER(4)
ENAME                   NOT_NULL        VARCHAR2(10)
JOB                     NOT_NULL        VARCHAR2(9)
SAL                     NOT_NULL        NUMBER(7,2)
DEPTNO                  NOT_NULL        NUMBER(2)
```

The following sections describe various native dynamic SQL operations that can be performed on the data in the `hr` database.

Sample DML Operation Using Native Dynamic SQL

The following native dynamic SQL procedure gives a raise to all employees with a particular job title:

```
CREATE OR REPLACE PROCEDURE salary_raise (raise_percent NUMBER, job VARCHAR2) IS
    TYPE loc_array_type IS TABLE OF VARCHAR2(40)
        INDEX BY binary_integer;
    dml_str VARCHAR2(200);
    loc_array loc_array_type;
BEGIN
    -- bulk fetch the list of office locations
    SELECT location BULK COLLECT INTO loc_array
        FROM offices;
    -- for each location, give a raise to employees with the given 'job'
    FOR i IN loc_array.first..loc_array.last LOOP
        dml_str := 'UPDATE emp_' || loc_array(i)
        || ' SET sal = sal * (1+(:raise_percent/100))'
        || ' WHERE job = :job_title';
    EXECUTE IMMEDIATE dml_str USING raise_percent, job;
    END LOOP;
END;
/
SHOW ERRORS;
```

Sample DDL Operation Using Native Dynamic SQL

The EXECUTE IMMEDIATE statement can perform DDL operations. For example, the following procedure adds an office location:

```
CREATE OR REPLACE PROCEDURE add_location (loc VARCHAR2) IS
BEGIN
    -- insert new location in master table
    INSERT INTO offices VALUES (loc);
    -- create an employee information table
    EXECUTE IMMEDIATE
    'CREATE TABLE ' || 'emp_' || loc ||
    '(
        empno   NUMBER(4) NOT NULL,
        ename   VARCHAR2(10),
        job     VARCHAR2(9),
        sal     NUMBER(7,2),
        deptno  NUMBER(2)
    )';
END;
/
SHOW ERRORS;
```

The following procedure deletes an office location:

```
CREATE OR REPLACE PROCEDURE drop_location (loc VARCHAR2) IS
BEGIN
    -- delete the employee table for location 'loc'
    EXECUTE IMMEDIATE 'DROP TABLE ' || 'emp_' || loc;
    -- remove location from master table
    DELETE FROM offices WHERE location = loc;
END;
/
SHOW ERRORS;
```

Sample Single-Row Query Using Native Dynamic SQL

The EXECUTE IMMEDIATE statement can perform dynamic single-row queries. You can specify bind variables in the USING clause and fetch the resulting row into the target specified in the INTO clause of the statement.

The following function retrieves the number of employees at a particular location performing a specified job:

```
CREATE OR REPLACE FUNCTION get_num_of_employees (loc VARCHAR2, job VARCHAR2)
```

```
    RETURN NUMBER IS
    query_str VARCHAR2(1000);
    num_of_employees NUMBER;
BEGIN
    query_str := 'SELECT COUNT(*) FROM '
        || ' emp_' || loc
        || ' WHERE job = :job_title';
    EXECUTE IMMEDIATE query_str
        INTO num_of_employees
        USING job;
    RETURN num_of_employees;
END;
/
SHOW ERRORS;
```

Sample Multiple-Row Query Using Native Dynamic SQL

The `OPEN-FOR`, `FETCH`, and `CLOSE` statements can perform dynamic multiple-row queries. For example, the following procedure lists all of the employees with a particular job at a specified location:

```
CREATE OR REPLACE PROCEDURE list_employees(loc VARCHAR2, job VARCHAR2) IS
    TYPE cur_typ IS REF CURSOR;
    c           cur_typ;
    query_str   VARCHAR2(1000);
    emp_name    VARCHAR2(20);
    emp_num     NUMBER;
BEGIN
    query_str := 'SELECT ename, empno FROM emp_' || loc
        || ' WHERE job = :job_title';
    -- find employees who perform the specified job
    OPEN c FOR query_str USING job;
    LOOP
        FETCH c INTO emp_name, emp_num;
        EXIT WHEN c%NOTFOUND;
        -- process row here
    END LOOP;
    CLOSE c;
END;
/
SHOW ERRORS;
```

Choosing Between Native Dynamic SQL and the DBMS_SQL Package

Oracle Database provides two methods for using dynamic SQL within PL/SQL: native dynamic SQL and the DBMS_SQL package. Native dynamic SQL lets you place dynamic SQL statements directly into PL/SQL code. These dynamic statements include DML statements (including queries), PL/SQL anonymous blocks, DDL statements, transaction control statements, and session control statements.

To process most native dynamic SQL statements, you use the EXECUTE IMMEDIATE statement. To process a multi-row query (SELECT statement), you use OPEN-FOR, FETCH, and CLOSE statements.

Note: To use native dynamic SQL, the COMPATIBLE initialization parameter must be set to 8.1.0 or higher. See *Oracle Database Upgrade Guide* for more information about the COMPATIBLE parameter.

The DBMS_SQL package is a PL/SQL library that offers an API to execute SQL statements dynamically. The DBMS_SQL package has procedures to open a cursor, parse a cursor, supply binds, and so on. Programs that use the DBMS_SQL package make calls to this package to perform dynamic SQL operations.

The following sections provide detailed information about the advantages of both methods.

See Also:

- The *PL/SQL User's Guide and Reference* for detailed information about using native dynamic SQL. Native dynamic SQL is referred to in that book as "dynamic SQL".
- *PL/SQL Packages and Types Reference* for detailed information about using the DBMS_SQL package

Advantages of Native Dynamic SQL

Native dynamic SQL provides the following advantages over the DBMS_SQL package:

Native Dynamic SQL is Easy to Use

Because native dynamic SQL is integrated with SQL, you can use it in the same way that you use static SQL within PL/SQL code. Native dynamic SQL code is typically more compact and readable than equivalent code that uses the `DBMS_SQL` package.

With the `DBMS_SQL` package you must call many procedures and functions in a strict sequence, making even simple operations require a lot of code. You can avoid this complexity by using native dynamic SQL instead.

Table 6–1 illustrates the difference in the amount of code required to perform the same operation using the `DBMS_SQL` package and native dynamic SQL.

Table 6–1 Code Comparison of DBMS_SQL Package and Native Dynamic SQL

DBMS_SQL Package

```
CREATE PROCEDURE insert_into_table (
      table_name  VARCHAR2,
      deptnumber  NUMBER,
      deptname    VARCHAR2,
      location    VARCHAR2) IS
   cur_hdl         INTEGER;
   stmt_str        VARCHAR2(200);
   rows_processed  BINARY_INTEGER;

BEGIN
   stmt_str := 'INSERT INTO ' ||
      table_name || ' VALUES
      (:deptno, :dname, :loc)';

   -- open cursor
   cur_hdl := dbms_sql.open_cursor;

   -- parse cursor
   dbms_sql.parse(cur_hdl, stmt_str,
      dbms_sql.native);

   -- supply binds
   dbms_sql.bind_variable
      (cur_hdl, ':deptno', deptnumber);
   dbms_sql.bind_variable
      (cur_hdl, ':dname', deptname);
   dbms_sql.bind_variable
      (cur_hdl, ':loc', location);

    -- execute cursor
    rows_processed :=
    dbms_sql.execute(cur_hdl);

    -- close cursor
    dbms_sql.close_cursor(cur_hdl);

END;
/
SHOW ERRORS;
```

Native Dynamic SQL

```
CREATE PROCEDURE insert_into_table (
      table_name  VARCHAR2,
      deptnumber  NUMBER,
      deptname    VARCHAR2,
      location    VARCHAR2) IS
   stmt_str    VARCHAR2(200);

BEGIN
   stmt_str := 'INSERT INTO ' ||
      table_name || ' values
      (:deptno, :dname, :loc)';

   EXECUTE IMMEDIATE stmt_str
      USING
      deptnumber, deptname, location;

END;
/
SHOW ERRORS;
```

Native Dynamic SQL is Faster than DBMS_SQL

Native dynamic SQL in PL/SQL performs comparably to the performance of static SQL, because the PL/SQL interpreter has built-in support for it. Programs that use native dynamic SQL are much faster than programs that use the `DBMS_SQL` package. Typically, native dynamic SQL statements perform 1.5 to 3 times better than equivalent `DBMS_SQL` calls. (Your performance gains may vary depending on your application.)

Native dynamic SQL bundles the statement preparation, binding, and execution steps into a single operation, which minimizes the data copying and procedure call overhead and improves performance.

The `DBMS_SQL` package is based on a procedural API and incurs high procedure call and data copy overhead. Each time you bind a variable, the `DBMS_SQL` package copies the PL/SQL bind variable into its space for use during execution. Each time you execute a fetch, the data is copied into the space managed by the `DBMS_SQL` package and then the fetched data is copied, one column at a time, into the appropriate PL/SQL variables, resulting in substantial overhead.

Performance Tip: Using Bind Variables When using either native dynamic SQL or the `DBMS_SQL` package, you can improve performance by using bind variables, because bind variables allow Oracle Database to share a single cursor for multiple SQL statements.

For example, the following native dynamic SQL code does not use bind variables:

```
CREATE OR REPLACE PROCEDURE del_dept (
   my_deptno  dept.deptno%TYPE) IS
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = ' || to_char (my_deptno);
END;
/
SHOW ERRORS;
```

For each distinct `my_deptno` variable, a new cursor is created, causing resource contention and poor performance. Instead, bind `my_deptno` as a bind variable:

```
CREATE OR REPLACE PROCEDURE del_dept (
   my_deptno  dept.deptno%TYPE) IS
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :1' USING my_deptno;
END;
/
SHOW ERRORS;
```

Here, the same cursor is reused for different values of the bind `my_deptno`, improving performance and scalabilty.

Native Dynamic SQL Supports User-Defined Types

Native dynamic SQL supports all of the types supported by static SQL in PL/SQL, including user-defined types such as user-defined objects, collections, and `REF`s. The `DBMS_SQL` package does not support these user-defined types.

> **Note:** The `DBMS_SQL` package provides limited support for arrays. See the *PL/SQL Packages and Types Reference* for information.

Native Dynamic SQL Supports Fetching Into Records

Native dynamic SQL and static SQL both support fetching into records, but the `DBMS_SQL` package does not. With native dynamic SQL, the rows resulting from a query can be directly fetched into PL/SQL records.

In the following example, the rows from a query are fetched into the `emp_rec` record:

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    c EmpCurTyp;
    emp_rec emp%ROWTYPE;
    stmt_str VARCHAR2(200);
    e_job emp.job%TYPE;

BEGIN
   stmt_str := 'SELECT * FROM emp WHERE job = :1';
    -- in a multi-row query
    OPEN c FOR stmt_str USING 'MANAGER';
    LOOP
        FETCH c INTO emp_rec;
        EXIT WHEN c%NOTFOUND;
    END LOOP;
    CLOSE c;
    -- in a single-row query
    EXECUTE IMMEDIATE stmt_str INTO emp_rec USING 'PRESIDENT';

END;
/
```

Advantages of the DBMS_SQL Package

The DBMS_SQL package provides the following advantages over native dynamic SQL:

DBMS_SQL is Supported in Client-Side Programs

The DBMS_SQL package is supported in client-side programs, but native dynamic SQL is not. Every call to the DBMS_SQL package from the client-side program translates to a PL/SQL remote procedure call (RPC); these calls occur when you need to bind a variable, define a variable, or execute a statement.

DBMS_SQL Supports DESCRIBE

The DESCRIBE_COLUMNS procedure in the DBMS_SQL package can be used to describe the columns for a cursor opened and parsed through DBMS_SQL. This feature is similar to the DESCRIBE command in SQL*Plus. Native dynamic SQL does not have a DESCRIBE facility.

See Also:

- *PL/SQL Packages and Types Reference* for an example of using DESCRIBE_COLUMNS to create a query in a situation where the SELECT list is not known until runtime

DBMS_SQL Supports SQL Statements Larger than 32KB

The DBMS_SQL package supports SQL statements larger than 32KB; native dynamic SQL does not.

DBMS_SQL Lets You Reuse SQL Statements

The PARSE procedure in the DBMS_SQL package parses a SQL statement once. After the initial parsing, you can use the statement multiple times with different sets of bind arguments.

Native dynamic SQL prepares a SQL statement each time the statement is used, which typically involves parsing, optimization, and plan generation. Although the extra prepare operations incur a small performance penalty, the slowdown is typically outweighed by the performance benefits of native dynamic SQL.

Examples of DBMS_SQL Package Code and Native Dynamic SQL Code

The following examples illustrate the differences in the code necessary to complete operations with the DBMS_SQL package and native dynamic SQL. Specifically, the following types of examples are presented:

- A query
- A DML operation
- A DML returning operation

In general, the native dynamic SQL code is more readable and compact, which can improve developer productivity.

Querying Using Dynamic SQL: Example

The following example includes a dynamic query statement with one bind variable (:jobname) and two select columns (ename and sal):

```
stmt_str := 'SELECT ename, sal FROM emp WHERE job = :jobname';
```

This example queries for employees with the job description SALESMAN in the job column of the emp table. Table 6–2 shows sample code that accomplishes this query using the DBMS_SQL package and native dynamic SQL.

Table 6–2 Querying Using the DBMS_SQL Package and Native Dynamic SQL

DBMS_SQL Query Operation	Native Dynamic SQL Query Operation

DECLARE
  stmt_str varchar2(200);
  cur_hdl int;
  rows_processed int;
  name varchar2(10);
  salary int;
BEGIN
cur_hdl := dbms_sql.open_cursor; -- open cursor
stmt_str := 'SELECT ename, sal FROM emp WHERE
job = :jobname';
dbms_sql.parse(cur_hdl, stmt_str, dbms_
sql.native);
 	
-- supply binds (bind by name)
dbms_sql.bind_variable(
  cur_hdl, 'jobname', 'SALESMAN');
 	
-- describe defines
dbms_sql.define_column(cur_hdl, 1, name, 200);
dbms_sql.define_column(cur_hdl, 2, salary);
 	
rows_processed := dbms_sql.execute(cur_hdl); --
execute
 	
LOOP
  -- fetch a row
  IF dbms_sql.fetch_rows(cur_hdl) > 0 then
 	
  -- fetch columns from the row
  dbms_sql.column_value(cur_hdl, 1, name);
  dbms_sql.column_value(cur_hdl, 2, salary);
 	
  -- <process data>
 	
  ELSE
  EXIT;
  END IF;
END LOOP;
dbms_sql.close_cursor(cur_hdl); -- close cursor
END;
/
	DECLARE
	  TYPE EmpCurTyp IS REF CURSOR;
	  cur EmpCurTyp;
	  stmt_str VARCHAR2(200);
	  name VARCHAR2(20);
	  salary NUMBER;
	BEGIN
	  stmt_str := 'SELECT ename, sal FROM emp
	  WHERE job = :1';
	  OPEN cur FOR stmt_str USING 'SALESMAN';
	
	LOOP
	  FETCH cur INTO name, salary;
	  EXIT WHEN cur%NOTFOUND;
	  -- <process data>
	END LOOP;
	CLOSE cur;
	END;
	/

Performing DML Using Dynamic SQL: Example

The following example includes a dynamic `INSERT` statement for a table with three columns:

```
stmt_str := 'INSERT INTO dept_new VALUES (:deptno, :dname, :loc)';
```

This example inserts a new row for which the column values are in the PL/SQL variables `deptnumber`, `deptname`, and `location`. Table 6–3 shows sample code that accomplishes this DML operation using the `DBMS_SQL` package and native dynamic SQL.

Table 6–3 DML Operation Using the DBMS_SQL Package and Native Dynamic SQL

DBMS_SQL DML Operation	Native Dynamic SQL DML Operation

DECLARE
  stmt_str VARCHAR2(200);
  cur_hdl NUMBER;
  deptnumber NUMBER := 99;
  deptname VARCHAR2(20);
  location VARCHAR2(10);
  rows_processed NUMBER;
BEGIN
  stmt_str := 'INSERT INTO dept_new VALUES
  (:deptno, :dname, :loc)';
  cur_hdl := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(
  cur_hdl, stmt_str, DBMS_SQL.NATIVE);
  -- supply binds
  DBMS_SQL.BIND_VARIABLE
  (cur_hdl, ':deptno', deptnumber);
  DBMS_SQL.BIND_VARIABLE
  (cur_hdl, ':dname', deptname);
  DBMS_SQL.BIND_VARIABLE
  (cur_hdl, ':loc', location);
  rows_processed := dbms_sql.execute(cur_hdl);
  -- execute
  DBMS_SQL.CLOSE_CURSOR(cur_hdl); -- close
END;
/
	DECLARE
	  stmt_str VARCHAR2(200);
	  deptnumber NUMBER := 99;
	  deptname VARCHAR2(20);
	  location VARCHAR2(10);
	BEGIN
	  stmt_str := 'INSERT INTO dept_new VALUES
	  (:deptno, :dname, :loc)';
	  EXECUTE IMMEDIATE stmt_str
	  USING deptnumber, deptname, location;
	END;
	/

Performing DML with RETURNING Clause Using Dynamic SQL: Example

The following example uses a dynamic `UPDATE` statement to update the location of a department, then returns the name of the department:

```
stmt_str := 'UPDATE dept_new
            SET loc = :newloc
```

```
    WHERE deptno = :deptno
    RETURNING dname INTO :dname';
```

Table 6–4 shows sample code that accomplishes this operation using both the DBMS_SQL package and native dynamic SQL.

Table 6–4 DML Returning Operation Using the DBMS_SQL Package and Native Dynamic SQL

DBMS_SQL DML Returning Operation	Native Dynamic SQL DML Returning Operation

DECLARE
  deptname_array dbms_sql.Varchar2_Table;
  cur_hdl INT;
  stmt_str VARCHAR2(200);
  location VARCHAR2(20);
  deptnumber NUMBER := 10;
  rows_processed NUMBER;
BEGIN
 stmt_str := 'UPDATE dept_new
  SET loc = :newloc
  WHERE deptno = :deptno
  RETURNING dname INTO :dname';
 	
  cur_hdl := dbms_sql.open_cursor;
  dbms_sql.parse
  (cur_hdl, stmt_str, dbms_sql.native);
  -- supply binds
  dbms_sql.bind_variable
  (cur_hdl, ':newloc', location);
  dbms_sql.bind_variable
  (cur_hdl, ':deptno', deptnumber);
  dbms_sql.bind_array
  (cur_hdl, ':dname', deptname_array);
  -- execute cursor
  rows_processed := dbms_sql.execute(cur_hdl);
  -- get RETURNING column into OUT bind array
  dbms_sql.variable_value
  (cur_hdl, ':dname', deptname_array);
  dbms_sql.close_cursor(cur_hdl);
END;
/
	DECLARE
	  deptname_array dbms_sql.Varchar2_Table;
	  stmt_str VARCHAR2(200);
	  location VARCHAR2(20);
	  deptnumber NUMBER := 10;
	  deptname VARCHAR2(20);
	BEGIN
	  stmt_str := 'UPDATE dept_new
	  SET loc = :newloc
	  WHERE deptno = :deptno
	  RETURNING dname INTO :dname';
	  EXECUTE IMMEDIATE stmt_str
	  USING location, deptnumber, OUT deptname;
	END;
	/

Using Dynamic SQL in Languages Other Than PL/SQL

Although this chapter discusses PL/SQL support for dynamic SQL, you can call dynamic SQL from other languages:

- If you use C/C++, you can call dynamic SQL with the Oracle Call Interface (OCI), or you can use the Pro*C/C++ precompiler to add dynamic SQL extensions to your C code.
- If you use COBOL, you can use the Pro*COBOL precompiler to add dynamic SQL extensions to your COBOL code.
- If you use Java, you can develop applications that use dynamic SQL with JDBC.

If you have an application that uses OCI, Pro*C/C++, or Pro*COBOL to execute dynamic SQL, you should consider switching to native dynamic SQL inside PL/SQL stored procedures and functions. The network round-trips required to perform dynamic SQL operations from client-side applications might hurt performance. Stored procedures can reside on the server, eliminating the network overhead. You can call the PL/SQL stored procedures and stored functions from the OCI, Pro*C/C++, or Pro*COBOL application.

> **See Also:** For information about calling Oracle Database stored procedures and stored functions from various languages:
>
> - *Oracle Call Interface Programmer's Guide*
> - *Pro*C/C++ Programmer's Guide*
> - *Pro*COBOL Programmer's Guide*

7

Using Procedures and Packages

This chapter describes some of the procedural capabilities of Oracle Database for application development, including:

- Overview of PL/SQL Program Units
- Hiding PL/SQL Code with the PL/SQL Wrapper
- Compiling PL/SQL Procedures for Native Execution
- Remote Dependencies
- Cursor Variables
- Handling PL/SQL Compile-Time Errors
- Handling Run-Time PL/SQL Errors
- Debugging Stored Procedures
- Calling Stored Procedures
- Calling Remote Procedures
- Calling Stored Functions from SQL Expressions
- Returning Large Amounts of Data from a Function
- Coding Your Own Aggregate Functions

Overview of PL/SQL Program Units

PL/SQL is a modern, block-structured programming language. It provides several features that make developing powerful database applications very convenient. For example, PL/SQL provides procedural constructs, such as loops and conditional statements, that are not available in standard SQL.

You can directly enter 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 runs on the server, so using PL/SQL lets you 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 that lets you run PL/SQL locally.

You can even use PL/SQL for some database applications in place of 3GL programs that use embedded SQL or Oracle Call Interface (OCI).

PL/SQL program units include:

- Anonymous Blocks
- Stored Program Units (Procedures, Functions, and Packages)
- Triggers

See Also:

- *PL/SQL User's Guide and Reference* for syntax and examples of operations on PL/SQL packages
- *PL/SQL Packages and Types Reference* for information about the PL/SQL packages that come with Oracle Database

Anonymous Blocks

An anonymous block is a PL/SQL program unit that has no name and it does not require the explicit presence of the `BEGIN` and `END` keywords to enclose the executable statements. An anonymous block consists of an optional *declarative* part, an *executable* part, and one or more optional *exception handlers*.

The declarative part declares 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_tab table, using the DBMS_OUTPUT package:

```
DECLARE
   Emp_name    VARCHAR2(10);
   Cursor      c1 IS SELECT Ename FROM Emp_tab
                  WHERE Deptno = 20;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO Emp_name;
      EXIT WHEN c1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(Emp_name);
   END LOOP;
END;
```

Note: If you test this block using SQL*Plus, then enter the statement SET SERVEROUTPUT ON, so that output using the DBMS_OUTPUT procedures (for example, PUT_LINE) is activated. Also, end the example with a slash (/) to activate it.

See Also: *PL/SQL Packages and Types Reference* for complete information about the DBMS_OUTPUT package

Exceptions let you handle Oracle Database 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 end. The following anonymous block handles the predefined Oracle Database 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_tab
```

```
      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_tab
         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* and "Handling Run-Time PL/SQL Errors" on page 7-35

Anonymous blocks are usually used interactively from a tool, such as SQL*Plus, or in a precompiler, OCI, or SQL*Module application. They are usually used to call stored procedures or to open cursor variables.

> **See Also:** "Cursor Variables" on page 7-30

Stored Program Units (Procedures, Functions, and Packages)

A stored procedure, function, or package is a PL/SQL program unit that:

- Has a name.
- Can take parameters, and can return values.

- Is stored in the data dictionary.
- Can be called by many users.

> **Note:** The term **stored procedure** is sometimes used generically for both stored procedures and stored functions. The only difference between procedures and functions is that functions always return a single value to the caller, while procedures do not return a value to the caller.

Naming Procedures and Functions

Because a procedure or function is stored in the database, it must be named. This distinguishes it from other stored procedures and makes it possible for applications to call it. Each publicly-visible procedure or function in a schema must have a unique name, and 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, then the stored procedure name must also be a legal identifier in the calling host 3GL language, such as Ada or C.

Parameters for Procedures and Functions

Stored procedures and functions can take parameters. The following example shows a stored procedure that is similar to the anonymous block in "Anonymous Blocks" on page 7-2.

> **Caution:** To execute the following, use `CREATE OR REPLACE PROCEDURE...`

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER) IS
   Emp_name       VARCHAR2(10);
   CURSOR         c1 (Depno NUMBER) IS
                     SELECT Ename FROM Emp_tab
                        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 this 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 attributes, described in Table 7–1.

Table 7–1 Attributes of Procedure Parameters

Parameter Attribute	Description
Name	This must be a legal PL/SQL identifier.
Mode	This 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, then `IN` is assumed.
Datatype	This is a standard PL/SQL datatype.

Parameter Modes Parameter modes 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 no arguments and return a single value. It is poor programming practice to have a function return multiple values. Also, functions should be free from side effects, which change the values of variables not local to the subprogram.

Table 7–2 summarizes the information about parameter modes.

Table 7–2 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.

Table 7–2 (Cont.) Parameter Modes

IN	OUT	IN OUT
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.

> **See Also:** *PL/SQL User's Guide and Reference* for details about parameter modes

Parameter Datatypes The datatype of a formal parameter consists of one of the following:

- An *unconstrained* type name, such as `NUMBER` or `VARCHAR2`.
- A type that is *constrained* using the `%TYPE` or `%ROWTYPE` attributes.

> **Note:** Numerically constrained types such as `NUMBER(2)` or `VARCHAR2(20)` are not allowed in a parameter list.

%TYPE and %ROWTYPE Attributes Use the type attributes `%TYPE` and `%ROWTYPE` to constrain the parameter. For example, the `Get_emp_names` procedure specification in "Parameters for Procedures and Functions" on page 7-5 could be written as the following:

```
PROCEDURE Get_emp_names(Dept_num IN Emp_tab.Deptno%TYPE)
```

This has the `Dept_num` parameter take the same datatype as the `Deptno` column in the `Emp_tab` table. The column and table must be available when a declaration using `%TYPE` (or `%ROWTYPE`) is elaborated.

Using `%TYPE` is recommended, because if the type of the column in the table changes, then 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);
```

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_tab table in a PL/SQL record for the given empno:

> **Caution:** To execute the following, use CREATE OR REPLACE PROCEDURE...

```
PROCEDURE Get_emp_rec (Emp_number  IN  Emp_tab.Empno%TYPE,
                       Emp_ret     OUT Emp_tab%ROWTYPE) IS
BEGIN
   SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
      INTO Emp_ret
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
```

You could call this procedure from a PL/SQL block as follows:

```
DECLARE
   Emp_row      Emp_tab%ROWTYPE;     -- declare a record matching a
                                     -- row in the Emp_tab table
BEGIN
   Get_emp_rec(7499, Emp_row);   -- call for Emp_tab# 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_tab.Deptno%TYPE)
   RETURN Emp_tab%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.

> **Note:** When passing a user defined type, such as a PL/SQL table or record to a remote procedure, to make PL/SQL use the same definition so that the type checker can verify the source, you must create a redundant loop back DBLINK so that when the PL/SQL compiles, both sources pull from the same location.

Default Parameter Values Parameters can take default values. 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 could be written as the following:

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER DEFAULT 20) IS ...
```

or

```
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.

> **Note:** 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 a text editor to write the procedure or function. At the beginning of the procedure, place the following statement:

```
CREATE PROCEDURE Procedure_name AS   ...
```

For example, to use the example in "%TYPE and %ROWTYPE Attributes" on page 7-7, create a text (source) file called `get_emp.sql` containing the following code:

```
CREATE PROCEDURE Get_emp_rec (Emp_number  IN  Emp_tab.Empno%TYPE,
                              Emp_ret     OUT Emp_tab%ROWTYPE) AS
BEGIN
   SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
      INTO Emp_ret
      FROM Emp_tab
```

```
      WHERE Empno = Emp_number;
END;
/
```

Then, using an interactive tool such as SQL*Plus, load the text file containing the procedure by entering the following statement:

```
SQL> @get_emp
```

This loads 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.

Use the `CREATE [OR REPLACE] FUNCTION...` statement to store functions.

> **Caution:** When developing a new procedure, it is usually much more convenient to use the `CREATE OR REPLACE PROCEDURE` statement. 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.

> **See Also:** *Oracle Database Reference* for the complete syntax of the `CREATE PROCEDURE` and `CREATE FUNCTION` statements

Privileges to Create Procedures and Functions To create a standalone 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 (to compile the procedure or package successfully) requires the following additional privileges:
  >
  > - The owner of the procedure or package must be explicitly granted the necessary object privileges for all objects referenced within the body of the code.
  > - The owner cannot obtain required privileges through roles.

If the privileges of the owner of a procedure or package change, then the procedure must be reauthenticated before it is run. If a necessary privilege to a referenced object is revoked from the owner of the procedure or package, then the procedure cannot be run.

The EXECUTE privilege on a procedure gives a user the right to run a procedure owned by another user. Privileged users run the procedure under the security domain of the owner of the procedure. Therefore, users never need 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.

Note: Package creation requires a sort. So the user creating the package should be able to create a sort segment in the temporary tablespace with which the user is associated.

See Also: "Privileges Required to Execute a Procedure" on page 7-45

Altering Stored Procedures and Functions

To alter a stored procedure or function, you must first drop it using the DROP PROCEDURE or DROP FUNCTION statement, then re-create it using the CREATE PROCEDURE or CREATE FUNCTION statement. Alternatively, use the CREATE OR REPLACE PROCEDURE or CREATE OR REPLACE FUNCTION statement, which first drops the procedure or function if it exists, then re-creates it as specified.

Caution: The procedure or function is dropped without any warning.

Dropping Procedures and Functions

A standalone procedure, a standalone function, a package body, or an entire package can be dropped using the SQL statements DROP PROCEDURE, DROP FUNCTION, DROP PACKAGE BODY, and DROP PACKAGE, respectively. A DROP PACKAGE statement drops both the specification and body of a package.

The following statement drops the Old_sal_raise procedure in your schema:

```
DROP PROCEDURE Old_sal_raise;
```

Privileges to Drop Procedures and Functions To drop a procedure, function, 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.

External Procedures

A PL/SQL procedure executing on an Oracle Database instance can call an external procedure written in a 3GL. The 3GL procedure runs in a separate address space from that of the database.

> **See Also:** Chapter 8, "Calling External Procedures" for information about external procedures

PL/SQL Packages

A *package* is an encapsulated collection of related program objects (for example, procedures, functions, variables, constants, cursors, and exceptions) stored together in the database.

Using packages is an alternative to creating procedures and functions as standalone schema objects. Packages have many advantages over standalone procedures and functions. For example, they:

- Let you organize your application development more efficiently.
- Let you grant privileges more efficiently.
- Let you modify package objects without recompiling dependent schema objects.
- Enable Oracle Database to read multiple package objects into memory at once.
- Can contain global variables and cursors that are available to all procedures and functions in the package.
- Let you **overload** procedures or functions. Overloading a procedure means creating multiple procedures with the same name in the same package, each taking arguments of different number or datatype.

> **See Also:** *PL/SQL User's Guide and Reference* for more information about subprogram name overloading

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.

Example of a PL/SQL Package Specification and Body The following example shows a package specification for a package named `Employee_management`. The package contains one stored function and two stored procedures. 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
       New_empno    NUMBER(10);

-- This 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.

   BEGIN
      SELECT Emp_sequence.NEXTVAL INTO New_empno FROM dual;
      INSERT INTO Emp_tab VALUES (New_empno, Name, Job, Mgr,
         Hiredate, Sal, Comm, Deptno);
      RETURN (New_empno);
   END Hire_emp;

   PROCEDURE fire_emp(emp_id IN NUMBER) AS

-- This procedure deletes the employee with an employee
-- number that corresponds to the argument Emp_id. If
-- no employee is found, then an exception is raised.

   BEGIN
      DELETE FROM Emp_tab 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
```

```
-- This 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. If employee exists, then update
-- salary with increase.

   BEGIN
      UPDATE Emp_tab
         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, then first create the sequence number `Emp_sequence`. Do this with the following SQL*Plus statement:

```
SQL> CREATE SEQUENCE Emp_sequence
   > START WITH 8000 INCREMENT BY 10;
```

PL/SQL Object Size Limitation

The size limitation for PL/SQL stored database objects such as procedures, functions, triggers, and packages is the size of the DIANA (Descriptive Intermediate Attributed Notation for Ada) code in the shared pool in bytes. The UNIX limit on the size of the flattened DIANA/pcode size is 64K but the limit may be 32K on desktop platforms such as DOS and Windows.

The most closely related number that a user can access is the `PARSED_SIZE` in the data dictionary view `USER_OBJECT_SIZE`. That gives the size of the DIANA in bytes as stored in the `SYS.IDL_xxx$` tables. This is not the size in the shared pool. The size of the DIANA part of PL/SQL code (used during compilation) is significantly larger in the shared pool than it is in the system table.

Size Limitation by Version The size limitation of a PL/SQL package is approximately 128K parsed size in release 7.3. For releases earlier than 7.3 the limitation is 64K.

Creating Packages

Each part of a package is created with a different statement. Create the package specification using the CREATE PACKAGE statement. The CREATE PACKAGE statement declares public package objects.

To create a package body, use the CREATE PACKAGE BODY statement. The CREATE PACKAGE BODY statement 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 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.

It is often more convenient to add the OR REPLACE clause in the CREATE PACKAGE or CREATE PACKAGE BODY statements 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 statements would then be the following:

```
CREATE OR REPLACE PACKAGE Package_name AS ...
```

and

```
CREATE OR REPLACE PACKAGE BODY Package_name AS ...
```

Creating Packaged Objects The body of a package can contain include:

- Procedures and 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 *and* the return type must agree in name and type.

Privileges to Create or Drop Packages The privileges required to create or drop a package specification or package body are the same as those required to create or drop a standalone procedure or function.

See Also:

- "Privileges to Create Procedures and Functions" on page 7-10
- "Privileges to Drop Procedures and Functions" on page 7-12

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.

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 invalidated, then all package instances in the session are invalidated and recompiled. As a result, the session state is lost for all package instances in the session.

When a package in a given session is invalidated, the session receives the following error the first time it attempts to use any object of the invalid package instance:

```
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: For optimal performance, Oracle Database returns this error message only once—each time the package state is discarded.

If you handle this error in your application, ensure that your error handling strategy can accurately handle this error. For example, when a procedure in one package calls a procedure in another package, your application should be aware that the session state is lost for both packages.

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 invalidations are common in your system during working hours, then you might want to code your applications to handle this error when package calls are made.

Packages Supplied With Oracle Database

There are many packages provided with Oracle Database, either to extend the functionality of the database or to give PL/SQL access to SQL features. You can call these packages from your application.

> **See Also:** *PL/SQL Packages and Types Reference* for an overview of these Oracle Database packages

Overview of Bulk Binds

Oracle Database uses two engines to run PL/SQL blocks and subprograms. The PL/SQL engine runs procedural statements, while the SQL engine runs SQL statements. During execution, every SQL statement causes a context switch between the two engines, resulting in performance overhead.

Performance can be improved substantially by minimizing the number of context switches required to run a particular block or subprogram. When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required by the block can cause poor performance. Collections include the following:

- Varrays
- Nested tables
- Index-by tables
- Host arrays

Binding is the assignment of values to PL/SQL variables in SQL statements. **Bulk binding** is binding an entire collection at once. Bulk binds pass the entire collection back and forth between the two engines in a single operation.

Typically, using bulk binds improves performance for SQL statements that affect four or more database rows. The more rows affected by a SQL statement, the greater the performance gain from bulk binds.

Note: This section provides an overview of bulk binds to help you decide if you should use them in your PL/SQL applications. For detailed information about using bulk binds, including ways to handle exceptions that occur in the middle of a bulk bind operation, see the *PL/SQL User's Guide and Reference*.

When to Use Bulk Binds

If you have scenarios like these in your applications, consider using bulk binds to improve performance.

DML Statements that Reference Collections The `FORALL` keyword can improve the performance of `INSERT`, `UPDATE`, or `DELETE` statements that reference collection elements.

For example, the following PL/SQL block increases the salary for employees whose manager's ID number is 7902, 7698, or 7839, both with and without using bulk binds:

```
DECLARE
   TYPE Numlist IS VARRAY (100) OF NUMBER;
   Id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN

-- Efficient method, using a bulk bind
   FORALL i IN Id.FIRST..Id.LAST   -- bulk-bind the VARRAY
      UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);

-- Slower method, running the UPDATE statements within a regular loop
   FOR i IN Id.FIRST..Id.LAST LOOP
      UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);
   END LOOP;
END;
```

Without the bulk bind, PL/SQL sends a SQL statement to the SQL engine for each employee that is updated, leading to context switches that hurt performance.

If you have a set of rows prepared in a PL/SQL table, you can bulk-insert or bulk-update the data using a loop like:

```
FORALL i in Emp_Data.FIRST..Emp_Data.LAST
    INSERT INTO Emp_tab VALUES(Emp_Data(i));
```

SELECT Statements that Reference Collections The BULK COLLECT INTO clause can improve the performance of queries that reference collections.

For example, the following PL/SQL block queries multiple values into PL/SQL tables, both with and without bulk binds:

```
-- Find all employees whose manager's ID number is 7698.
DECLARE
   TYPE Var_tab IS TABLE OF VARCHAR2(20) INDEX BY BINARY_INTEGER;
   Empno VAR_TAB;
   Ename VAR_TAB;
   Counter NUMBER;
   CURSOR C IS
      SELECT Empno, Ename FROM Emp_tab WHERE Mgr = 7698;
BEGIN

-- Efficient method, using a bulk bind
    SELECT Empno, Ename BULK COLLECT INTO Empno, Ename
        FROM Emp_Tab WHERE Mgr = 7698;

-- Slower method, assigning each collection element within a loop.

   counter := 1;
   FOR rec IN C LOOP
      Empno(Counter) := rec.Empno;
      Ename(Counter) := rec.Ename;
      Counter := Counter + 1;
   END LOOP;
END;
```

You can use BULK COLLECT INTO with tables of scalar values, or tables of %TYPE values.

Without the bulk bind, PL/SQL sends a SQL statement to the SQL engine for each employee that is selected, leading to context switches that hurt performance.

FOR Loops that Reference Collections and the Returning Into Clause You can use the FORALL keyword along with the BULK COLLECT INTO keywords to improve the performance of FOR loops that reference collections and return DML.

For example, the following PL/SQL block updates the EMP_TAB table by computing bonuses for a collection of employees; then it returns the bonuses in a column called Bonlist. The actions are performed both with and without using bulk binds:

```
DECLARE
   TYPE Emplist IS VARRAY(100) OF NUMBER;
   Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
   TYPE Bonlist IS TABLE OF Emp_tab.sal%TYPE;
   Bonlist_inst BONLIST;
BEGIN
   Bonlist_inst := BONLIST(1,2,3,4,5);

   FORALL i IN Empids.FIRST..empIDs.LAST
      UPDATE Emp_tab SET Bonus = 0.1 * Sal
      WHERE Empno = Empids(i)
      RETURNING Sal BULK COLLECT INTO Bonlist;

   FOR i IN Empids.FIRST..Empids.LAST LOOP
      UPDATE Emp_tab Set Bonus = 0.1 * sal
         WHERE Empno = Empids(i)
       RETURNING Sal INTO BONLIST(i);
   END LOOP;
END;
```

Without the bulk bind, PL/SQL sends a SQL statement to the SQL engine for each employee that is updated, leading to context switches that hurt performance.

Triggers

A 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. You can also define `INSTEAD OF` triggers or system triggers (triggers on `DATABASE` and `SCHEMA`).

See Also: Chapter 9, "Using Triggers"

Hiding PL/SQL Code with the PL/SQL Wrapper

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` statement at your system prompt using the following syntax:

```
wrap INAME=input_file [ONAME=output_file]
```

See Also: *PL/SQL User's Guide and Reference* for complete instructions on using the PL/SQL Wrapper

Compiling PL/SQL Procedures for Native Execution

You can speed up PL/SQL procedures by compiling them into native code residing in shared libraries. The procedures are translated into C code, then compiled with your usual C compiler and linked into the Oracle Database process.

You can use this technique with both the supplied Oracle Database PL/SQL packages, and procedures you write yourself. You can use the `ALTER SYSTEM` or `ALTER SESSION` command, or update your initialization file, to set the parameter `PLSQL_COMPILER_FLAGS` to include the value `NATIVE`. The default setting includes the value `INTERPRETED`, and you must remove this keyword from the parameter value.

Because this technique cannot do much to speed up SQL statements called from these procedures, it is most effective for compute-intensive procedures that do not spend much time executing SQL.

With Java, you can use the `ncomp` tool to compile your own packages and classes.

See Also:

- *PL/SQL User's Guide and Reference* for details on PL/SQL native compilation
- *Oracle Database Java Developer's Guide* for details on Java native compilation

Remote Dependencies

Dependencies among PL/SQL program units can be handled in two ways:

- Timestamps
- Signatures

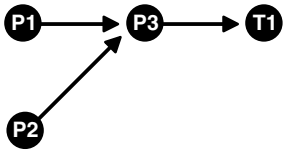
Timestamps

If timestamps are used to handle dependencies among PL/SQL program units, then whenever you alter a program unit or a relevant schema object, all of its dependent units are marked as invalid and must be recompiled before they can be run.

Each program unit carries a timestamp that is set by the server when the unit is created or recompiled. Figure 7–1 demonstrates this graphically. Procedures `P1` and `P2` call stored procedure `P3`. Stored procedure `P3` references table `T1`. In this

example, each of the procedures is dependent on table T1. P3 depends upon T1 directly, while P1 and P2 depend upon T1 indirectly.

Figure 7–1 Dependency Relationships



If P3 is altered, then 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. Therefore, if the procedure P3 is altered and recompiled, then 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 Database instance in a distributed environment, then 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 it 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, because 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. For example, 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 if the server procedure is changed or automatically recompiled, then 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 Database 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.

A signature is associated with each compiled stored program unit. It identifies the unit using the following criteria:

- The name of the unit (the package, procedure, or function name).
- The types of each of the parameters of the subprogram.
- The modes of the parameters (`IN`, `OUT`, `IN OUT`).
- The number of parameters.
- The type of the return value for a function.

The user has control over whether signatures or timestamps govern remote dependencies.

See Also: "Controlling Remote Dependencies" on page 7-28

When the signature dependency model is used, a dependency on a remote program unit causes an invalidation of the dependent unit if the dependent unit contains a call to a subprogram in the parent unit, and if the signature of this subprogram has been changed in an incompatible manner.

For example, consider a procedure `get_emp_name` stored on a server in Boston (`BOSTON_SERVER`). The procedure is defined as the following:

Note: You may need to set up data structures, similar to the following, for certain examples to work:

```
CONNECT system/manager
CREATE PUBLIC DATABASE LINK boston_server USING 'inst1_alias';
CONNECT scott/tiger
```

```
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 BOSTON_SERVER, its signature, as well as its timestamp, is recorded.

Suppose 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(hire_date);
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 to the Boston server, regardless of whether or not there were any changes.

If the timestamp dependency mode is in effect, then a mismatch in timestamps causes an error status to be returned to the calling procedure.

However, if the signature mode is in effect, then 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, then the call succeeds. If they do not match, then 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, possibly due to 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.

> **Note:** `DETERMINISTIC`, `PARALLEL_ENABLE`, and purity information do not show in the signature mode. Optimizations based on these settings are not automatically reconsidered if a function on a remote system is redefined with different settings. This may lead to incorrect query results when calls to the remote function occur, even indirectly, in a SQL statement, or if the remote function is used, even indirectly, in a function-based index.

When Does a Signature Change?

Switching Datatype Classes

A signature changes when you switch 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. Datatypes that are not listed in the following table, such as `NCHAR` or `TIMESTAMP`, are not part of any class; changing their type always causes a signature mismatch.

VARCHAR types: `VARCHAR2, VARCHAR, STRING, LONG, ROWID`

Character types: `CHARACTER, CHAR`

Raw types: `RAW, LONG RAW`

Integer types: `BINARY_INTEGER, PLS_INTEGER, BOOLEAN, NATURAL, POSITIVE, POSITIVEN, NATURALN`

Number types: `NUMBER, INTEGER, INT, SMALLINT, DECIMAL, DEC, REAL, FLOAT, NUMERIC, DOUBLE PRECISION, DOUBLE PRECISION, NUMERIC`

Date types: `DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND`

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 between:

```
PROCEDURE P1 (Param1 NUMBER);
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 Changing Procedure Signatures

Using the `Get_emp_names` procedure defined in "Parameters for Procedures and Functions" on page 7-5, if the procedure body is changed to the following:

```
DECLARE
   Emp_number  NUMBER;
   Hire_date   DATE;
BEGIN
-- date format model changes

   SELECT Ename, To_char(Hiredate, 'DD/MON/YYYY')
      INTO Emp_name, Hire_date
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
```

The specification of the procedure has not changed, so its signature has not changed.

But if the procedure specification is changed to the following:

```
CREATE OR REPLACE PROCEDURE Get_emp_name (
   Emp_number  IN  NUMBER,
   Hire_date   OUT DATE,
   Emp_name    OUT VARCHAR2) AS
```

And if 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`, 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_tab
           WHERE Empno = Emp_data.Emp_number;
   END;
END;
```

If the package specification is changed so that the record's field names are changed, but the types remain the same, then 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:

```
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

The dynamic initialization parameter REMOTE_DEPENDENCIES_MODE controls whether the timestamp or the signature dependency model is in effect.

- If the initialization parameter file contains the following specification:

  ```
  REMOTE_DEPENDENCIES_MODE = TIMESTAMP
  ```

  Then only timestamps are used to resolve dependencies (if this is not explicitly overridden dynamically).

- If the initialization parameter file contains the following parameter specification:

  ```
  REMOTE_DEPENDENCIES_MODE = SIGNATURE
  ```

  Then signatures are used to resolve dependencies (if this not explicitly overridden dynamically).

- You can alter the mode dynamically by using the DDL statements. For example, this example alters the dependency model for the current session:

  ```
  ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =
      {SIGNATURE | TIMESTAMP}


  Thise example alters the dependency model systemwide after startup:
  ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =
      {SIGNATURE | TIMESTAMP}
  ```

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 statements, then timestamp is the default value. Therefore, unless you explicitly use the REMOTE_DEPENDENCIES_MODE parameter, or the appropriate DDL statement, your server is operating using the timestamp dependency model.

When you use REMOTE_DEPENDENCIES_MODE=SIGNATURE:

- If you change the default value of a parameter of a remote procedure, then the local procedure calling the remote procedure is not invalidated. If the call to the

remote procedure does not supply the parameter, then the default value is used. In this case, because invalidation/recompilation does not automatically occur, the old default value is used. If you want to see the new default values, then 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), then local procedures that call the remote procedure are not invalidated. If it turns out that this overloading results in a rebinding of existing calls from the local procedure under the timestamp mode, then 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, then the local calling procedure is not invalidated or 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 program 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, then the calling (dependent) unit is invalidated and must be recompiled. In this case, if there is no local PL/SQL compiler, then the calling application cannot proceed.

In the timestamp dependency mode, signatures are not compared. If there is a local PL/SQL compiler, then recompilation happens automatically when the calling procedure is run.

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. 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 "When Does a Signature Change?" on page 7-25), then an error is returned to the calling session.

Suggestions for Managing Dependencies

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 they want 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, add new procedures to the end of the procedure (or function) declarations in a package specification. Adding a new procedure in the middle of the list of declarations can cause unnecessary invalidation and recompilation of dependent procedures.

Cursor Variables

A cursor is a static object; a cursor variable is a pointer to a cursor. Because cursor variables are pointers, they can be passed and returned as parameters to procedures and functions. A cursor variable can also refer to different cursors in its lifetime.

Some additional advantages of cursor variables include:

- *Encapsulation* Queries are centralized in the stored procedure that opens the cursor variable.
- *Ease of maintenance* If you need to change the cursor, then 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 Also:** *PL/SQL User's Guide and Reference* for details on cursor variables

Declaring and Opening Cursor Variables

Memory is usually allocated for a cursor variable in the client application using the appropriate ALLOCATE statement. In Pro*C, use the EXEC SQL ALLOCATE <cursor_name> statement. In OCI, 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++ Programmer's Guide*
- *Pro*COBOL Programmer's Guide*
- *Oracle Call Interface Programmer's Guide*
- *Oracle SQL*Module for Ada Programmer's Guide*

Fetching Data

The following package defines a PL/SQL cursor variable type Emp_val_cv_type, and two procedures. The first procedure, Open_emp_cv, opens the cursor variable using a bind variable in the WHERE clause. The second procedure, Fetch_emp_data, fetches rows from the Emp_tab table using the cursor variable.

```
CREATE OR REPLACE PACKAGE Emp_data AS
  TYPE Emp_val_cv_type IS REF CURSOR RETURN Emp_tab%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_tab%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_tab WHERE deptno = dept_number;
  END open_emp_cv;
  PROCEDURE Fetch_emp_data (Emp_cv      IN  Emp_val_cv_type,
```

```
                              Emp_row     OUT Emp_tab%ROWTYPE) IS
  BEGIN
    FETCH Emp_cv INTO Emp_row;
  END Fetch_emp_data;
END Emp_data;
```

The following example shows how to 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_tab.Deptno%TYPE;
   Emp_row Emp_tab%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_tab WHERE Sal > 2000;
```

```
    ELSIF Discrim = 2 THEN
      OPEN Cv FOR SELECT * FROM Dept_tab;
    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_tab` table or the `Dept_tab` table. The following PL/SQL block shows how to fetch using the cursor variable, and then use the `ROWTYPE_MISMATCH` predefined exception to handle either fetch:

```
DECLARE
  Emp_rec  Emp_tab%ROWTYPE;
  Dept_rec Dept_tab%ROWTYPE;
  Cv       Emp_dept_data.CV_TYPE;

BEGIN
  Emp_dept_data.open_cv(Cv, 1); -- Open Cv For Emp_tab Fetch
  Fetch cv INTO Dept_rec;       -- but fetch into Dept_tab 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_tab data...');
      FETCH Cv INTO Emp_rec;
      DBMS_OUTPUT.PUT(Emp_rec.Deptno);
      DBMS_OUTPUT.PUT_LINE('  ' || Emp_rec.Ename);
    END;
```

Handling PL/SQL Compile-Time Errors

When you use SQL*Plus to submit PL/SQL code, and when the code contains errors, you receive notification that compilation errors have occurred, but there is no immediate indication of what the errors are. For example, if you submit a standalone (or stored) procedure `PROC1` in the file `proc1.sql` as follows:

```
SQL> @proc1
```

And, if there are one or more errors in the code, then you receive a notice such as the following:

```
MGR-00072: Warning: Procedure proc1 created with compilation errors
```

In this case, use the SHOW ERRORS statement 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 Also:** *SQL*Plus User's Guide and Reference* for complete information about the SHOW ERRORS statement

---

Note: Before issuing the SHOW ERRORS statement, use the SET LINESIZE statement to get long lines on output. The value 132 is usually a good choice. For example:

```
SET LINESIZE 132
```

Assume that you want to create a simple procedure that deletes records from the employee table using SQL*Plus:

```
CREATE OR REPLACE PROCEDURE Fire_emp(Emp_id NUMBER) AS
   BEGIN
      DELETE FROM Emp_tab 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 entered and an error is returned, a SHOW ERRORS statement returns 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 statement.

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 it is 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:** *Oracle Database Reference* for more information about these data dictionary views

Handling Run-Time PL/SQL Errors

Oracle Database 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. After received, the client application can handle the error based on the user-specified error number and message returned by Oracle Database.

User-specified error messages are returned using the RAISE_APPLICATION_ERROR procedure. For example:

```
RAISE_APPLICATION_ERROR(Error_number, 'text', Keep_error_stack)
```

This procedure stops 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 Database 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.parameter = "NLS_LANGUAGE";
   Raise_application_error(-20101, Message);
...
```

> **See Also:** "Handling Errors in Remote Procedures" on page 7-38 for information on exception handling when calling remote procedures

The following section includes an example of passing a user-specified error number from a trigger to a procedure.

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 usual execution of the PL/SQL block stops, and a routine called an exception handler is called. 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, then two options are available:

- Enter a RAISE statement that names the appropriate exception. A RAISE statement stops the execution of the procedure, and control passes to an exception handler (if any).

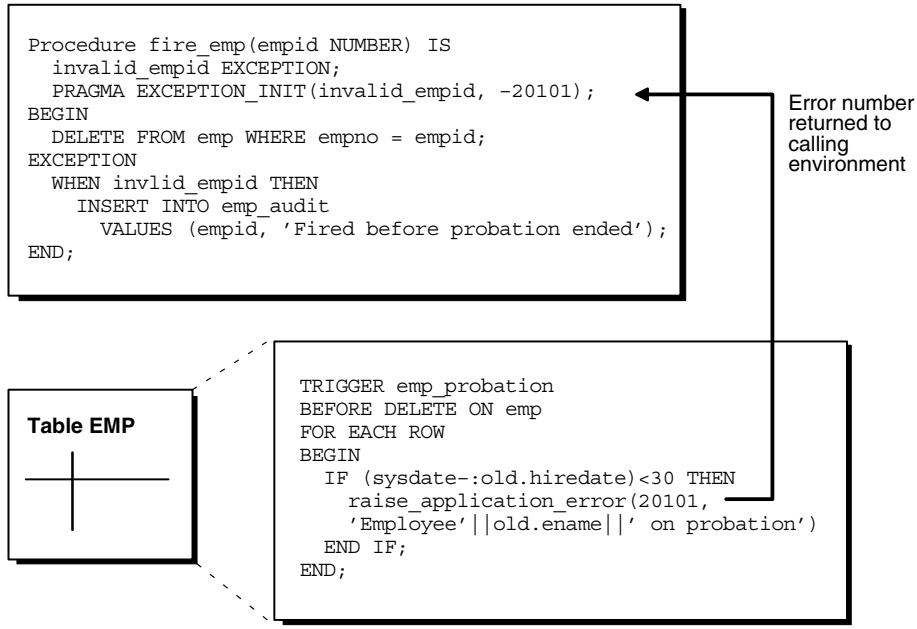
- Call the `RAISE_APPLICATION_ERROR` procedure to return a user-specified error number and message.

You can also define an exception handler to handle user-specified error messages. For example, Figure 7–2 on page 7-37 illustrates the following:

- An exception and associated exception handler in a procedure
- A conditional statement that checks for an error (such as transferring funds not available) and enters a user-specified error number and message within a trigger
- How user-specified error numbers are returned to the calling environment (in this case, a procedure), and how that application can define an exception that corresponds to the user-specified error number

Declare a user-defined exception in a procedure or package body (private exceptions), or in the specification of a package (public exceptions). *Define* an exception handler in the body of a procedure (standalone or package).

Figure 7–2 Exceptions and User-Defined Errors



Unhandled Exceptions

In database PL/SQL program units, an unhandled user-error condition or internal error condition that is not trapped by an appropriate exception handler causes the implicit rollback of the program unit. If the program unit includes a `COMMIT` statement before the point at which the unhandled exception is observed, then 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, then 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.

Handling Errors in Distributed Queries

You can use a trigger or a stored procedure to create a distributed query. This distributed query is decomposed by the local Oracle Database instance into a corresponding number of remote queries, which are sent to the remote nodes for execution. The remote nodes run 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, possibly due to an integrity constraint violation, then Oracle Database returns error number `ORA-02055`. Subsequent statements, or procedure calls, return error number `ORA-02067` until a rollback or a rollback to savepoint is entered.

You should design your application to check for any returned error messages that indicates that a portion of the distributed update has failed. If you detect a failure, then 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 run 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, then 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.

Debugging Stored Procedures

Compiling a stored procedure involves fixing any syntax errors in the code. You might need to do additional debugging to make sure that the procedure works correctly, performs well, and recovers from errors. Such debugging might involve:

- Adding extra output statements to verify execution progress and check data values at certain points within the procedure.
- Running a separate debugger to analyze execution in greater detail.

Oracle JDeveloper

Recent releases of Oracle JDeveloper have extensive features for debugging PL/SQL, Java, and multi-language programs. You can get Oracle JDeveloper as part of various Oracle product suites. Often, a more recent release is available as a download at `http://otn.oracle.com/`.

Oracle Procedure Builder and TEXT_IO Package

Oracle Procedure Builder is an advanced client/server debugger that transparently debugs your database applications. It lets you run 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. Oracle Procedure Builder is part of the Oracle Developer tool set. It also provides the `TEXT_IO` package that is useful for printing debug information.

DBMS_OUTPUT Package

You can also debug stored procedures and triggers using the `DBMS_OUTPUT` supplied package. Put `PUT` and `PUT_LINE` statements in your code to output the value of variables and expressions to your terminal.

Privileges for Debugging PL/SQL and Java Stored Procedures

Starting with Oracle Database 10*g*, a new privilege model applies to debugging PL/SQL and Java code running within the database. This model applies whether you are using Oracle JDeveloper, Oracle Developer, or any of the various third-party PL/SQL or Java development environments, and it affects both the `DBMS_DEBUG` and `DBMS_DEBUG_JDWP` APIs.

For a session to connect to a debugger, the effective user at the time of the connect operation must have the `DEBUG CONNECT SESSION` system privilege. This effective user may be the owner of a definer's rights routine involved in making the connect call.

When a debugger becomes connected to a session, the session login user and the currently enabled session-level roles are fixed as the privilege environment for that debugging connection. Any DEBUG or EXECUTE privileges needed for debugging must be granted to that combination of user and roles.

- To be able to display and change Java public variables or variables declared in a PL/SQL package specification, the debugging connection must be granted either EXECUTE or DEBUG privilege on the relevant code.
- To be able to either display and change private variables or breakpoint and execute code lines step by step, the debugging connection must be granted DEBUG privilege on the relevant code

> **Caution:** The DEBUG privilege effectively allows a debugging session to do anything that the procedure being debugged could have done if that action had been included in its code.

In addition to these privilege requirements, the ability to stop on individual code lines and debugger access to variables are allowed only in code compiled with debug information generated. The PLSQL_DEBUG parameter and the DEBUG keyword on commands such as ALTER PACKAGE can be used to control whether the PL/SQL compiler includes debug information in its results. If it does not, variables will not be accessible, and neither stepping nor breakpoints will stop on code lines. The PL/SQL compiler will never generate debug information for code that has been obfuscated using the PL/SQL wrap utility.

The DEBUG ANY PROCEDURE system privilege is equivalent to the DEBUG privilege granted on *all* objects in the database. Objects owned by SYS are included if the value of the O7_DICTIONARY_ACCESSIBILITY parameter is TRUE.

A debug role mechanism is available to carry privileges needed for debugging that are not normally enabled in the session. Refer to the documentation on the DBMS_DEBUG and DBMS_DEBUG_JDWP packages for details on how to specify a debug role and any necessary related password.

The JAVADEBUGPRIV role carries the DEBUG CONNECT SESSION and DEBUG ANY PROCEDURE privileges. Grant it only with the care those privileges warrant.

> **Caution:** Granting DEBUG ANY PROCEDURE privilege, or granting DEBUG privilege on any object owned by SYS, means granting *complete rights to the database*.

Writing Low-Level Debugging Code

If you are actually writing code that will be part of a debugger, you might need to use packages such as DBMS_DEBUG_JDWP or DBMS_DEBUG.

DBMS_DEBUG_JDWP Package

The DBMS_DEBUG_JDWP package, provided starting with Oracle9*i* Release 2, provides a framework for multi-language debugging that is expected to supersede the DBMS_DEBUG package over time. It is especially useful for programs that combine PL/SQL and Java.

DBMS_DEBUG Package

The DBMS_DEBUG package, provided starting with Oracle8*i*, implements server-side debuggers and provides a way to debug server-side PL/SQL program units. Several of the debuggers available, such as Oracle Procedure Builder and various third-party vendor solutions, use this API.

See Also:

- *Oracle Procedure Builder Developer's Guide*
- *PL/SQL Packages and Types Reference* for more information about the DBMS_DEBUG and DBMS_OUTPUT packages and associated privileges
- The Oracle JDeveloper documentation for information on using package DBMS_DEBUG_JDWP
- *Oracle Database SQL Reference* for more details on privileges
- The PL/SQL page at http://otn.oracle.com/ for information about writing low-level debug code

Calling Stored Procedures

Note: You may need to set up data structures, similar to the following, for certain examples to work:

```
CREATE TABLE Emp_tab (
   Empno    NUMBER(4) NOT NULL,
   Ename    VARCHAR2(10),
   Job      VARCHAR2(9),
   Mgr      NUMBER(4),
   Hiredate DATE,
   Sal      NUMBER(7,2),
   Comm     NUMBER(7,2),
   Deptno   NUMBER(2));

CREATE OR REPLACE PROCEDURE fire_emp1(Emp_id NUMBER) AS
   BEGIN
      DELETE FROM Emp_tab WHERE Empno = Emp_id;
   END;
VARIABLE Empnum NUMBER;
```

Procedures can be called 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 Database tool.
- A procedure can be explicitly called within an application, such as a SQL*Forms or a 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`.

This section includes some common examples of calling procedures from within these environments.

See Also: "Calling Stored Functions from SQL Expressions" on page 7-50

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 following line:

```
. . .
```

```
Sal_raise(Emp_id, 200);
. . .
```

This line calls the `Sal_raise` procedure. `Emp_id` is a variable within the context of the procedure. Recursive procedure calls are allowed within PL/SQL: A procedure can call itself.

Interactively Calling Procedures From Oracle Database Tools

A procedure can be called interactively from an Oracle Database tool, such as SQL*Plus. For example, to call a procedure named `SAL_RAISE`, owned by you, you can use an anonymous PL/SQL block, as follows:

```
BEGIN
    Sal_raise(7369, 200);
END;
```

> **Note:** Interactive tools, such as SQL*Plus, require you to follow these lines with a slash (/) to run the PL/SQL block.

An easier way to run a block is to use the SQL*Plus statement `EXECUTE`, which wraps `BEGIN` and `END` statements around the code you enter. For example:

```
EXECUTE Sal_raise(7369, 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
```

After defined, any session variable can be used for the duration of the session. For example, you might run 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 Also:

- *SQL*Plus User's Guide and Reference*
- Your tools documentation for information about performing similar operations using your development tool

Calling Procedures within 3GL Applications

A 3GL database application, such as a precompiler or an OCI application, can include a call to a procedure within the code of the application.

To run 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_emp1(:Empnun);
```

In this case, `:Empno` is a host (bind) variable within the context of the application.

To run 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_emp1(:Empnum);
   END;
END-EXEC;
```

See Also: For information about calling PL/SQL procedures from within 3GL applications:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*
- *Oracle SQL*Module for Ada Programmer's Guide*

Name Resolution When Calling Procedures

References to procedures and packages are resolved according to the algorithm described in the "Rules for Name Resolution in SQL Statements" section of Chapter 2, "Designing Schema Objects".

Privileges Required to Execute a Procedure

If you are the owner of a standalone procedure or package, then you can run the standalone procedure or packaged procedure, or any public procedure or packaged

procedure at any time, as described in the previous sections. If you want to run a standalone or packaged procedure owned by another user, then the following conditions apply:

- You must have the EXECUTE privilege for the standalone procedure or package containing the procedure, or you must have the EXECUTE ANY PROCEDURE system privilege. If you are executing a remote procedure, then you must be granted the EXECUTE privilege or EXECUTE ANY PROCEDURE system privilege directly, *not* through a role.
- You must include the name of the owner in the call. For example:[3]

```
EXECUTE Jward.Fire_emp (1043);
EXECUTE Jward.Hire_fire.Fire_emp (1043);
```

- If the procedure is a **definer's-rights procedure**, then it runs with the privileges of the procedure *owner*. The owner must have all the necessary object privileges for any referenced objects.
- If the procedure is an **invoker's-rights procedure**, then it runs with your privileges (as the invoker). In this case, you also need privileges on all referenced objects; that is, all objects accessed by the procedure through external references that are resolved in your schema. You may hold these privileges directly or through a role. Roles are enabled unless an invoker's-rights procedure is called directly or indirectly by a definer's-rights procedure.

Specifying Values for Procedure Arguments

When you call 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.

---

[3] You may need to set up the following data structures for certain examples to work:

```
CONNECT sys/change_on_install AS Sysdba;
CREATE USER Jward IDENTIFIED BY Jward;
GRANT CREATE ANY PACKAGE TO Jward;
GRANT CREATE SESSION TO Jward;
GRANT EXECUTE ANY PROCEDURE TO Jward;
CONNECT Scott/Tiger
```

For example, these statements each call the procedure `Sal_raise` 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, then 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, then values identified in order must precede values identified by name.

If you used the `DEFAULT` option to define default values for `IN` parameters to a subprogram (see the *PL/SQL User's Guide and Reference*),then you can pass different numbers of actual parameters to the first subprogram, accepting or overriding the default values as you please. If an actual value is not passed, then 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), then you must explicitly designate the name of the argument, as well as its value.

Calling Remote Procedures

Call remote procedures using an appropriate database link and the procedure name. The following SQL*Plus statement runs the procedure `Fire_emp` located in the database and pointed to by the local database link named `BOSTON_SERVER`:

```
EXECUTE fire_emp1@boston_server(1043);
```

> **See Also:** "Handling Errors in Remote Procedures" on page 7-38 for information on exception handling when calling remote procedures

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 OR REPLACE PROCEDURE fire_emp(emp_id NUMBER) IS
BEGIN
    DELETE FROM emp@boston_server WHERE empno = emp_id;
END;
```

The following list explains how to properly call remote procedures, depending on the calling environment.

- Remote procedures (standalone and packaged) can be called from within a procedure, an OCI application, or a precompiler application by specifying the remote procedure name, a database link, and the arguments for the remote procedure.

  ```
  CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
  BEGIN
    fire_emp1@boston_server(arg);
  END;
  ```

- In the previous example, you could create a synonym for `FIRE_EMP1@BOSTON_SERVER`. This would enable you to call the remote procedure from an Oracle Database tool application, such as a SQL*Forms application, as well from within a procedure, OCI application, or precompiler application.

  ```
  CREATE SYNONYM synonym1 for  fire_emp1@boston_server;
  CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
  BEGIN
    synonym1(arg);
  END;
  ```

- If you do not want to use a synonym, then you could write a local cover procedure to call the remote procedure.

  ```
  DECLARE
     arg NUMBER;
  BEGIN
     local_procedure(arg);
  END;
  ```

  Here, `local_procedure` is defined as in the first item of this list.

See Also: "Synonyms for Procedures and Packages" on page 7-49

Caution: Unlike 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, then the work done by the remote procedure is also rolled back.

A procedure called remotely can usually execute a `COMMIT`, `ROLLBACK`, or `SAVEPOINT` statement, the same as a local procedure. However, there are some differences in behavior:

- If the transaction was originated by a non-Oracle database, as may be the case in XA applications, these operations are not allowed in the remote procedure.
- After doing one of these operations, the remote procedure cannot start any distributed transactions of its own.
- If the remote procedure does not commit or roll back its work, the commit is done implicitly when the database link is closed. In the meantime, further calls to the remote procedure are not allowed because it is still considered to be performing a transaction.

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, then 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, then the remote procedure is not run, and the local procedure is invalidated.

Synonyms for Procedures and Packages

Synonyms can be created for standalone procedures and packages to do the following:

- Hide the identity of the name and owner of a procedure or package.
- Provide location transparency for remotely stored procedures (standalone or within a package).

When a privileged user needs to call a procedure, an associated synonym can be used. Because the procedures defined within a package are not individual objects (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 higher.) 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 need 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, then SQL statements in your PL/SQL function may also be run in parallel (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, or, wherever expressions can occur in SQL. For example, they can be called from the following:

- The select list of the `SELECT` statement.
- 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 statement.
- The SET clause of the UPDATE statement.

You cannot call stored PL/SQL functions from a CHECK constraint clause of a CREATE or ALTER TABLE statement 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 for SQL Calling a PL/SQL Function

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 you created that is called My_func, in the My_funcs_pkg package, in the Scott schema, that takes two numeric parameters, you could call the following:

```
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, then 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 Database proceeds as follows:

- Oracle Database first checks for the Payroll package in the current schema.
- If the PAYROLL package is found in the current schema, then Oracle Database looks for a Tax_rate function in the Payroll package. If a Tax_rate function is not found in the Payroll package, then an error message is returned.
- If a Payroll package is not found, then Oracle Database 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, then 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_tab(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_tab.New_sal:

```
SELECT New_sal FROM Emp_tab;
SELECT Emp_tab.New_sal FROM Emp_tab;
```

To access the function new_sal, enter the following:

```
SELECT Scott.New_sal FROM Emp_tab;
```

Example of Calling a PL/SQL Function from SQL For example, to call the Tax_rate PL/SQL function from schema Scott, run it against the Ss_no and sal columns in Tax_table, and place the results in the variable Income_tax, specify the following:

Note: You may need to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE Tax_table (
   Ss_no  NUMBER,
   Sal    NUMBER);

CREATE OR REPLACE FUNCTION tax_rate (ssn IN NUMBER, salary IN
NUMBER) RETURN NUMBER IS
   sal_out NUMBER;
   BEGIN
      sal_out := salary * 1.1;
   END;
```

```
DECLARE
   Tax_id     NUMBER;
   Income_tax NUMBER;
BEGIN
   SELECT scott.tax_rate (Ss_no, Sal)
      INTO Income_tax
      FROM Tax_table
      WHERE Ss_no = Tax_id;
END;
```

These sample calls to PL/SQL functions are allowed in SQL expressions:

```
Circle_area(Radius)
Payroll.Tax_rate(Empno)
scott.Payroll.Tax_rate@boston_server(Dependents, Empno)
```

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 supported. For functions that do not accept arguments, use `()`.

Using Default Values

The stored function `Gross_pay` initializes two of its formal parameters to default values using the `DEFAULT` clause. For example:

```
CREATE OR REPLACE 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`. This is because you can use named notation, which lets you skip parameters. For example:

```
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`. This is because you cannot use named notation.

Privileges

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 must have SELECT privileges on the view. No separate EXECUTE privileges are necessary to select from the view.

Requirements for Calling PL/SQL Functions from SQL Expressions

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; in other words, 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 built-in 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 built-in type.

For example, the following stored function meets the basic requirements:

Note: You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Payroll(
                 Srate             NUMBER,
                 Orate             NUMBER,
                 Acctno            NUMBER);
```

```
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

The **purity** of a stored subprogram (function or procedure) refers to the side effects of that subprogram on database tables or package 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. Various side effects are not allowed when a subprogram is called from a SQL query or DML statement.

In releases prior to Oracle8*i*, Oracle Database leveraged the PL/SQL compiler to enforce restrictions during the compilation of a stored subprogram or a SQL statement. Starting with Oracle8*i*, the compile-time restrictions were relaxed, and a smaller set of restrictions are enforced during execution.

This change provides uniform support for stored subprograms written in PL/SQL, Java, and C, and it allows programmers the most flexibility possible.

> **See Also:** "Restrictions" on page 7-55 for information on the runtime restrictions

Restrictions

When a SQL statement is run, checks are made to see if it is logically embedded within the execution of an already running SQL statement. This occurs if the statement is run from a trigger or from a subprogram that was in turn called from the already running SQL statement. In these cases, further checks occur to determine if the new SQL statement is safe in the specific context.

The following restrictions are enforced on subprograms:

- A subprogram called from a query or DML statement may not end the current transaction, create or rollback to a savepoint, or `ALTER` the system or session.
- A subprogram called from a query (`SELECT`) statement or from a parallelized DML statement may not execute a DML statement or otherwise modify the database.
- A subprogram called from a DML statement may not read or modify the particular table being modified by that DML statement.

These restrictions apply regardless of what mechanism is used to run the SQL statement inside the subprogram or trigger. For example:

- They apply to a SQL statement called from PL/SQL, whether embedded directly in a subprogram or trigger body, run using the native dynamic mechanism (`EXECUTE IMMEDIATE`), or run using the `DBMS_SQL` package.
- They apply to statements embedded in Java with SQLJ syntax or run using JDBC.
- They apply to statements run with OCI using the callback context from within an "external" C function.

You can avoid these restrictions if the execution of the new SQL statement is not logically embedded in the context of the already running statement. PL/SQL's autonomous transactions provide one escape (see "Autonomous Transactions" on page 5-28). Another escape is available using Oracle Call Interface (OCI) from an external C function, if you create a new connection rather than using the handle available from the `OCIExtProcContext` argument.

Declaring a Function

The keywords `DETERMINISTIC` and `PARALLEL_ENABLE` can be used in the syntax for declaring a function. These are optimization hints, informing the query optimizer and other software components about those functions that need not be called redundantly and about those that may be used within a parallelized query or parallelized DML statement. Only functions that are `DETERMINISTIC` are allowed in function-based indexes and in certain snapshots and materialized views.

A function that is dependent solely on the values passed into it as arguments, and does not reference or modify the contents of package variables or the database, or have any other side-effects, is called **deterministic**. Such a function reliably produces the exact same result value for any particular combination of argument values passed into it.

The `DETERMINISTIC` keyword is placed after the return value type in a declaration of the function. For example:

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN P1 * 2;
END;
```

This keyword may be placed on a function defined in a `CREATE FUNCTION` statement, in a function declaration in a `CREATE PACKAGE` statement, or on a method declaration in a `CREATE TYPE` statement. It should not be repeated on the

function's or method's body in a CREATE PACKAGE BODY or CREATE TYPE BODY statement.

Certain performance optimizations occur on calls to functions that are marked DETERMINISTIC, without any other action being required. The database cannot recognize if the function's behavior is indeed deterministic. If the DETERMINISTIC keyword is applied to a function whose behavior is not truly deterministic, then the result of queries involving that function is unpredictable.

The following features require that any function used with them be declared DETERMINISTIC.

- Any function used in a function-based index.
- Any function used in a materialized view, if that view is to qualify for Fast Refresh or is marked ENABLE QUERY REWRITE.

Both of these features attempt to use previously calculated results rather than calling the function when it is possible to do so.

Functions that are used in a WHERE, ORDER BY, or GROUP BY clause, are MAP or ORDER methods of a SQL type, or in any other way are part of determining whether or where a row should appear in a result set also should be DETERMINISTIC as discussed previously. Oracle Database cannot require that they be explicitly declared DETERMINISTIC without breaking existing applications, but the use of the keyword might be a wise choice of style within your application.

Parallel Query and Parallel DML

Oracle Database's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement which is run in parallel may have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that process.

Each process has its own copy of package variables. When parallel execution begins, these are initialized based on the information in the package specification and body as if a new user is logging into the system; the values in package variables are not copied from the original login session. And changes made to package variables are not automatically propagated between the various sessions or back to the original session. Java STATIC class attributes are similarly initialized and modified independently in each process. Because a function can use package (or Java STATIC) variables to accumulate some value across the various rows it encounters, Oracle Database cannot assume that it is safe to parallelize the execution of all user-defined functions.

For query (SELECT) statements in Oracle Database versions prior to 8.1.5, the parallel query optimization looked to see if a function was noted as RNPS and WNPS in a PRAGMA RESTRICT_REFERENCES declaration; those functions that were marked as both RNPS and WNPS could be run in parallel. Functions defined with a CREATE FUNCTION statement had their code implicitly examined to determine if they were pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

> **See Also:** "PRAGMA RESTRICT_REFERENCES – for Backward Compatibility" on page 7-59

For DML statements in Oracle Database versions prior to 8.1.5, the parallelization optimization looked to see if a function was noted as having all four of RNDS, WNDS, RNPS and WNPS specified in a PRAGMA RESTRICT_REFERENCES declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a CREATE FUNCTION statement had their code implicitly examined to determine if they were actually pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

Oracle Database versions 8.1.5 and later continue to parallelize those functions that earlier versions recognize as parallelizable. The PARALLEL_ENABLE keyword is the preferred way to mark your code as safe for parallel execution. This keyword is syntactically similar to DETERMINISTIC as described previously; it is placed after the return value type in a declaration of the function, as in:

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN P1 * 2;
END;
```

This keyword may be placed on a function defined in a CREATE FUNCTION statement, in a function declaration in a CREATE PACKAGE statement, or on a method declaration in a CREATE TYPE statement. It should not be repeated on the function's or method's body in a CREATE PACKAGE BODY or CREATE TYPE BODY statement.

Note that a PL/SQL function that is defined with CREATE FUNCTION may still be run in parallel without any explicit declaration that it is safe to do so, if the system can determine that it neither reads nor writes package variables nor calls any function that might do so. A Java method or C function is never seen by the system as safe to run in parallel, unless the programmer explicitly indicates PARALLEL_

ENABLE on the "call specification", or provides a PRAGMA RESTRICT_REFERENCES indicating that the function is sufficiently pure.

An additional runtime restriction is imposed on functions run in parallel as part of a parallelized DML statement. Such a function is not permitted to in turn execute a DML statement; it is subject to the same restrictions that are enforced on functions that are run inside a query (SELECT) statement.

See Also: "Restrictions" on page 7-55

PRAGMA RESTRICT_REFERENCES – for Backward Compatibility

In Oracle Database versions prior to 8.1.5 (Oracle8*i*), programmers used the pragma RESTRICT_REFERENCES to assert the purity level of a subprogram. In subsequent versions, use the hints parallel-enable and deterministic, instead, to communicate subprogram purity to Oracle Database.

You can remove RESTRICT_REFERENCES from your code. However, this pragma remains available for *backward compatibility* in situations where one of the following is true:

- It is impossible or impractical to edit existing code to remove RESTRICT_REFERENCES completely. If you do not remove it from a subprogram *S1* that depends on another subprogram *S2*, then RESTRICT_REFERENCES might also be needed in *S2*, so that *S1* will compile.
- Replacing RESTRICT_REFERENCES in existing code with hints parallel-enable and deterministic would negatively affect the behavior of new, dependent code. Use RESTRICT_REFERENCES to preserve the behavior of the existing code.

An existing PL/SQL application can thus continue using the pragma even on new functionality, to ease integration with the existing code. Do not use the pragma in a wholly new application.

If you use the pragma RESTRICT_REFERENCES, place it in a package specification, not in a package body. It must follow the declaration of a subprogram (function or procedure), but it need not follow immediately. Only one pragma can reference a given subprogram declaration.

Note: The pragma RESTRICT_REFERENCES applies to both functions and procedures. Purity levels are important for functions, but also for procedures that are called by functions.

To code the pragma RESTRICT_REFERENCES, use the following syntax:

```
PRAGMA RESTRICT_REFERENCES (
    Function_name, WNDS [, WNPS] [, RNDS] [, RNPS] [, TRUST] );
```

Where:

Keyword	Description
WNDS	The subprogram writes no database state (does not modify database tables).
RNDS	The subprogram reads no database state (does not query database tables).
WNPS	The subprogram writes no package state (does not change the values of packaged variables).
RNPS	The subprogram reads no package state (does not reference the values of packaged variables).
TRUST	The other restrictions listed in the pragma are not enforced; they are simply assumed to be true. This allows easy calling from functions that have RESTRICT_REFERENCES declarations to those that do not.

You can pass the arguments in any order. If any SQL statement inside the subprogram body violates a rule, then you get an error when the statement is parsed.

In the following example, the function compound neither reads nor writes database or package state; therefore, you can assert the maximum purity level. Always assert the highest purity level that a subprogram allows. That way, the PL/SQL compiler never rejects the subprogram unnecessarily.

Note: You may need to set up the following data structures for certain examples here to work:

```
CREATE TABLE Accts (
 Yrs      NUMBER,
 Amt      NUMBER,
 Acctno   NUMBER,
 Rte      NUMBER);
```

```
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:

```
DECLARE
   Interest NUMBER;
   Acct_id NUMBER;
BEGIN
   SELECT Finance.Compound(Yrs, Amt, Rte)  -- function call
   INTO   Interest
   FROM   Accounts
   WHERE  Acctno = Acct_id;
```

Using the Keyword TRUST The keyword `TRUST` in the `RESTRICT_REFERENCES` syntax allows easy calling from functions that have `RESTRICT_REFERENCES` declarations to those that do not. When `TRUST` is present, the restrictions listed in the pragma are not actually enforced, but rather are simply assumed to be true.

When calling from a section of code that is using pragmas to one that is not, there are two likely usage styles. One is to place a pragma on the routine to be called, for example on a "call specification" for a Java method. Then, calls from PL/SQL to this method will complain if the method is less restricted than the calling subprogram. For example:

```
CREATE OR REPLACE PACKAGE P1 IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
      PRAGMA RESTRICT_REFERENCES(F1,WNDS,TRUST);
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;

   PRAGMA RESTRICT_REFERENCES(F2,WNDS);
END;
```

```
CREATE OR REPLACE PACKAGE BODY P1 IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

Here, F2 can call F1, as F1 has been declared to be WNDS.

The other approach is to mark only the caller, which may then make a call to any subprogram without complaint. For example:

```
CREATE OR REPLACE PACKAGE P1a IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES(F2,WNDS,TRUST);
END;

CREATE OR REPLACE PACKAGE BODY P1a IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

Here, F2 can call F1 because while F2 is promised to be WNDS (because TRUST is specified), the body of F2 is not actually examined to determine if it truly satisfies the WNDS restriction. Because F2 is not examined, its call to F1 is allowed, even though there is no PRAGMA RESTRICT_REFERENCES for F1.

Differences between Static and Dynamic SQL Statements. Static INSERT, UPDATE, and DELETE statements do not violate RNDS if these statements do not explicitly read any database states, such as columns of a table. However, dynamic INSERT, UPDATE, and DELETE statements *always* violate RNDS, regardless of whether or not the statements explicitly read database states.

The following INSERT violates RNDS if it is executed dynamically, but it does *not* violate RNDS if it is executed statically.

```
INSERT INTO my_table values(3, 'SCOTT');
```

The following UPDATE always violates RNDS statically and dynamically, because it explicitly reads the column name of my_table.

```
UPDATE my_table SET id=777 WHERE name='SCOTT';
```

Overloading Packaged PL/SQL Functions PL/SQL lets you **overload** packaged (but not standalone) functions: 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. Therefore, a pragma that references the name of overloaded functions always applies to the nearest preceding function declaration.

In this 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);
 END;
```

Serially Reusable PL/SQL Packages

PL/SQL packages usually 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 for each user; rather, 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 RPC call from a client to a server, or an RPC call from a server to another server.

Package States

The state of a nonreusable package (one not marked `SERIALLY_REUSABLE`) persists for the lifetime of a session. A package **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, then Oracle Database creates a new *instantiation* 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 Database allocates memory or configures the instantiation object. Oracle Database looks for an available instantiation work area (which is allocated and configured) for this package in a least-recently used (LRU) pool in the 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?

Because 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, after they are done using the package.

With `SERIALLY_REUSABLE` packages, 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 of Serially Reusable Packages

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, then the body must have the serially reusable pragma, if its corresponding specification has the pragma; it cannot have the serially reusable pragma unless the specification also has the pragma.

Semantics of Serially Reusable Packages

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, then 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 run again.
- At the "end work" boundary, cleanup is done.
  - If any cursors were left open, then they are silently closed.
  - Some non-reusable secondary memory is freed (such as memory for collection variables or long `VARCHAR2`s).
  - This package instantiation is returned back to the pool of reusable instantiations kept for this package.
- Serially reusable packages cannot be accessed from database triggers or other PL/SQL subprograms that are called from SQL statements. If you try, then Oracle Database generates an error.

Examples of Serially Reusable Packages

Example 1: How Package Variables Act Across Call Boundaries This example has a serially reusable package specification (there is no body).

```
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;
```

This program prints:

```
5
```

> **Note:** If the package had not had the pragma `SERIALLY_REUSABLE`, the program would have printed '10'.

Example 2: How Package Variables Act Across Call Boundaries This example has both a package specification and package body, which are serially reusable.

```
CONNECT Scott/Tiger

DROP PACKAGE Sr_pkg;
CREATE OR REPLACE PACKAGE Sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   TYPE Str_table_type IS TABLE OF VARCHAR2(200) INDEX BY BINARY_INTEGER;
   Num     NUMBER        := 10;
   Str     VARCHAR2(200) := 'default-init-str';
   Str_tab STR_TABLE_TYPE;

    PROCEDURE Print_pkg;
    PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2);
END Sr_pkg;
CREATE OR REPLACE PACKAGE BODY Sr_pkg IS
   -- the body is required to have the pragma because the
  -- specification of this package has the pragma
  PRAGMA SERIALLY_REUSABLE;
   PROCEDURE Print_pkg IS
```

```
   BEGIN
      DBMS_OUTPUT.PUT_LINE('num: ' || Sr_pkg.Num);
      DBMS_OUTPUT.PUT_LINE('str: ' || Sr_pkg.Str);
      DBMS_OUTPUT.PUT_LINE('number of table elems: ' || Sr_pkg.Str_tab.Count);
      FOR i IN 1..Sr_pkg.Str_tab.Count LOOP
         DBMS_OUTPUT.PUT_LINE(Sr_pkg.Str_tab(i));
      END LOOP;
   END;
   PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2) IS
   BEGIN
   -- init the package globals
      Sr_pkg.Num := N;
      Sr_pkg.Str := V;
      FOR i IN 1..n LOOP
         Sr_pkg.Str_tab(i) := V || ' ' || i;
   END LOOP;
   -- print the package
   Print_pkg;
   END;
 END Sr_pkg;

SET SERVEROUTPUT ON;

Rem SR package access in a CALL:

BEGIN
   -- initialize and print the package
   DBMS_OUTPUT.PUT_LINE('Initing and printing pkg state..');
   Sr_pkg.Init_and_print_pkg(4, 'abracadabra');
   -- print it in the same call to the server.
   -- we should see the initialized values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the same CALL...');
   Sr_pkg.Print_pkg;
END;

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

REM SR package access in subsequent CALL:
BEGIN
   -- print the package in the next call to the server.
   -- We should that the package state is reset to the initial (default) values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the next CALL...');
   Sr_pkg.Print_pkg;
END;
Statement processed.
Printing package state in the next CALL...
num: 10
str: default-init-str
number of table elems: 0
```

Example 3: Open Cursors in Serially Reusable Packages at Call Boundaries 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). Also, 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  Because the cursor is closed - every time we fetch we
REM  will start at the first row again.

CONNECT Scott/Tiger
DROP PACKAGE  Sr_pkg;
DROP TABLE People;
CREATE TABLE People (Name VARCHAR2(20));
INSERT INTO  People  VALUES ('ET');
INSERT INTO  People  VALUES ('RAMBO');
CREATE OR REPLACE PACKAGE Sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   CURSOR C IS SELECT Name FROM People;
END Sr_pkg;
SQL> SET SERVEROUTPUT ON;
SQL>
CREATE OR REPLACE PROCEDURE Fetch_from_cursor IS
```

```
Name VARCHAR2(200);
BEGIN
   IF (Sr_pkg.C%ISOPEN) THEN
      DBMS_OUTPUT.PUT_LINE('cursor is already open.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('cursor is closed; opening now.');
      OPEN Sr_pkg.C;
   END IF;
   -- fetching from cursor.
   FETCH sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   FETCH Sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   -- Oops forgot to close the cursor (Sr_pkg.C).
   -- But, because it is a Serially Reusable pkg's cursor,
   -- it will be closed at the end of this CALL to the server.
END;
EXECUTE fetch_from_cursor;
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
```

Returning Large Amounts of Data from a Function

In a data warehousing environment, you might use a PL/SQL function to transform large amounts of data. Perhaps the data is passed through a series of transformations, each performed by a different function. PL/SQL table functions let you perform such transformations without significant memory overhead or the need to store the data in tables between each transformation stage. These functions can accept and return multiple rows, can return rows as they are ready rather than all at once, and can be parallelized.

In this technique:

- The producer function uses the `PIPELINED` keyword in its declaration.
- The producer function uses an `OUT` parameter that is a record, corresponding to a row in the result set.
- As each output record is completed, it is sent to the consumer function using the `PIPE ROW` keyword.
- The producer function ends with a `RETURN` statement that does not specify any return value.

- The consumer function or SQL statement uses the `TABLE` keyword to treat the resulting rows like a regular table.

For example:

```
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet PIPELINED
IS
  out_rec TickerType := TickerType(NULL,NULL,NULL);
  in_rec p%ROWTYPE;
BEGIN
  LOOP
-- Function accepts multiple rows through a REF CURSOR argument.
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
-- Return value is a record type that matches the table definition.
    out_rec.ticker := in_rec.Ticker;
    out_rec.PriceType := 'O';
    out_rec.price := in_rec.OpenPrice;
-- Once a result row is ready, we send it back to the calling program,
-- and continue processing.
    PIPE ROW(out_rec);
-- This function outputs twice as many rows as it receives as input.
    out_rec.PriceType := 'C';
    out_rec.Price := in_rec.ClosePrice;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
-- The function ends with a RETURN statement that does not specify any value.
  RETURN;
END;
/

-- Here we use the result of this function in a SQL query.
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));

-- Here we use the result of this function in a PL/SQL block.
DECLARE
  total NUMBER := 0;
  price_type VARCHAR2(1);
BEGIN
  FOR item IN (SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM
StockTable))))
  LOOP
-- Access the values of each output row.
-- We know the column names based on the declaration of the output type.
-- This computation is just for illustration.
```

```
    total := total + item.price;
    price_type := item.price_type;
  END LOOP;
END;
/
```

Coding Your Own Aggregate Functions

To analyze a set of rows and compute a result value, you can code your own aggregate function that works the same as a built-in aggregate like `SUM`:

- Define a SQL object type that defines these member functions:
  - `ODCIAggregateInitialize`
  - `ODCIAggregateIterate`
  - `ODCIAggregateMerge`
  - `ODCIAggregateTerminate`
- Code the member functions. In particular, `ODCIAggregateIterate` accumulates the result as it is called once for each row that is processed. Store any intermediate results using the attributes of the object type.
- Create the aggregate function, and associate it with the new object type.
- Call the aggregate function from SQL queries, DML statements, or other places that you might use the built-in aggregates. You can include typical options such as `DISTINCT` and `ALL` in the calls to the aggregate function.

> **See Also:** *Oracle Data Cartridge Developer's Guide* for complete details of this process and the requirements for the member functions

8

Calling External Procedures

In situations where a particular language does not provide the features you need, or when you want to reuse existing code written in another language, you can use code written in some other language by calling external procedures.

This chapter discusses the following topics:

Overview of Multi-Language Programs

Oracle Database lets you work in different languages:

- *PL/SQL*, as described in the *PL/SQL User's Guide and Reference*
- *C*, by means of the Oracle Call Interface (OCI), as described in the *Oracle Call Interface Programmer's Guide*
- *C* or *C++*, by means of the Pro*C/C++ precompiler, as described in the *Pro*C/C++ Programmer's Guide*
- *COBOL*, by means of the Pro*COBOL precompiler, as described in the *Pro*COBOL Programmer's Guide*
- *Visual Basic*, by means of Oracle Objects for OLE (OO4O), as described in *Oracle Objects for OLE Developer's Guide*.
- *Java*, by means of the JDBC Application Programmers Interface (API). See *Oracle Database Java Developer's Guide*.

How should you choose between these different implementation possibilities? Each of these languages offers different advantages: ease of use, the availability of programmers with specific expertise, the need for portability, and the existence of legacy code are powerful determinants.

The choice may narrow depending on how your application needs to work with Oracle Database:

- PL/SQL is a powerful development tool, specialized for SQL transaction processing.
- Some computation-intensive tasks are executed most efficiently in a lower level language, such as C.
- The need for portability, together with the need for security, may influence you to select Java.

Most significantly, from the point of view of performance, you should note that only PL/SQL and Java methods run within the address space of the server. C/C++ methods are dispatched as external procedures, and run on the server machine but outside the address space of the database server. Pro*COBOL and Pro*C are precompilers, and Visual Basic accesses Oracle Database through the OCI, which is implemented in C.

Taking all these factors into account suggests that there may be a number of situations in which you may need to implement your application in more than one language. For instance, the introduction of Java running within the address space of

the server suggest that you may want to import existing Java applications into the database, and then leverage this technology by calling Java functions from PL/SQL and SQL.

PL/SQL *external procedures* allow you to write C function calls as PL/SQL bodies. These C functions are callable directly from PL/SQL, and from SQL through PL/SQL procedure calls. The database provides a special-purpose interface, the call specification, that lets you call external procedures from other languages. While this service is designed for intercommunication between SQL, PL/SQL, C, and Java, it is accessible from any base language that can call these languages. For example, your procedure can be written in a language other than Java or C and still be usable by SQL or PL/SQL, as long as your procedure is callable by C. Therefore, if you have a candidate C++ procedure, you would use a C++ `extern "C"` statement in that procedure to make it callable by C.

This means that the strengths and capabilities of different languages are available to you, regardless of your programmatic environment. You are not restricted to one language with its inherent limitations. External procedures promote reusability and modularity because you can deploy specific languages for specific purposes.

What Is an External Procedure?

An **external procedure**, also sometimes referred to as an **external routine**, is a procedure stored in a dynamic link library (DLL), or libunit in the case of a Java class method. You register the procedure with the base language, and then call it to perform special-purpose processing.

For instance, when you work in PL/SQL, the language loads the library dynamically at runtime, and then calls the procedure as if it were a PL/SQL subprogram. These procedures participate fully in the current transaction and can call back to the database to perform SQL operations.

The procedures are loaded only when necessary, so memory is conserved. Because the decoupling of the call specification from its implementation body means that the procedures can be enhanced without affecting the calling programs.

External procedures let you:

- Isolate execution of client applications and processes from the database instance to ensure that any problems on the client side do not adversely impact the database.
- Move computation-bound programs from client to server where they execute faster (because they avoid the round-trips of network communication)

- Interface the database server with external systems and data sources
- Extend the functionality of the database server itself

Overview of The Call Specification for External Procedures

You publish external procedures through **call specifications**, which provide a superset of the `AS EXTERNAL` function through the `AS LANGUAGE` clause. `AS LANGUAGE` call specifications allow the publishing of external C procedures, but also Java class methods.

> **Note:** To support legacy applications, call specifications also allow you to publish with the `AS EXTERNAL` clause. For new application development, however, using the `AS LANGUAGE` clause is recommended.

In general, call specifications enable:

- Dispatching the appropriate C or Java target procedure
- Datatype conversions
- Parameter mode mappings
- Automatic memory allocation and cleanup
- Purity constraints to be specified, where necessary, for packaged functions called from SQL.
- Calling Java methods or C procedures from database triggers
- Location flexibility: you can put `AS LANGUAGE` call specifications in package or type specifications, or package (or type) bodies to optimize performance and hide implementation details

To use an already-existing program as an external procedure, load, publish, and then call it.

Loading External Procedures

To make your external C procedures or Java methods available to PL/SQL, you must first load them. The manner of doing this depends upon whether the procedure is written in C or Java.

Loading Java Class Methods

One way to load Java programs is to use the CREATE JAVA statement, which you can execute interactively from SQL*Plus. When implicitly invoked by the CREATE JAVA statement, the Java Virtual Machine (JVM)] library manager loads Java binaries (.class files) and resources from local BFILEs or LOB columns into RDBMS libunits.

Suppose a compiled Java class is stored in the following operating system file:

```
/home/java/bin/Agent.class
```

Creating a class libunit in schema scott from file Agent.class requires two steps: First, create a directory object on the server's file system. The name of the directory object is an alias for the directory path leading to Agent.class.

To create the directory object, you must grant user scott the CREATE ANY DIRECTORY privilege, then execute the CREATE DIRECTORY statement, as follows:

```
CONNECT System/Manager
GRANT CREATE ANY DIRECTORY TO Scott IDENTIFIED BY Tiger;
CONNECT Scott/Tiger
CREATE DIRECTORY Bfile_dir AS '/home/java/bin';
```

You are ready to create the class libunit, as follows:

```
CREATE JAVA CLASS USING BFILE (Bfile_dir, 'Agent.class');
```

The name of the libunit is derived from the name of the class.

Alternatively, you can use the command-line utility LoadJava. This uploads Java binaries and resources into a system-generated database table, then uses the CREATE JAVA statement to load the Java files into RDBMS libunits. You can upload Java files from file systems, Java IDEs, intranets, or the Internet.

Loading External C Procedures

In order to set up your database configuration to use external procedures that are written in C, or can be called from C applications, you and your database administrator must take the following steps:

Note:

- This feature is available only on platforms that support dynamically linked libraries (DLLs) or dynamically loadable shared libraries such as Solaris `.so` libraries.
- The external procedure agent can call procedures in any library that complies with the calling standard used. The supported calling standard is C. See "CALLING STANDARD" on page 8-12 for more information on the calling standard sub clause used with external procedures in PL/SQL.

Step 1 Set Up the Environment

Your database administrator must perform the following tasks to configure your database to use external procedures that are written in C, or can be called from C applications:

- Set configuration parameters for the agent, named `extproc` by default, in the configuration files `tnsnames.ora` and `listener.ora`. This establishes the connection for the external procedure agent when the database is started.
- Start a listener process exclusively for external procedures.

  The Listener sets a few required environment variables (such as `ORACLE_HOME`, `ORACLE_SID`, and `LD_LIBRARY_PATH`) for the external procedure agent. It can also define specific environment variables in the `ENVS` section of its `listener.ora` entry, and these variables are passed to the agent process. Otherwise, it provides the agent with a "clean" environment. The environment variables set for the agent are independent of those set for the client and server. Therefore, external procedures, which run in the agent process, cannot read environment variables set for the client or server processes.

  Note: It is possible for you to set and read environment variables themselves by using the standard C procedures `setenv()` and `getenv()`, respectively. Environment variables, set this way, are specific to the agent process, which means that they can be read by all functions executed in that process, but not by any other process running on the same host.

- Determine whether the agent for your external procedure will run in *dedicated mode* (the default mode) or *multithreaded mode*. In dedicated mode, one

"dedicated" agent is launched for each user. In multithreaded mode, a single *multithreaded agent* is launched. The multithreaded agent handles calls using different threads for different users. In a configuration where many users will call the external procedures, using a multithreaded agent is recommended to conserve system resources.

If the agent will run in dedicated mode, additional configuration of the agent process is not necessary.

If the agent will run in multithreaded mode, your database administrator must configure the database system to start the agent in multithreaded mode (as a multithreaded agent). This is done using the agent control utility `agtctl`. For example, start the agent using the agent control utility startup command:

```
agtctl startup extproc agent_sid
```

where *`agent_sid`* is the system identifier which this agent will service. An entry for this system identifier is typically added as an entry in the file `tnsnames.ora`. Details on the agent control utility are in the *Oracle Database Heterogeneous Connectivity Administrator's Guide*.

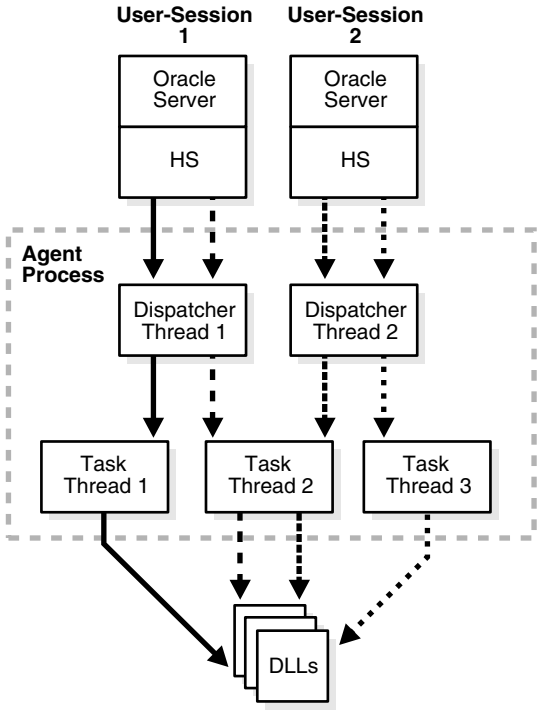
Note:

- If you use a multithreaded agent, the library you call must be thread safe—to avoid errors such as a corrupt call stack.
- The database server, the agent process, and the listener process that spawns the agent process must all reside on the same host.
- By default, the agent process runs on the same database instance as your main application. In situations where reliability is critical, you might want to run the agent process for the external procedure on a separate database instance (still on the same host), so that any problems in the agent do not affect the primary database server. To do so, specify the separate database instance using a database link.

Figure 8–1 illustrates the architecture of the multithreaded external procedure agent. User sessions 1 and 2 issue requests for callouts to functions in some DLLs. These requests get serviced through heterogeneous services to the multithreaded `extproc` agent. These requests get handled by the agent's dispatcher threads, which then pass them on to the task threads. The task thread that is actually handling a request is responsible for loading the respective DLL and calling the function therein.

- All requests from a user session get handled by the same dispatcher thread. For example, dispatcher 1 handles communication with user session 1, and dispatcher 2 handles communication with user session 2. This is the case for the lifetime of the session.
- The individual requests can, however, be serviced by different task threads. For example, task thread 1 could handle the request from user session 1 at one time, and handle the request from user session 2 at another time.

Figure 8–1 Multithreaded External Procedure Agent Architecture



See Also:

- *Oracle Database Heterogeneous Connectivity Administrator's Guide* for details on configuring the agent process for the external procedure to use multithreaded mode, architecture, and for additional details on multithreaded agents
- *Oracle Database Administrator's Guide*. for details on managing processes for external procedures

Step 2 Identify the DLL

In this context, a DLL is any dynamically loadable operating-system file that stores external procedures.

For security reasons, your DBA controls access to the DLL. Using the `CREATE LIBRARY` statement, the DBA creates a schema object called an **alias library**, which represents the DLL. Then, if you are an authorized user, the DBA grants you `EXECUTE` privileges on the alias library. Alternatively, the DBA may grant you `CREATE ANY LIBRARY` privileges, in which case you can create your own alias libraries using the following syntax:

```
CREATE LIBRARY [schema_name.]library_name
  {IS | AS} 'file_path'
  [AGENT 'agent_link'];
```

It is recommended that you specify the full path to the DLL, rather than just the DLL name. In the following example, you create alias library `c_utils`, which represents DLL `utils.so`:

```
CREATE LIBRARY C_utils AS '/DLLs/utils.so';
```

To allow flexibility in specifying the DLLs, you can specify the root part of the path as an environment variable using the notation `${VAR_NAME}`, and set up that variable in the `ENVS` section of the `listener.ora` entry.

In the following example, the agent specified by the name `agent_link` is used to run any external procedure in the library `C_Utils`. The environment variable `EP_LIB_HOME` is expanded by the agent to the appropriate path for that instance, such as `/usr/bin/dll`. Variable `EP_LIB_HOME` must be set in the file `listener.ora`, for the agent to be able to access it.

```
create or replace database link agent_link using 'agent_tns_alias';
create or replace library C_utils is
  '${EP_LIB_HOME}/utils.so' agent 'agent_link';
```

For security reasons, EXTPROC, by default, will only load DLLs that are in directory $ORACLE_HOME/bin or $ORACLE_HOME/lib. Also, only local sessions—that is, Oracle Database client processes that are running on the same machine—are allowed to connect to EXTPROC.

To load DLLs from other directories, the environment variable EXTPROC_DLLS should be set. The value for this environment variable is a colon- separated (:) list of DLL names qualified with the complete path. For example:

```
EXTPROC_DLLS=/private1/home/scott/dll/myDll.so:/private1/home/scott/dll/newDll.so
```

The preferred method to set this environment variable is through the ENVS parameter in the file listener.ora. Refer to the Oracle Net manual for more information on the EXTPROC feature.

Note the following:

- On a Windows system, you would specify the path using a drive letter and backslash characters (\) in the path.
- This technique is not applicable for VMS systems, where the ENVS section of listener.ora is not supported.

Step 3 Publish the External Procedure

You find or write a new external C procedure, then add it to the DLL. When the procedure is in the DLL, you publish it using the call specification mechanism described in the following section.

Publishing External Procedures

Oracle Database can only use external procedures that are published through a call specification, which maps names, parameter types, and return types for your Java class method or C external procedure to their SQL counterparts. It is written like any other PL/SQL stored subprogram except that, in its body, instead of declarations and a BEGIN ... END block, you code the AS LANGUAGE clause.

The AS LANGUAGE clause specifies:

- Which language the procedure is written in.
- For a Java method:
  - The signature of the Java method.
- For a C procedure:

- The alias library corresponding to the DLL for a C procedure.
- The name of the C procedure in a DLL.
- Various options for specifying how parameters are passed.
- Which parameter (if any) holds the name of the external procedure agent, for running the procedure on a different machine.

You begin the declaration using the normal `CREATE OR REPLACE` syntax for a procedure, function, package specification, package body, type specification, or type body.

The call specification follows the name and parameter declarations. Its syntax is:

```
{IS | AS} LANGUAGE {C | JAVA}
```

> **Note:** Oracle Database uses a PL/SQL variant of the ANSI SQL92 External Procedure, but replaces the ANSI keyword `AS EXTERNAL` with this call specification syntax. This new syntax, first introduced for Java class methods, has been extended to C procedures.

This is then followed by either:

```
NAME  java_string_literal_name
```

Where *java_string_literal_name* is the signature of your Java method, or by:

```
LIBRARY library_name
[NAME c_string_literal_name]
[WITH CONTEXT]
[PARAMETERS (external_parameter[, external_parameter]...)];
```

Where `library_name` is the name of your alias library, `c_string_literal_name` is the name of your external C procedure, and `external_parameter` stands for:

```
{  CONTEXT
 | SELF [{TDO | property}]
 | {parameter_name | RETURN} [property] [BY REFERENCE] [external_datatype]}
```

`property` stands for:

```
{INDICATOR [{STRUCT | TDO}] | LENGTH | DURATION | MAXLEN | CHARSETID |
CHARSETFORM}
```

Note: Unlike Java, C does not understand SQL types; therefore, the syntax is more intricate

The AS LANGUAGE Clause for Java Class Methods

The `AS LANGUAGE` clause is the interface between PL/SQL and a Java class method.

The AS LANGUAGE Clause for External C Procedures

The following subclauses tell PL/SQL where to locate the external C procedure, how to call it, and what to pass to it. Only the `LIBRARY` subclause is required.

LIBRARY

Specifies a local alias library. (You cannot use a database link to specify a remote library.) The library name is a PL/SQL identifier. Therefore, if you enclose the name in double quotes, then it becomes case sensitive. (By default, the name is stored in upper case.) You must have `EXECUTE` privileges on the alias library.

NAME

Specifies the external C procedure to be called. If you enclose the procedure name in double quotes, then it becomes case sensitive. (By default, the name is stored in upper case.) If you omit this subclause, then the procedure name defaults to the upper-case name of the PL/SQL subprogram.

Note: The terms `LANGUAGE` and `CALLING STANDARD` apply only to the superseded `AS EXTERNAL` clause.

LANGUAGE

Specifies the third-generation language in which the external procedure was written. If you omit this subclause, then the language name defaults to C.

CALLING STANDARD

Specifies the calling standard under which the external procedure was compiled. The supported calling standard is C. If you omit this subclause, then the calling standard defaults to C.

WITH CONTEXT

Specifies that a context pointer will be passed to the external procedure. The context data structure is opaque to the external procedure but is available to service procedures called by the external procedure.

PARAMETERS

Specifies the positions and datatypes of parameters passed to the external procedure. It can also specify parameter properties, such as current length and maximum length, and the preferred parameter passing method (by value or by reference).

AGENT IN

Specifies which parameter holds the name of the agent process that should run this procedure. This is intended for situations where external procedure agents are run using multiple agent processes, to ensure robustness if the agent process of one external procedure fails. You can pass the name of the agent process (corresponding to the name of a database link), and if `tnsnames.ora` and `listener.ora` are set up properly across both instances, the external procedure is invoked on the other instance. Both instances must be on the same host.

This is similar to the `AGENT` clause of the `CREATE LIBRARY` statement; specifying the value at runtime through `AGENT IN` allows greater flexibility.

When the agent name is specified this way, it overrides any agent name declared in the alias library. If no agent name is specified, the default is the `extproc` agent on the same instance as the calling program.

Publishing Java Class Methods

Java classes and their methods are stored in RDBMS libunits in which you can load Java sources, binaries and resources using the `LOADJAVA` utility or the `CREATEJAVA` SQL statements. Libunits can be considered analogous to DLLs written, for example, in C—although they map one-to-one with Java classes, whereas DLLs can contain more than one procedure.

The `NAME`-clause string uniquely identifies the Java method. The PL/SQL function or procedure and Java must correspond with regard to parameters. If the Java method takes no parameters, then you must code an empty parameter list for it.

When you load Java classes into the RDBMS, they are not published to SQL automatically. This is because the methods of many Java classes are called only from other Java classes, or take parameters for which there is no appropriate SQL type.

Suppose you want to publish the following Java method named `J_calcFactorial`, which returns the factorial of its argument:

```
package myRoutines.math;
public class Factorial {
   public static int J_calcFactorial (int n) {
      if (n == 1) return 1;
      else return n * J_calcFactorial(n - 1);
   }
}
```

The following call specification publishes Java method `J_calcFactorial` as PL/SQL stored function `plsToJavaFac_func`, using SQL*Plus:

```
CREATE OR REPLACE FUNCTION Plstojavafac_func (N NUMBER) RETURN NUMBER AS
   LANGUAGE JAVA
   NAME 'myRoutines.math.Factorial.J_calcFactorial(int) return int';
```

Publishing External C Procedures

In the following example, you write a PL/SQL standalone function named `plsCallsCdivisor_func` that publishes C function `Cdivisor_func` as an external function:

```
CREATE OR REPLACE FUNCTION Plscallscdivisor_func (
/* Find greatest common divisor of x and y: */
   x     BINARY_INTEGER,
   y     BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
   LIBRARY C_utils
   NAME "Cdivisor_func"; /* Quotation marks preserve case. */
```

Locations of Call Specifications

For both Java class methods and external C procedures, call specifications can be specified in any of the following locations:

- Standalone PL/SQL Procedures and Functions
- PL/SQL Package Specifications

- PL/SQL Package Bodies
- Object Type Specifications
- Object Type Bodies

> **Note:** Under Oracle Database version 8.0, `AS EXTERNAL` call specifications could not be placed in package or type bodies.

We have already shown an example of call specification located in a standalone PL/SQL function. Here are some examples showing some of the other locations.

> **Note:** In the following examples, the `AUTHID` and `SQL_NAME_RESOLVE` clauses may or may not be required to fully stipulate a call specification.
>
> See the *PL/SQL User's Guide and Reference* and the *Oracle Database SQL Reference* for more information.

Example: Locating a Call Specification in a PL/SQL Package

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
   PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
END;
```

Example: Locating a Call Specification in a PL/SQL Package Body

```
CREATE OR REPLACE PACKAGE Demo_pack
   AUTHID CURRENT_USER
AS
   PROCEDURE plsToC_demoExternal_proc(x BINARY_INTEGER, y VARCHAR2, z DATE);
END;

CREATE OR REPLACE PACKAGE BODY Demo_pack
   SQL_NAME_RESOLVE CURRENT_USER
AS
   PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
```

```
   AS LANGUAGE JAVA
      NAME 'pkg1.class4.methodProc1(int,java.lang.String,java.sql.Date)';
END;
```

Example: Locating a Call Specification in an Object Type Specification

> **Note:** You may need to set up the following data structures for certain examples to work:
>
> ```
> CONN SYS/CHANGE_ON_INSTALL AS SYSDBA;
> GRANT CREATE ANY LIBRARY TO scott;
> CONNECT scott/tiger
> CREATE OR REPLACE LIBRARY SOMELIB AS '/tmp/lib.so';
> ```

```
CREATE OR REPLACE TYPE Demo_typ
AUTHID DEFINER
AS OBJECT
   (Attribute1   VARCHAR2(2000), SomeLib varchar2(20),
   MEMBER PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z
DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
    --  PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE)
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE, SELF)
);
```

Example: Locating a Call Specification in an Object Type Body

```
CREATE OR REPLACE TYPE Demo_typ
AUTHID CURRENT_USER
AS OBJECT
   (attribute1 NUMBER,
   MEMBER PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z
DATE)
);

CREATE OR REPLACE TYPE BODY Demo_typ
AS
   MEMBER PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z
DATE)
   AS LANGUAGE JAVA
```

```
      NAME 'pkg1.class4.J_demoExternal(int,java.lang.String,java.sql.Date)';
END;
```

Example: Java with AUTHID

Here is an example of a publishing a Java class method in a standalone PL/SQL subprogram.

```
CREATE OR REPLACE PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y
VARCHAR2, z DATE)
   AUTHID CURRENT_USER
AS LANGUAGE JAVA
   NAME 'pkg1.class4.methodProc1(int,java.lang.String,java.sql.Date)';
```

Example: C with Optional AUTHID

Here is an example of `AS EXTERNAL` publishing a C procedure in a standalone PL/SQL program, in which the `AUTHID` clause is optional. This maintains compatibility with the external procedures of Oracle Database version 8.0.

```
CREATE OR REPLACE PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y
VARCHAR2, z DATE)
AS
   EXTERNAL
   LANGUAGE C
   NAME "C_demoExternal"
   LIBRARY SomeLib
   WITH CONTEXT
   PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
```

Example: Mixing Call Specifications in a Package

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
   PROCEDURE plsToC_InBodyOld_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToC_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToJ_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);

   PROCEDURE plsToJ_InSpec_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   IS LANGUAGE JAVA
      NAME 'pkg1.class4.J_InSpec_meth(int,java.lang.String,java.sql.Date)';

PROCEDURE C_InSpec_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
```

```
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
END;

CREATE OR REPLACE PACKAGE BODY Demo_pack
AS
PROCEDURE plsToC_InBodyOld_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS EXTERNAL
      LANGUAGE C
      NAME "C_InBodyOld"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);

PROCEDURE plsToC_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_InBody"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
PROCEDURE plsToJ_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   IS LANGUAGE JAVA
      NAME 'pkg1.class4.J_InBody_meth(int,java.lang.String,java.sql.Date)';
END;
```

Passing Parameters to External C Procedures with Call Specifications

Call specifications allows a mapping between PL/SQL and C datatypes. See "Specifying Datatypes" for datatype mappings.

Passing parameters to an external C procedure is complicated by several circumstances:

- The available set of PL/SQL datatypes does not correspond one-to-one with the set of C datatypes.

- Unlike C, PL/SQL includes the RDBMS concept of nullity. Therefore, PL/SQL parameters can be NULL, whereas C parameters cannot.
- The external procedure might need the current length or maximum length of CHAR, LONG RAW, RAW, and VARCHAR2 parameters.
- The external procedure might need character set information about CHAR, VARCHAR2, and CLOB parameters.
- PL/SQL might need the current length, maximum length, or null status of values returned by the external procedure.

In the following sections, you learn how to specify a parameter list that deals with these circumstances.

> **Note:** The maximum number of parameters that you can pass to a C external procedure is 128. However, if you pass float or double parameters by value, then the maximum is less than 128. How much less depends on the number of such parameters and your operating system. To get a rough estimate, count each float or double passed by value as two parameters.

Specifying Datatypes

Do not pass parameters to an external procedure directly. Instead, pass them to the PL/SQL subprogram that published the external procedure. Therefore, you must specify PL/SQL datatypes for the parameters. PL/SQL datatypes map to default external datatypes, as shown in Table 8–1.

Table 8–1 Parameter Datatype Mappings

PL/SQL Datatype	Supported External Types	Default External Type
BINARY_INTEGER	[UNSIGNED] CHAR	INT
BOOLEAN	[UNSIGNED] SHORT	
PLS_INTEGER	[UNSIGNED] INT	
	[UNSIGNED] LONG	
	SB1, SB2, SB4	
	UB1, UB2, UB4	
	SIZE_T	

Table 8–1 (Cont.) Parameter Datatype Mappings

PL/SQL Datatype	Supported External Types	Default External Type
NATURAL[1] NATURALN[1] POSITIVE[1] POSITIVEN[1] SIGNTYPE[1]	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	UNSIGNED INT
FLOAT REAL	FLOAT	FLOAT
DOUBLE PRECISION	DOUBLE	DOUBLE
CHAR CHARACTER LONG NCHAR NVARCHAR2 ROWID VARCHAR VARCHAR2	STRING OCISTRING	STRING
LONG RAW RAW	RAW OCIRAW	RAW
BFILE BLOB CLOB NCLOB	OCILOBLOCATOR	OCILOBLOCATOR
NUMBER DEC[1] DECIMAL[1] INT[1] INTEGER[1] NUMERIC[1] SMALLINT[1]	OCINUMBER	OCINUMBER
DATE	OCIDATE	OCIDATE
TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime	OCIDateTime

Table 8–1 (Cont.) Parameter Datatype Mappings

PL/SQL Datatype	Supported External Types	Default External Type
INTERVAL DAY TO SECOND INTERVAL YEAR TO MONTH	OCIInterval	OCIInterval
composite object types: ADTs	dvoid	dvoid
composite object types: collections (varrays, nested tables)	OCICOLL	OCICOLL

[1] This PL/SQL type will only compile if you use AS EXTERNAL in your callspec.

External Datatype Mappings

Each external datatype maps to a C datatype, and the datatype conversions are performed implicitly. To avoid errors when declaring C prototype parameters, refer to Table 8–2, which shows the C datatype to specify for a given external datatype and PL/SQL parameter mode. For example, if the external datatype of an OUT parameter is STRING, then specify the datatype char * in your C prototype.

Table 8–2 External Datatype Mappings

External Datatype Corresponding to PL/SL Type	If Mode is IN or RETURN, Specify in C Prototype...	If Mode is IN by Reference or RETURN by Reference, Specify in C Prototype...	If Mode is IN OUT or OUT, Specify in C Prototype...
CHAR	char	char *	char *
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
CHAR	char	char *	char *

Table 8–2 (Cont.) External Datatype Mappings

External Datatype Corresponding to PL/SL Type	If Mode is IN or RETURN, Specify in C Prototype...	If Mode is IN by Reference or RETURN by Reference, Specify in C Prototype...	If Mode is IN OUT or OUT, Specify in C Prototype...
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
SIZE_T	size_t	size_t *	size_t *
SB1	sb1	sb1 *	sb1 *
UB1	ub1	ub1 *	ub1 *
SB2	sb2	sb2 *	sb2 *
UB2	ub2	ub2 *	ub2 *
SB4	sb4	sb4 *	sb4 *
UB4	ub4	ub4 *	ub4 *
FLOAT	float	float *	float *
DOUBLE	double	double *	double *
STRING	char *	char *	char *
RAW	unsigned char *	unsigned char *	unsigned char *
OCILOBLOCATOR	OCILobLocator *	OCILobLocator **	OCILobLocator **
OCINUMBER	OCINumber *	OCINumber *	OCINumber *
OCISTRING	OCIString *	OCIString *	OCIString *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *

Table 8–2 (Cont.) External Datatype Mappings

External Datatype Corresponding to PL/SL Type	If Mode is IN or RETURN, Specify in C Prototype...	If Mode is IN by Reference or RETURN by Reference, Specify in C Prototype...	If Mode is IN OUT or OUT, Specify in C Prototype...
OCIDATE	OCIDate *	OCIDate *	OCIDate *
OCICOLL	OCIColl * or OCIArray *, or OCITable *	OCIColl ** or OCIArray **, or OCITable **	OCIColl ** or OCIArray **, or OCITable **
OCITYPE	OCIType *	OCIType *	OCIType *
TDO	OCIType *	OCIType *	OCIType *
ADT (final types)	dvoid*	dvoid*	dvoid*
ADT (non-final types)	dvoid*	dvoid*	dvoid**

Composite object types are not self describing. Their description is stored in a **Type Descriptor Object** (TDO). Objects and indicator structs for objects have no predefined OCI datatype, but must use the datatypes generated by Oracle Database's **Object Type Translator** (OTT). The optional TDO argument for INDICATOR, and for composite objects, in general, has the C datatype, OCIType *.

OCICOLL for REF and collection arguments *is* optional and only exists for the sake of completeness. You cannot map REFs or collections onto any other datatype and vice versa.

BY VALUE/REFERENCE for IN and IN OUT Parameter Modes

If you specify BY VALUE, then scalar IN and RETURN arguments are passed by value (which is also the default). Alternatively, you may have them passed by reference by specifying BY REFERENCE.

By default, or if you specify BY REFERENCE, then scalar IN OUT, and OUT arguments are passed by reference. Specifying BY VALUE for IN OUT, and OUT arguments is not supported for C. The usefulness of the BY REFERENCE/VALUE clause is restricted to external datatypes that are, by default, passed by value. This is true for IN, and RETURN arguments of the following external types:

```
[UNSIGNED] CHAR
```

```
[UNSIGNED] SHORT
[UNSIGNED] INT
[UNSIGNED] LONG
SIZE_T
SB1
SB2
SB4
UB1
UB2
UB4
FLOAT
DOUBLE
```

All `IN` and `RETURN` arguments of external types not on this list, all `IN OUT` arguments, and all `OUT` arguments are passed by reference.

The PARAMETERS Clause

Generally, the PL/SQL subprogram that publishes an external procedure declares a list of formal parameters, as the following example shows:

> **Note:** You may need to set up the following data structures for certain examples to work:
>
> ```
> CREATE LIBRARY MathLib AS '/tmp/math.so';
> ```

```
CREATE OR REPLACE FUNCTION Interp_func (
/* Find the value of y at x degrees using Lagrange interpolation: */
   x    IN FLOAT,
   y    IN FLOAT)
RETURN FLOAT AS
   LANGUAGE C
   NAME "Interp_func"
   LIBRARY MathLib;
```

Each formal parameter declaration specifies a name, parameter mode, and PL/SQL datatype (which maps to the default external datatype). That might be all the information the external procedure needs. If not, then you can provide more information using the `PARAMETERS` clause, which lets you specify the following:

- Nondefault external datatypes
- The current or maximum length of a parameter

- NULL/NOT NULL indicators for parameters
- Character set IDs and forms
- The position of parameters in the list
- How IN parameters are passed (by value or by reference)

If you decide to use the PARAMETERS clause, keep in mind:

- For every formal parameter, there must be a corresponding parameter in the PARAMETERS clause.
- If you include the WITH CONTEXT clause, then you must specify the parameter CONTEXT, which shows the position of the context pointer in the parameter list.
- If the external procedure is a function, then you may specify the RETURN parameter, but it must be in the last position. If RETURN is not specified, the default external type is used.

Overriding Default Datatype Mapping

In some cases, you can use the PARAMETERS clause to override the default datatype mappings. For example, you can re-map the PL/SQL datatype BOOLEAN from external datatype INT to external datatype CHAR.

Specifying Properties

You can also use the PARAMETERS clause to pass additional information about PL/SQL formal parameters and function results to an external procedure. Do this by specifying one or more of the following properties:

```
INDICATOR [{STRUCT | TDO}]
LENGTH
DURATION
MAXLEN
CHARSETID
CHARSETFORM
SELF
```

Table 8–3 shows the allowed and the default external datatypes, PL/SQL datatypes, and PL/SQL parameter modes allowed for a given property. Notice that MAXLEN (used to specify data returned from C back to PL/SQL) cannot be applied to an IN parameter.

Table 8–3 Properties and Datatypes

Property	Allowed External Types (C)	Default External Type (C)	Allowed PL/SQL Types	Allowed PL/SQL Modes	Default PL/SQL Passing Method
INDICATOR	SHORT	SHORT	all scalars	IN	BY VALUE
				IN OUT	BY REFERENCE
				OUT	BY REFERENCE
				RETURN	BY REFERENCE
LENGTH	[UNSIGNED] SHORT	INT	CHAR	IN	BY VALUE
	[UNSIGNED] INT		LONG RAW	IN OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	OUT	BY REFERENCE
			VARCHAR2	RETURN	BY REFERENCE
MAXLEN	[UNSIGNED] SHORT	INT	CHAR	IN OUT	BY REFERENCE
	[UNSIGNED] INT		LONG RAW	OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	RETURN	BY REFERENCE
			VARCHAR2		
CHARSETID	UNSIGNED SHORT	UNSIGNED INT	CHAR	IN	BY VALUE
CHARSETFORM	UNSIGNED INT		CLOB	IN OUT	BY REFERENCE
	UNSIGNED LONG		VARCHAR2	OUT	BY REFERENCE
				RETURN	BY REFERENCE

In the following example, the PARAMETERS clause specifies properties for the PL/SQL formal parameters and function result:

```
CREATE OR REPLACE FUNCTION plsToCparse_func  (
   x   IN BINARY_INTEGER,
   Y   IN OUT CHAR)
RETURN CHAR AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_parse"
   PARAMETERS (
      x,            -- stores value of x
      x INDICATOR,  -- stores null status of x
      y,            -- stores value of y
      y LENGTH,     -- stores current length of y
      y MAXLEN,     -- stores maximum length of y
      RETURN INDICATOR,
      RETURN);
```

With this PARAMETERS clause, the C prototype becomes:

```
char * C_parse(int x, short x_ind, char *y, int *y_len,
               int *y_maxlen, short *retind);
```

The additional parameters in the C prototype correspond to the INDICATOR (for x), LENGTH (of y), and MAXLEN (of y), as well as the INDICATOR for the function result in the PARAMETERS clause. The parameter RETURN corresponds to the C function identifier, which stores the result value.

INDICATOR

An INDICATOR is a parameter whose value indicates whether or not another parameter is NULL. PL/SQL does not need indicators, because the RDBMS concept of nullity is built into the language. However, an external procedure might need to know if a parameter or function result is NULL. Also, an external procedure might need to signal the server that a returned value is actually a NULL, and should be treated accordingly.

In such cases, you can use the property INDICATOR to associate an indicator with a formal parameter. If the PL/SQL subprogram is a function, then you can also associate an indicator with the function result, as shown earlier.

To check the value of an indicator, you can use the constants OCI_IND_NULL and OCI_IND_NOTNULL. If the indicator equals OCI_IND_NULL, then the associated parameter or function result is NULL. If the indicator equals OCI_IND_NOTNULL, then the parameter or function result is not NULL.

For IN parameters, which are inherently read-only, INDICATOR is passed by value (unless you specify BY REFERENCE) and is read-only (even if you specify BY REFERENCE). For OUT, IN OUT, and RETURN parameters, INDICATOR is passed by reference by default.

The INDICATOR can also have a STRUCT or TDO option. Because specifying INDICATOR as a property of an object is not supported, and because arguments of objects have complete indicator structs instead of INDICATOR scalars, you must specify this by using the STRUCT option. You must use the type descriptor object (TDO) option for composite objects and collections,

LENGTH and MAXLEN

In PL/SQL, there is no standard way to indicate the length of a RAW or string parameter. However, in many cases, you want to pass the length of such a parameter to and from an external procedure. Using the properties LENGTH and MAXLEN, you can specify parameters that store the current length and maximum length of a formal parameter.

> **Note:** With a parameter of type RAW or LONG RAW, you must use the property LENGTH. Also, if that parameter is IN OUT and NULL or OUT and NULL, then you must set the length of the corresponding C parameter to zero.

For IN parameters, LENGTH is passed by value (unless you specify BY REFERENCE) and is read-only. For OUT, IN OUT, and RETURN parameters, LENGTH is passed by reference.

As mentioned earlier, MAXLEN does not apply to IN parameters. For OUT, IN OUT, and RETURN parameters, MAXLEN is passed by reference and is read-only.

CHARSETID and CHARSETFORM

Oracle Database provides globalization support, which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments.

By default, if the server and agent use the exact same $ORACLE_HOME value, the agent uses the same globalization support settings as the server (including any settings that have been specified with ALTER SESSION commands).

If the agent is running in a separate $ORACLE_HOME (even if the same location is specified by two different aliases or symbolic links), the agent uses the same globalization support settings as the server except for the character set; the default character set for the agent is defined by the NLS_LANG and NLS_NCHAR environment settings for the agent.

The properties CHARSETID and CHARSETFORM identify the nondefault character set from which the character data being passed was formed. With CHAR, CLOB, and VARCHAR2 parameters, you can use CHARSETID and CHARSETFORM to pass the character set ID and form to the external procedure.

For IN parameters, CHARSETID and CHARSETFORM are passed by value (unless you specify BY REFERENCE) and are read-only (even if you specify BY REFERENCE). For OUT, IN OUT, and RETURN parameters, CHARSETID and CHARSETFORM are passed by reference and are read-only.

The OCI attribute names for these properties are OCI_ATTR_CHARSET_ID and OCI_ATTR_CHARSET_FORM.

See Also: *Oracle Call Interface Programmer's Guide* and the *Oracle Database Globalization Support Guide* for more information about using national language data with the OCI

Repositioning Parameters

Remember, each formal parameter of the external procedure must have a corresponding parameter in the `PARAMETERS` clause. Their positions can differ, because PL/SQL associates them by name, not by position. However, the `PARAMETERS` clause and the C prototype for the external procedure must have the same number of parameters, and they must be in the same order.

Using SELF

`SELF` is the always-present argument of an object type's member function or procedure, namely the object instance itself. In most cases, this argument is implicit and is not listed in the argument list of the PL/SQL procedure. However, `SELF` must be explicitly specified as an argument of the `PARAMETERS` clause.

For example, assume that a user wants to create a `Person` object, consisting of a person's name and date of birth, and then further a table of this object type. The user would eventually like to determine the age of each `Person` object in this table.

Note: You may need to set up data structures similar to the following for certain examples to work:

```
CONNECT system/manager
GRANT CONNECT,RESOURCE,CREATE LIBRARY TO scott IDENTIFIED BY
tiger;
CONNECT scott/tiger
CREATE OR REPLACE LIBRARY agelib UNTRUSTED IS
  '/tmp/scott1.so';.
```

This example is only for Solaris; other libraries and include paths might be needed for other platforms.

In SQL*Plus, the `Person` object type can be created by:

```
CREATE OR REPLACE TYPE Person1_typ AS OBJECT
( Name      VARCHAR2(30),
  B_date    DATE,
  MEMBER FUNCTION calcAge_func RETURN NUMBER)
);
```

Normally, the member function would be implemented in PL/SQL, but for this example, we make it an external procedure. To realize this, the body of the member function is declared as follows:

```
CREATE OR REPLACE TYPE BODY Person1_typ AS
  MEMBER FUNCTION calcAge_func RETURN NUMBER
  AS LANGUAGE C
  NAME "age"
  LIBRARY agelib
  WITH CONTEXT
  PARAMETERS
  ( CONTEXT,
    SELF,
    SELF INDICATOR STRUCT,
    SELF TDO,
    RETURN INDICATOR
  );
END;
```

Notice that the `calcAge_func` member function does not take any arguments, but only returns a number. A member function is always invoked on an instance of the associated object type. The object instance itself always is an implicit argument of the member function. To refer to the implicit argument, the `SELF` keyword is used. This is incorporated into the external procedure syntax by supporting references to `SELF` in the parameters clause.

The matching table is created and populated.

```
CREATE TABLE Person_tab OF Person1_typ;

INSERT INTO Person_tab VALUES
   ('SCOTT', TO_DATE('14-MAY-85'));

INSERT INTO Person_tab VALUES
   ('TIGER', TO_DATE('22-DEC-71'));
```

Finally, we retrieve the information of interest from the table.

```
SELECT p.name, p.b_date, p.calcAge_func() FROM Person_tab p;

NAME                           B_DATE    P.CALCAGE_
------------------------------ --------- ----------
SCOTT                          14-MAY-85          0
TIGER                          22-DEC-71          0
```

The following is sample C code that implements the `external` member function and the Object-Type-Translator (OTT)-generated `struct` definitions:

```
#include <oci.h>

struct PERSON
{
    OCIString   *NAME;
    OCIDate      B_DATE;
};
typedef struct PERSON PERSON;

struct PERSON_ind
{
    OCIInd    _atomic;
    OCIInd    NAME;
    OCIInd    B_DATE;
};
typedef struct PERSON_ind PERSON_ind;

OCINumber *age (ctx, person_obj, person_obj_ind, tdo, ret_ind)
OCIExtProcContext *ctx;
PERSON         *person_obj;
PERSON_ind     *person_obj_ind;
OCIType        *tdo;
OCIInd         *ret_ind;
{
    sword      err;
    text       errbuf[512];
    OCIEnv    *envh;
    OCISvcCtx *svch;
    OCIError  *errh;
    OCINumber *age;
    int        inum = 0;
    sword      status;

    /* get OCI Environment */
    err = OCIExtProcGetEnv( ctx, &envh, &svch, &errh );

    /* initialize return age to 0 */
    age = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
    status = OCINumberFromInt(errh, &inum, sizeof(inum), OCI_NUMBER_SIGNED,
                              age);
    if (status != OCI_SUCCESS)
    {
```

```
      OCIExtProcRaiseExcp(ctx, (int)1476);
      return (age);
    }

    /* return NULL if the person object is null or the birthdate is null */
    if ( person_obj_ind->_atomic == OCI_IND_NULL ||
         person_obj_ind->B_DATE  == OCI_IND_NULL )
    {
        *ret_ind = OCI_IND_NULL;
        return (age);
    }

    /* The actual implementation to calculate the age is left to the reader,
       but an easy way of doing this is a callback of the form:
            select trunc(months_between(sysdate, person_obj->b_date) / 12)
            from dual;
    */
    *ret_ind = OCI_IND_NOTNULL;
    return (age);
}
```

Passing Parameters by Reference

In C, you can pass `IN` scalar parameters by value (the value of the parameter is passed) or by reference (a pointer to the value is passed). When an external procedure expects a pointer to a scalar, specify `BY REFERENCE` phrase to pass the parameter by reference:

```
CREATE OR REPLACE PROCEDURE findRoot_proc (
   x IN REAL)
AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_findRoot"
   PARAMETERS (
      x BY REFERENCE);
```

In this case, the C prototype would be:

```
void C_findRoot(float *x);
```

This is rather than the default, which would be used when there is no `PARAMETERS` clause:

```
void C_findRoot(float x);
```

WITH CONTEXT

By including the WITH CONTEXT clause, you can give an external procedure access to information about parameters, exceptions, memory allocation, and the user environment. The WITH CONTEXT clause specifies that a context pointer will be passed to the external procedure. For example, if you write the following PL/SQL function:

```
CREATE OR REPLACE FUNCTION getNum_func (
   x IN REAL)
RETURN BINARY_INTEGER AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_getNum"
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      x BY REFERENCE,
      RETURN INDICATOR);
```

Then, the C prototype would be:

```
int C_getNum(
   OCIExtProcContext *with_context,
   float *x,
   short *retind);
```

The context data structure is opaque to the external procedure; but, is available to service procedures called by the external procedure.

If you also include the PARAMETERS clause, then you must specify the parameter CONTEXT, which shows the position of the context pointer in the parameter list. If you omit the PARAMETERS clause, then the context pointer is the first parameter passed to the external procedure.

Inter-Language Parameter Mode Mappings

PL/SQL supports the IN, IN OUT, and OUT parameter modes, as well as the RETURN clause for procedures returning values.

Executing External Procedures with the CALL Statement

Now that you have published your Java class method or external C procedure, you are ready to invoke it.

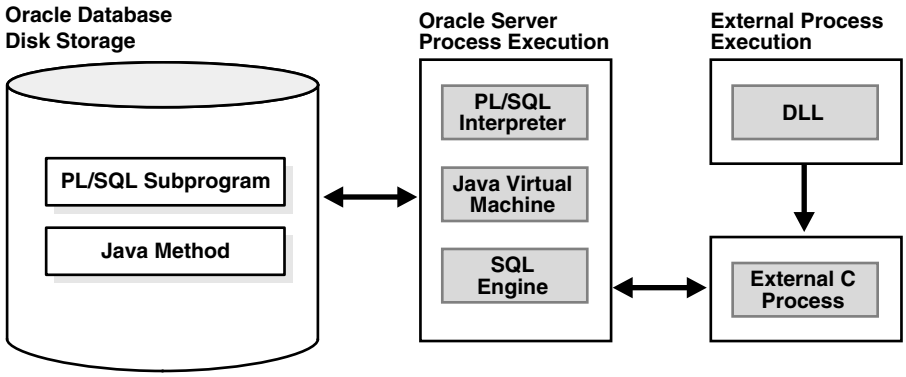
Do not call an external procedure directly. Instead, use the `CALL` statement to call the PL/SQL subprogram that published the external procedure. See "CALL Statement Syntax".

Such calls, which you code in the same manner as a call to a regular PL/SQL procedure or function, can appear in the following:

- Anonymous blocks
- Standalone and packaged subprograms
- Methods of an object type
- Database triggers
- SQL statements (calls to packaged functions only).

Any PL/SQL block or subprogram executing on the server side, or on the client side, (for example, in a tool such as Oracle Forms) can call an external procedure. On the server side, the external procedure runs in a separate process address space, which safeguards your database. Figure 8–2 shows how Oracle Database and external procedures interact.

Figure 8–2 Oracle Database and External Procedures



Preconditions for External Procedures

Before calling external procedures, you should consider the privileges, permissions, and synonyms that exist in the execution environment.

Privileges of External Procedures

When external procedures are called through CALL specifications, they execute with *definer's privileges*, rather than invoker's privileges.

A program executing with invoker's privileges is not bound to a particular schema. It executes at the calling site and accesses database items (such as tables and views) with the caller's visibility and permissions. However, a program executing with definer's privileges is bound to the schema in which it is defined. It executes at the defining site, in the definer's schema, and accesses database items with the definer's visibility and permissions.

Managing Permissions

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager
GRANT CREATE ANY DIRECTORY to scott;
CONNECT scott/tiger
CREATE OR REPLACE DIRECTORY bfile_dir AS '/tmp';
CREATE OR REPLACE JAVA RESOURCE NAMED "appImages" USING BFILE
(bfile_dir,'bfile_audio');
```

To call external procedures, a user must have the EXECUTE privilege on the call specification and on any resources used by the procedure.

In SQL*Plus, you can use the GRANT and REVOKE data control statements to manage permissions. For example:

```
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Public;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Public;
GRANT EXECUTE ON JAVA RESOURCE "appImages" TO Public;
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Scott;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Scott;
```

See Also: *Oracle Database SQL Reference*

Creating Synonyms for External Procedures

For convenience, you or your DBA can create synonyms for external procedures using the CREATE [PUBLIC] SYNONYM statement. In the following example, your

DBA creates a public synonym, which is accessible to all users. If PUBLIC is not specified, then the synonym is private and accessible only within its schema.

```
CREATE PUBLIC SYNONYM Rfac FOR Scott.RecursiveFactorial;
```

CALL Statement Syntax

Invoke the external procedure by means of the SQL CALL statement. You can execute the CALL statement interactively from SQL*Plus. The syntax is:

```
CALL [schema.][{object_type_name | package_name}]procedure_name[@dblink_name]
   [(parameter_list)] [INTO :host_variable][INDICATOR][:indicator_variable];
```

This is essentially the same as executing a procedure foo() using a SQL statement of the form "SELECT foo(...) FROM dual," except that the overhead associated with performing the SELECT is not incurred.

For example, here is an anonymous PL/SQL block that uses dynamic SQL to call plsToC_demoExternal_proc, which we published. PL/SQL passes three parameters to the external C procedure C_demoExternal_proc.

```
DECLARE
   xx NUMBER(4);
   yy VARCHAR2(10);
   zz DATE;
 BEGIN
    EXECUTE IMMEDIATE 'CALL plsToC_demoExternal_proc(:xxx, :yyy, :zzz)' USING
xx,yy,zz;
 END;
```

The semantics of the CALL statement is identical to the that of an equivalent BEGIN..END block.

Note: CALL is the only SQL statement that cannot be put, by itself, in a PL/SQL BEGIN...END block. It can be part of an EXECUTE IMMEDIATE statement within a BEGIN...END block.

Calling Java Class Methods

Here is how you would call the J_calcFactorial class method published earlier. First, declare and initialize two SQL*Plus host variables, as follows:

```
VARIABLE x NUMBER
VARIABLE y NUMBER
```

```
EXECUTE :x := 5;
```

Call `J_calcFactorial`:

```
CALL J_calcFactorial(:x) INTO :y;
PRINT y
```

The result:

```
Y
------
   120
```

How the Database Server Calls External C Procedures

To call an external C procedure, PL/SQL must find the path of the appropriate DLL. The PL/SQL engine retrieves the path from the data dictionary, based on the library alias from the `AS LANGUAGE` clause of the procedure declaration.

Next, PL/SQL alerts a Listener process which, in turn, spawns a session-specific agent. By default, this agent is named `extproc`, although you can specify other names in the `listener.ora` file. The Listener hands over the connection to the agent, and PL/SQL passes to the agent the name of the DLL, the name of the external procedure, and any parameters.

Then, the agent loads the DLL and runs the external procedure. Also, the agent handles service calls (such as raising an exception) and callbacks to Oracle Database. Finally, the agent passes to PL/SQL any values returned by the external procedure.

> **Note:** Although some DLL caching takes place, there is no guarantee that your DLL will remain in the cache; therefore, do not store global variables in your DLL.

After the external procedure completes, the agent remains active throughout your Oracle Database session; when you log off, the agent is killed. Consequently, you incur the cost of launching the agent only once, no matter how many calls you make. Still, you should call an external procedure only when the computational benefits outweigh the cost.

Here, we call PL/SQL function `plsCallsCdivisor_func`, which we published previously, from an anonymous block. PL/SQL passes the two integer parameters to external function `Cdivisor_func`, which returns their greatest common divisor.

```
DECLARE
   g    BINARY_INTEGER;
   a    BINARY_INTEGER;
   b    BINARY_INTEGER;
CALL plsCallsCdivisor_func(a, b);
IF g IN (2,4,8) THEN ...
```

Handling Errors and Exceptions in Multi-Language Programs

Generic Compile Time Call specification Errors

The PL/SQL compiler raises compile time errors if the following conditions are detected in the syntax:

- An `AS EXTERNAL` call specification is found in a `TYPE` or `PACKAGE` specification.

C Exception Handling

C programs can raise exceptions through the `OCIExtproc...` functions.

Using Service Procedures with External C Procedures

When called from an external procedure, a **service routine** can raise exceptions, allocate memory, and invoke OCI handles for callbacks to the server. To use a service routine, you must specify the `WITH CONTEXT` clause, which lets you pass a context structure to the external procedure. The context structure is declared in header file `ociextp.h` as follows:

```
typedef struct OCIExtProcContext OCIExtProcContext;
```

Note: `ociextp.h` is located in `$ORACLE_HOME/plsql/public` on UNIX.

OCIExtProcAllocCallMemory

This service routine allocates *n* bytes of memory for the duration of the external procedure call. Any memory allocated by the function is freed automatically as soon as control returns to PL/SQL.

> **Note:** The external procedure does not need to (and should not) call the C function `free()` to free memory allocated by this service routine as this is handled automatically.

The C prototype for this function is as follows:

```
dvoid *OCIExtProcAllocCallMemory(
   OCIExtProcContext *with_context,
   size_t amount);
```

The parameters `with_context` and `amount` are the context pointer and number of bytes to allocate, respectively. The function returns an untyped pointer to the allocated memory. A return value of zero indicates failure.

In SQL*Plus, suppose you publish external function `plsToC_concat_func`, as follows:

> **Note:** You may need to set up data structures similar to the following for certain examples to work:
>
> ```
> CONNECT system/manager
> DROP USER y CASCADE;
> GRANT CONNECT,RESOURCE,CREATE LIBRARY TO y IDENTIFIED BY y;
> CONNECT y/y
> CREATE LIBRARY stringlib AS
> '/private/varora/ilmswork/Cexamples/john2.so';
> ```

```
CREATE OR REPLACE FUNCTION plsToC_concat_func (
   str1 IN VARCHAR2,
   str2 IN VARCHAR2)
RETURN VARCHAR2 AS LANGUAGE C
NAME "concat"
LIBRARY stringlib
WITH CONTEXT
PARAMETERS (
CONTEXT,
str1   STRING,
str1   INDICATOR short,
str2   STRING,
str2   INDICATOR short,
RETURN INDICATOR short,
RETURN LENGTH short,
```

```
RETURN STRING);
```

When called, C_concat concatenates two strings, then returns the result:

```
select plsToC_concat_func('hello ', 'world') from dual;
PLSTOC_CONCAT_FUNC('HELLO','WORLD')
--------------------------------------------------------------------------------
hello world
```

If either string is NULL, the result is also NULL. As the following example shows, C_concat uses OCIExtProcAllocCallMemory to allocate memory for the result string:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>
#include <ociextp.h>

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char   *str1;
short  str1_i;
char   *str2;
short  str2_i;
short  *ret_i;
short  *ret_l;
{
  char *tmp;
  short len;
  /* Check for null inputs. */
  if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
  {
      *ret_i = (short)OCI_IND_NULL;
      /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
      tmp = OCIExtProcAllocCallMemory(ctx, 1);
      tmp[0] = '\0';
      return(tmp);
  }
  /* Allocate memory for result string, including NULL terminator. */
  len = strlen(str1) + strlen(str2);
  tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

  strcpy(tmp, str1);
  strcat(tmp, str2);
```

```
  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}

#ifdef LATER
static void checkerr (/*_ OCIError *errhp, sword status _*/);

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                       errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
```

```
    break;
  }
}

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char   *str1;
short  str1_i;
char   *str2;
short  str2_i;
short  *ret_i;
short  *ret_l;
{
  char *tmp;
  short len;
  /* Check for null inputs. */
  if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
  {
      *ret_i = (short)OCI_IND_NULL;
      /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
      tmp = OCIExtProcAllocCallMemory(ctx, 1);
      tmp[0] = '\0';
      return(tmp);
  }
  /* Allocate memory for result string, including NULL terminator. */
  len = strlen(str1) + strlen(str2);
  tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

  strcpy(tmp, str1);
  strcat(tmp, str2);

  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}

/*=========================================================================*/
int main(char *argv, int argc)
{
  OCIExtProcContext *ctx;
  char           *str1;
  short          str1_i;
  char           *str2;
```

```
short          str2_i;
short          *ret_i;
short          *ret_l;
/* OCI Handles */
OCIEnv        *envhp;
OCIServer     *srvhp;
OCISvcCtx     *svchp;
OCIError      *errhp;
OCISession    *authp;
OCIStmt       *stmthp;
OCILobLocator *clob, *blob;
OCILobLocator *Lob_loc;

/* Initialize and Logon */
(void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                     (dvoid * (*)(dvoid *, size_t)) 0,
                     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                     (void (*)(dvoid *, dvoid *)) 0 );

(void) OCIEnvInit( (OCIEnv **) &envhp,
                   OCI_DEFAULT, (size_t) 0,
                   (dvoid **) 0 );

(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                  (size_t) 0, (dvoid **) 0);

/* Server contexts */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                 (size_t) 0, (dvoid **) 0);

/* Service context */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                 (size_t) 0, (dvoid **) 0);

/* Attach to Oracle Database */
(void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);

/* Set attribute server context in the service context */
(void) OCIAttrSet ((dvoid *) svchp, OCI_HTYPE_SVCCTX,
                   (dvoid *)srvhp, (ub4) 0,
                  OCI_ATTR_SERVER, (OCIError *) errhp);

(void) OCIHandleAlloc((dvoid *) envhp,
                      (dvoid **)&authp, (ub4) OCI_HTYPE_SESSION,
                      (size_t) 0, (dvoid **) 0);
```

```
  (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) "samp", (ub4)4,
                (ub4) OCI_ATTR_USERNAME, errhp);

  (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) "samp", (ub4) 4,
                (ub4) OCI_ATTR_PASSWORD, errhp);

  /* Begin a User Session */
  checkerr(errhp, OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,
                        (ub4) OCI_DEFAULT));

  (void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                  (dvoid *) authp, (ub4) 0,
                  (ub4) OCI_ATTR_SESSION, errhp);

  /* ----------------------User Logged In-----------------------------*/
  printf ("user logged in \n");

  /* allocate a statement handle */
  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
           OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  checkerr(errhp, OCIDescriptorAlloc((dvoid *)envhp, (dvoid **) &Lob_loc,
                                  (ub4) OCI_DTYPE_LOB,
                                  (size_t) 0, (dvoid **) 0));

  /* ------- subroutine called  here-----------------------*/
  printf ("calling concat...\n");
  concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l);

  return 0;
}

#endif
```

OCIExtProcRaiseExcp

This service routine raises a predefined exception, which must have a valid Oracle Database error number in the range 1..32767. After doing any necessary cleanup, your external procedure must return immediately. (No values are assigned to `OUT` or `IN OUT` parameters.) The C prototype for this function follows:

```
int OCIExtProcRaiseExcp(
```

```
   OCIExtProcContext *with_context,
   size_t errnum);
```

The parameters `with_context` and `error_number` are the context pointer and Oracle Database error number. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

In SQL*Plus, suppose you publish external procedure `plsTo_divide_proc`, as follows:

```
CREATE OR REPLACE PROCEDURE plsTo_divide_proc (
   dividend IN BINARY_INTEGER,
   divisor  IN BINARY_INTEGER,
   result   OUT FLOAT)
AS LANGUAGE C
   NAME "C_divide"
   LIBRARY MathLib
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      dividend INT,
      divisor  INT,
      result   FLOAT);
```

When called, `C_divide` finds the quotient of two numbers. As the following example shows, if the divisor is zero, `C_divide` uses `OCIExtProcRaiseExcp` to raise the predefined exception `ZERO_DIVIDE`:

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int    dividend;
int    divisor;
float  *result;
{
  /* Check for zero divisor. */
  if (divisor == (int)0)
  {
    /* Raise exception ZERO_DIVIDE, which is Oracle error 1476. */
    if (OCIExtProcRaiseExcp(ctx, (int)1476) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /* Incorrect parameters were passed. */
```

```
      assert(0);
    }
  }
  *result = (float)dividend / (float)divisor;
}
```

OCIExtProcRaiseExcpWithMsg

This service routine raises a user-defined exception and returns a user-defined error message. The C prototype for this function follows:

```
int OCIExtProcRaiseExcpWithMsg(
   OCIExtProcContext *with_context,
   size_t  error_number,
   text   *error_message,
   size_t  len);
```

The parameters `with_context`, `error_number`, and `error_message` are the context pointer, Oracle Database error number, and error message text. The parameter `len` stores the length of the error message. If the message is a null-terminated string, then `len` is zero. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

In the previous example, we published external procedure `plsTo_divide_proc`. In the following example, you use a different implementation. With this version, if the divisor is zero, then `C_divide` uses `OCIExtProcRaiseExcpWithMsg` to raise a user-defined exception:

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int    dividend;
int    divisor;
float  *result;
  /* Check for zero divisor. */
  if (divisor == (int)0)
  {
    /* Raise a user-defined exception, which is Oracle error 20100,
       and return a null-terminated error message. */
    if (OCIExtProcRaiseExcpWithMsg(ctx, (int)20100,
          "divisor is zero", 0) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /*  Incorrect parameters were passed. */
```

```
      assert(0);
    }
  }
  *result = dividend / divisor;

}
```

Doing Callbacks with External C Procedures

OCIExtProcGetEnv

This service routine enables OCI callbacks to the database during an external procedure call. The environment handles obtained by using this function reuse the existing connection to go back to the database. If you need to establish a new connection to the database, you cannot use these handles; instead, you must create your own.

The C prototype for this function follows:

```
sword OCIExtProcGetEnv ( OCIExtProcContext *with_context,
   OCIEnv envh,
   OCISvcCtx svch,
   OCIError errh )
```

The parameter `with_context` is the context pointer, and the parameters `envh`, `svch`, and `errh` are the OCI environment, service, and error handles, respectively. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

Both external C procedures and Java class methods can call-back to the database to do SQL operations. For a working example, see "Demo Program" on page 8-51.

> **Note:** Callbacks are not necessarily a same-session phenomenon; you may execute an SQL statement in a different session through `OCIlogon`.

An external C procedure executing on Oracle Database can call a service routine to obtain OCI environment and service handles. With the OCI, you can use callbacks to execute SQL statements and PL/SQL subprograms, fetch data, and manipulate LOBs. Callbacks and external procedures operate in the same user session and transaction context, and so have the same user privileges.

In SQL*Plus, suppose you run the following script:

```
CREATE TABLE Emp_tab (empno NUMBER(10))

CREATE PROCEDURE plsToC_insertIntoEmpTab_proc (
   empno BINARY_INTEGER)
AS LANGUAGE C
   NAME "C_insertEmpTab"
   LIBRARY insert_lib
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      empno LONG);
```

Later, you might call service routine `OCIExtProcGetEnv` from external procedure `plsToC_insertIntoEmpTab_proc`, as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <oratypes.h>
#include <oci.h>   /* includes ociextp.h */
...
void C_insertIntoEmpTab (ctx, empno)
OCIExtProcContext *ctx;
long empno;
{
  OCIEnv    *envhp;
  OCISvcCtx *svchp;
  OCIError  *errhp;
  int        err;
  ...
  err = OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp);
  ...
}
```

If you do not use callbacks, you do not need to include `oci.h`; instead, just include `ociextp.h`.

Object Support for OCI Callbacks

To execute object-related callbacks from your external procedures, the OCI environment in the `extproc` agent is fully initialized in object mode. You retrieve handles to this environment with the `OCIExtProcGetEnv`() procedure.

The object runtime environment lets you use static, as well as dynamic, object support provided by OCI. To utilize static support, use the OTT to generate C

structs for the appropriate object types, and then use conventional C code to access the object attributes.

For those objects whose types are unknown at external procedure creation time, an alternative, dynamic, way of accessing objects is first to invoke `OCIDescribeAny`() to obtain attribute and method information about the type. Then, `OCIObjectGetAttr`() and `OCIObjectSetAttr`() can be called to retrieve and set attribute values.

Because the current external procedure model is stateless, `OCIExtProcGetEnv`() must be called in every external procedure that wants to execute callbacks, or invoke `OCIExtProc`...() service routines. After every external procedure invocation, the callback mechanism is cleaned up and all OCI handles are freed.

Restrictions on Callbacks

With callbacks, the following SQL commands and OCI procedures are not supported:

- Transaction control commands such as `COMMIT`
- Data definition commands such as `CREATE`
- The following object-oriented OCI procedures:

```
OCIObjectNew
OCIObjectPin
OCIObjectUnpin
OCIObjectPinCountReset
OCIObjectLock
OCIObjectMarkUpdate
OCIObjectUnmark
OCIObjectUnmarkByRef
OCIObjectAlwaysLatest
OCIObjectNotAlwaysLatest
OCIObjectMarkDeleteByRef
OCIObjectMarkDelete
OCIObjectFlush
OCIObjectFlushRefresh
OCIObjectGetTypeRef
OCIObjectGetObjectRef
OCIObjectExists
OCIObjectIsLocked
OCIObjectIsDirtied
OCIObjectIsLoaded
OCIObjectRefresh
```

```
OCIObjectPinTable
OCIObjectArrayPin
OCICacheFlush,
OCICacheFlushRefresh,
OCICacheRefresh
OCICacheUnpin
OCICacheFree
OCICacheUnmark
OCICacheGetObjects
OCICacheRegister
```

- Polling-mode OCI procedures such as `OCIGetPieceInfo`
- The following OCI procedures:

```
OCIEnvInit
OCIInitialize
OCIPasswordChange
OCIServerAttach
OCIServerDetach
OCISessionBegin
OCISessionEnd
OCISvcCtxToLda
OCITransCommit
OCITransDetach
OCITransRollback
OCITransStart
```

Also, with OCI procedure `OCIHandleAlloc`, the following handle types are not supported:

```
OCI_HTYPE_SERVER
OCI_HTYPE_SESSION
OCI_HTYPE_SVCCTX
OCI_HTYPE_TRANS
```

Debugging External Procedures

Usually, when an external procedure fails, its prototype is faulty. In other words, the prototype does not match the one generated internally by PL/SQL. This can happen if you specify an incompatible C datatype. For example, to pass an OUT parameter of type REAL, you must specify `float *`. Specifying `float`, `double *`, or any other C datatype will result in a mismatch.

In such cases, you might get:

```
lost RPC connection to external routine agent
```

This error, which means that agent `extproc` terminated abnormally because the external procedure caused a core dump. To avoid errors when declaring C prototype parameters, refer to the preceding tables.

Using Package DEBUG_EXTPROC

To help you debug external procedures, PL/SQL provides the utility package `DEBUG_EXTPROC`. To install the package, run the script `dbgextp.sql` which you can find in the PL/SQL demo directory. (For the location of the directory, see your Oracle Database Installation or User's Guide.)

To use the package, follow the instructions in `dbgextp.sql`. Your Oracle Database account must have `EXECUTE` privileges on the package and `CREATE LIBRARY` privileges.

Note: `DEBUG_EXTPROC` works only on platforms with debuggers that can attach to a running process.

Demo Program

Also in the PL/SQL demo directory is the script `extproc.sql`, which demonstrates the calling of an external procedure. The companion file `extproc.c` contains the C source code for the external procedure.

To run the demo, follow the instructions in `extproc.sql`. You must use the `SCOTT/TIGER` account, which must have `CREATE LIBRARY` privileges.

Guidelines for External C Procedures

Handling Global Variables

A global variable is declared outside of a function, and its value is shared by all functions of a program. In case of external procedures, this means that all functions in a DLL share the value of the global. The usage of global variables is discouraged for two reasons:

- *Threading:* In the non-threaded configuration of the agent process, there is only one function active at a time. However, in the case of multithreaded agents, multiple functions can be active at the same time. In that case, it is possible that

two or more functions would try to access the global variable concurrently, with unsuccessful results.

- *DLL caching*: Global variables are also used to store data that is intended to persist beyond the lifetime of a function. For example, consider two functions *func1*() and *func2*() trying to pass data to each other. Because of the DLL caching feature, it is possible that after *func1*()'s completion, the DLL will be unloaded, which results in all global variables losing their values. When *func2*() is executed, the DLL is reloaded, and all globals are initialized to 0, which will be inconsistent with their values at the completion of *func1*().

Handling Static Variables

There are two types of static variables: external and internal. An external static variable is a special case of a global variable, so its usage is discouraged. Internal static variables are local to a particular function, but remain in existence rather than coming and going each time the function is activated. Therefore, they provide private, permanent storage within a single function. These variables are used to pass on data to subsequent invocations of the same function. But, because of the DLL caching feature mentioned previously, the DLL might be unloaded and reloaded between invocations, which means that the internal static variable would lose its value.

> **See Also:** Template `makefile` in the RDBMS subdirectory `/public` for help creating a dynamic link library

Guidelines for Call Specifications and CALL Statements

When calling external procedures:

- Never write to `IN` parameters or overflow the capacity of `OUT` parameters. (PL/SQL does no run time checks for these error conditions.)
- Never read an `OUT` parameter or a function result.
- Always assign a value to `IN OUT` and `OUT` parameters and to function results. Otherwise, your external procedure will not return successfully.
- If you include the `WITH CONTEXT` and `PARAMETERS` clauses, then you must specify the parameter `CONTEXT`, which shows the position of the context pointer in the parameter list.
- If you include the `PARAMETERS` clause, and if the external procedure is a function, then you must specify the parameter `RETURN` in the last position.

- For every formal parameter, there must be a corresponding parameter in the PARAMETERS clause. Also, make sure that the datatypes of parameters in the PARAMETERS clause are compatible with those in the C prototype, because no implicit conversions are done.
- With a parameter of type RAW or LONG RAW, you must use the property LENGTH. Also, if that parameter is IN OUT or OUT and null, then you must set the length of the corresponding C parameter to zero.

Restrictions on External C Procedures

The following restrictions apply to external procedures:

- This feature is available only on platforms that support DLLs.
- Only C procedures and procedures callable from C code are supported.
- You cannot pass PL/SQL cursor variables or records to an external procedure. For records, use instances of object types instead.
- In the LIBRARY subclause, you cannot use a database link to specify a remote library.
- The maximum number of parameters that you can pass to a external procedure is 128. However, if you pass float or double parameters by value, then the maximum is less than 128. How much less depends on the number of such parameters and your operating system. To get a rough estimate, count each float or double passed by value as two parameters.

Part III

The Active Database

To take advantage of the reliability and performance of the database server, and to reuse program logic across all the applications in a database, you can move some of your program logic into the database itself, so that the database does cleanup operations and responds to events without the need for a separate application.

This part contains the following chapters:

9

Using Triggers

Triggers are procedures that are stored in the database and implicitly run, or **fired**, when something happens.

Traditionally, triggers supported the execution of a PL/SQL block when an `INSERT`, `UPDATE`, or `DELETE` occurred on a table or view. Starting with Oracle8*i*, triggers support system and other data events on `DATABASE` and `SCHEMA`. Oracle Database also supports the execution of a PL/SQL or Java procedure.

This chapter discusses DML triggers, `INSTEAD OF` triggers, and system triggers (triggers on `DATABASE` and `SCHEMA`). Topics include:

Designing Triggers

Use the following guidelines when designing your triggers:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Do not define triggers that duplicate features already built into Oracle Database. For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints.
- Limit the size of triggers. 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.
- Use 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 create recursive triggers.* For example, creating an `AFTER UPDATE` statement trigger on the `Emp_tab` table that itself issues an `UPDATE` statement on `Emp_tab`, causes the trigger to fire recursively until it has run out of memory.
- Use triggers on `DATABASE` judiciously. They are executed for *every* user *every* time the event occurs on which the trigger is created.

Creating Triggers

Triggers are created using the `CREATE TRIGGER` statement. This statement can be used with any interactive tool, such as SQL*Plus or Enterprise Manager. When using an interactive tool, a single slash (/) on the last line is necessary to activate the `CREATE TRIGGER` statement.

The following statement creates a trigger for the `Emp_tab` table.

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab
  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;
/
```

The trigger is fired when DML operations (`INSERT`, `UPDATE`, and `DELETE` statements) are performed on the table. You can choose what combination of operations should fire the trigger.

Because the trigger uses the `BEFORE` keyword, it can access the new values before they go into the table, and can change the values if there is an easily-corrected error by assigning to `:NEW.`*`column_name`*. You might use the `AFTER` keyword if you want the trigger to query or change the same table, because triggers can only do that after the initial changes are applied and the table is back in a consistent state.

Because the trigger uses the `FOR EACH ROW` clause, it might be executed multiple times, such as when updating or deleting multiple rows. You might omit this clause if you just want to record the fact that the operation occurred, but not examine the data for each row.

Once the trigger is created, entering the following SQL statement:

```
UPDATE Emp_tab SET sal = sal + 500.00 WHERE deptno = 10;
```

fires the trigger once for each row that is updated, in each case printing the new salary, old salary, and the difference.

The `CREATE` (or `CREATE OR REPLACE`) statement fails if any errors exist in the PL/SQL block.

> **Note:** The size of the trigger cannot be more than 32K.

The following sections use this example to illustrate the way that parts of a trigger are specified.

> **See Also:** "Examples of Trigger Applications" on page 9-31 for more realistic examples of `CREATE TRIGGER` statements

Types of Triggers

A trigger is either a stored PL/SQL block or a PL/SQL, C, or Java procedure associated with a table, view, schema, or the database itself. Oracle Database automatically executes a trigger when a specified event takes place, which may be in the form of a system event or a DML statement being issued against the table.

Triggers can be:

- DML triggers on tables.
- `INSTEAD OF` triggers on views.
- System triggers on `DATABASE` or `SCHEMA`: With `DATABASE`, triggers fire for each event for all users; with `SCHEMA`, triggers fire for each event for that specific user.

> **See Also:** *Oracle Database SQL Reference* for information on trigger creation syntax

Overview of System Events

You can create triggers to be fired on any of the following:

- DML statements (`DELETE`, `INSERT`, `UPDATE`)
- DDL statements (`CREATE`, `ALTER`, `DROP`)
- Database operations (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, `SHUTDOWN`)

Getting the Attributes of System Events

You can get certain event-specific attributes when the trigger is fired.

> **See Also:** Chapter 10, "Working With System Events" for a complete list of the functions you can call to get the event attributes

Creating a trigger on `DATABASE` implies that the triggering event is outside the scope of a user (for example, database `STARTUP` and `SHUTDOWN`), and it applies to all users (for example, a trigger created on `LOGON` event by the DBA).

Creating a trigger on `SCHEMA` implies that the trigger is created in the current user's schema and is fired only for that user.

For each trigger, publication can be specified on DML and system events.

> **See Also:** "Responding to System Events through Triggers" on page 9-50

Naming Triggers

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need 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 (however, to avoid confusion, this is not recommended).

When Is the Trigger Fired?

A trigger is fired based on a **triggering statement**, which specifies:

- The SQL statement or the system event, database event, or DDL event that fires the trigger body. The options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
- The table, view, DATABASE, or SCHEMA associated with the trigger.

> **Note:** Exactly one table or view can be specified in the triggering statement. If the INSTEAD OF option is used, then the triggering statement may only specify a view; conversely, if a view is specified in the triggering statement, then only the INSTEAD OF option may be used.

For example, the PRINT_SALARY_CHANGES trigger fires after any DELETE, INSERT, or UPDATE on the Emp_tab table. Any of the following statements trigger the PRINT_SALARY_CHANGES trigger given in the previous example:

```
DELETE FROM Emp_tab;
INSERT INTO Emp_tab VALUES ( ... );
INSERT INTO Emp_tab SELECT ... FROM ... ;
UPDATE Emp_tab SET ... ;
```

Do Import and SQL*Loader Fire Triggers?

INSERT triggers fire during SQL*Loader conventional loads. (For direct loads, triggers are disabled before the load.)

The IGNORE parameter of the IMP command determines whether triggers fire during import operations:

- If IGNORE=N (default) and the table already exists, then import does not change the table and no existing triggers fire.
- If the table does not exist, then import creates and loads it before any triggers are defined, so again no triggers fire.

- If IGNORE=Y, then import loads rows into existing tables. Any existing triggers fire, and indexes are updated to account for the imported data.

How Column Lists Affect UPDATE Triggers

An UPDATE statement might include a list of columns. If a triggering statement includes a column list, the trigger is fired only when one of the specified columns is updated. If a triggering statement omits 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 could include a column list in the triggering statement. For example:

```
... BEFORE DELETE OR INSERT OR UPDATE OF ename ON Emp_tab ...
```

Notes:

- You cannot specify a column list for UPDATE with INSTEAD OF triggers.
- If the column specified in the UPDATE OF clause is an object column, then the trigger is also fired if any of the attributes of the object are modified.
- You cannot specify UPDATE OF clauses on collection columns.

Controlling When a Trigger Is Fired (BEFORE and AFTER Options)

The BEFORE or AFTER option in the CREATE TRIGGER statement specifies exactly when to fire the trigger body in relation to the triggering statement that is being run. 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.

In general, you use BEFORE or AFTER triggers to achieve the following results:

- Use BEFORE row triggers to modify the row before the row data is written to disk.
- Use AFTER row triggers to obtain, and perform operations, using the row ID.

Note: BEFORE row triggers are slightly more efficient than AFTER row triggers. With AFTER 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 BEFORE row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

BEFORE Triggers Fired Multiple Times

If an UPDATE or DELETE statement detects a conflict with a concurrent UPDATE, then Oracle Database 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. Your package should include a counter variable to detect this situation.

Ordering of Triggers

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 are 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, then 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 Database 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 run 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 Database 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, then Oracle Database chooses an arbitrary order to execute these triggers.

> **See Also:** *Oracle Database Concepts* 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).

Modifying Complex Views (INSTEAD OF Triggers)

An **updatable** view is one that lets you perform DML on the underlying table. Some views are inherently updatable, but others are not because they were created with one or more of the constructs listed in"Views that Require INSTEAD OF Triggers".

Any view that contains one of those constructs can be made updatable by using an `INSTEAD OF` trigger. `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 Database fires the trigger *instead* of executing the triggering statement. The trigger must determine what operation was intended and perform `UPDATE`, `INSERT`, or `DELETE` operations directly on the underlying tables.

With an `INSTEAD OF` trigger, you can 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.

`INSTEAD OF` triggers can only be activated for each row.

> **See Also:** "Firing Triggers One or Many Times (FOR EACH ROW Option)" on page 9-13

Note:

- The INSTEAD OF option can *only* be used for triggers created over views.
- The BEFORE and AFTER options *cannot* be used for triggers created over views.
- The CHECK option for views is not enforced when inserts or updates to the view are done using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

Views that Require INSTEAD OF Triggers

A view cannot be modified by UPDATE, INSERT, or DELETE statements if the view query contains any of the following constructs:

- A set operator
- A DISTINCT operator
- An aggregate or analytic function
- A GROUP BY, ORDER BY, MODEL, CONNECT BY, or START WITH clause
- A collection expression in a SELECT list
- A subquery in a SELECT list
- A subquery designated WITH READ ONLY
- Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

If a view contains pseudocolumns or expressions, then you can only update the view with an UPDATE statement that does not refer to any of the pseudocolumns or expressions.

INSTEAD OF Trigger Example

Note: You may need to set up the following data structures for this example to work:

```
CREATE TABLE Project_tab (
   Prj_level NUMBER,
   Projno    NUMBER,
   Resp_dept NUMBER);
CREATE TABLE Emp_tab (
   Empno     NUMBER NOT NULL,
   Ename     VARCHAR2(10),
   Job       VARCHAR2(9),
   Mgr       NUMBER(4),
   Hiredate  DATE,
   Sal       NUMBER(7,2),
   Comm      NUMBER(7,2),
   Deptno    NUMBER(2) NOT NULL);

CREATE TABLE Dept_tab (
   Deptno    NUMBER(2) NOT NULL,
   Dname     VARCHAR2(14),
   Loc       VARCHAR2(13),
   Mgr_no    NUMBER,
   Dept_type NUMBER);
```

The following example shows an `INSTEAD OF` trigger for inserting rows into the `MANAGER_INFO` view.

```
CREATE OR REPLACE VIEW manager_info AS
    SELECT e.ename, e.empno, d.dept_type, d.deptno, p.prj_level,
           p.projno
        FROM   Emp_tab e, Dept_tab d, Project_tab p
        WHERE  e.empno =  d.mgr_no
        AND    d.deptno = p.resp_dept;

CREATE OR REPLACE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info
REFERENCING NEW AS n                -- new manager information

FOR EACH ROW
DECLARE
   rowcnt number;
BEGIN
```

```
   SELECT COUNT(*) INTO rowcnt FROM Emp_tab WHERE empno = :n.empno;
   IF rowcnt = 0  THEN
       INSERT INTO Emp_tab (empno,ename) VALUES (:n.empno, :n.ename);
   ELSE
      UPDATE Emp_tab SET Emp_tab.ename = :n.ename
         WHERE Emp_tab.empno = :n.empno;
   END IF;
   SELECT COUNT(*) INTO rowcnt FROM Dept_tab WHERE deptno = :n.deptno;
   IF rowcnt = 0 THEN
      INSERT INTO Dept_tab (deptno, dept_type)
         VALUES(:n.deptno, :n.dept_type);
   ELSE
      UPDATE Dept_tab SET Dept_tab.dept_type = :n.dept_type
         WHERE Dept_tab.deptno = :n.deptno;
   END IF;
   SELECT COUNT(*) INTO rowcnt FROM Project_tab
      WHERE Project_tab.projno = :n.projno;
   IF rowcnt = 0 THEN
      INSERT INTO Project_tab (projno, prj_level)
         VALUES(:n.projno, :n.prj_level);
   ELSE
      UPDATE Project_tab SET Project_tab.prj_level = :n.prj_level
         WHERE Project_tab.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 Also:** *Oracle Call Interface Programmer's Guide*

To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, you must specify INSTEAD OF triggers, unless the object view is modifiable. If the object is read only, then it is not necessary to define triggers to pin it.

Triggers on Nested Table View Columns

INSTEAD OF triggers can also be created over nested table view columns. These triggers provide a way of updating elements of the nested table. They fire for each nested table element being modified. The row correlation variables inside the trigger correspond to the nested table element. This type of trigger also provides an additional correlation name for accessing the parent row that contains the nested table being modified.

Note: These triggers:

- Can only be defined over nested table columns in views.
- Fire only when the nested table elements are modified using the THE() or TABLE() clauses. They do not fire when a DML statement is performed on the view.

For example, consider a department view that contains a nested table of employees.

```
CREATE OR REPLACE VIEW Dept_view AS
SELECT d.Deptno, d.Dept_type, d.Dept_name,
   CAST (MULTISET ( SELECT e.Empno, e.Empname, e.Salary)
      FROM Emp_tab e
      WHERE e.Deptno = d.Deptno) AS Amp_list_ Emplist
FROM Dept_tab d;
```

The CAST (MULTISET..) operator creates a multi-set of employees for each department. If you want to modify the emplist column, which is the nested table of employees, then you can define an INSTEAD OF trigger over the column to handle the operation.

The following example shows how an insert trigger might be written:

```
CREATE OR REPLACE TRIGGER Dept_emplist_tr
   INSTEAD OF INSERT ON NESTED TABLE Emplist OF Dept_view
   REFERENCING NEW AS Employee
      PARENT AS Department
   FOR EACH ROW
BEGIN
-- The insert on the nested table is translated to an insert on the base table:
   INSERT INTO Emp_tab VALUES (
      :Employee.Empno, :Employee.Empname,:Employee.Salary, :Department.Deptno);
END;
```

Any INSERT into the nested table fires the trigger, and the Emp_tab table is filled with the correct values. For example:

```
INSERT INTO TABLE (SELECT d.Emplist FROM Dept_view d WHERE Deptno = 10)
   VALUES (1001, 'John Glenn', 10000);
```

The :department.deptno correlation variable in this example would have a value of 10.

Firing Triggers One or Many Times (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, then 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 indicates 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:

Note: You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Emp_log (
   Emp_id     NUMBER,
   Log_date   DATE,
   New_salary NUMBER,
   Action     VARCHAR2(20));
```

```
CREATE OR REPLACE TRIGGER Log_salary_increase
AFTER UPDATE ON Emp_tab
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;
```

Then, you enter the following SQL statement:

```
UPDATE Emp_tab SET Sal = Sal + 1000.0
    WHERE Deptno = 20;
```

If there are five employees in department 20, then the trigger fires five times when this statement is entered, because five rows are affected.

The following trigger fires only once for each UPDATE of the Emp_tab table:

```
CREATE OR REPLACE TRIGGER Log_emp_update
AFTER UPDATE ON Emp_tab
BEGIN
    INSERT INTO Emp_log (Log_date, Action)
        VALUES (SYSDATE, 'Emp_tab COMMISSIONS CHANGED');
END;
```

See Also: *Oracle Database Concepts* for the order of trigger firing

The statement level triggers are useful for performing validation checks for the entire statement.

Firing Triggers Based on Conditions (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.

Note: A WHEN clause cannot be included in the definition of a statement trigger.

If included, then the expression in the WHEN clause is evaluated for each row that the trigger affects.

If the expression evaluates to TRUE for a row, then the trigger body is fired on behalf of that row. However, if the expression evaluates to FALSE or NOT TRUE for a row (unknown, as with nulls), then 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 (in other words, 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 is not run 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 later. The expression in a WHEN clause must be a SQL expression, and it cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in the WHEN clause.

Note: You cannot specify the WHEN clause for INSTEAD OF triggers.

Coding the Trigger Body

The trigger body is a CALL procedure or a PL/SQL block that can include SQL and PL/SQL statements. The CALL procedure can be either a PL/SQL or a Java procedure that is encapsulated in a PL/SQL wrapper. These statements are run if the triggering statement is entered and if 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.

Note: The INSERTING, DELETING, and UPDATING conditional predicates cannot be used for the CALL procedures; they can only be used in a PL/SQL block.

Example: Monitoring Logons with a Trigger

Note: You may need to set up data structures similar to the following for certain examples to work:

```
CONNECT system/manager
GRANT ADMINISTER DATABASE TRIGGER TO scott;
CONNECT scott/tiger
CREATE TABLE audit_table (
   seq number,
   user_at  VARCHAR2(10),
   time_now DATE,
   term     VARCHAR2(10),
   job      VARCHAR2(10),
   proc     VARCHAR2(10),
   enum     NUMBER);
```

```
CREATE OR REPLACE PROCEDURE foo (c VARCHAR2) AS
   BEGIN
      INSERT INTO Audit_table (user_at) VALUES(c);
```

```
   END;

CREATE OR REPLACE TRIGGER logontrig AFTER LOGON ON DATABASE
-- Just call an existing procedure. The ORA_LOGIN_USER is a function
-- that returns information about the event that fired the trigger.
CALL foo (ora_login_user)
/
```

Example: Calling a Java Procedure from a Trigger

Although triggers are declared using PL/SQL, they can call procedures in other languages, such as Java:

```
CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS language Java
name 'thjvTriggers.beforeDelete (oracle.sql.NUMBER, oracle.sql.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
FOR EACH ROW
CALL Before_delete (:old.Id, :old.Ename)
/
```

The corresponding Java file is `thjvTriggers.java`:

```
import java.sql.*
import java.io.*
import oracle.sql.*
import oracle.oracore.*
public class thjvTriggers
{
public state void
beforeDelete (NUMBER old_id, CHAR old_name)
Throws SQLException, CoreException
   {
   Connection conn = JDBCConnection.defaultConnection();
   Statement stmt = conn.CreateStatement();
   String sql = "insert into logtab values
   ("+ old_id.intValue() +", '"+ old_ename.toString() + ", BEFORE DELETE');
   stmt.executeUpdate (sql);
   stmt.close();
   return;
   }
}
```

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 no longer exists after the row is deleted, the :new values are NULL. However, you cannot modify :new values: ORA-4084 is raised if you try to modify :new values.

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_tab table (with the columns SAL, COMM, and so on), then you can include statements in the trigger body. For example:

```
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, then 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 body, but a colon is not allowed when using the qualifiers in the WHEN clause or the REFERENCING option.

Example: Modifying LOB Columns with a Trigger

You can treat LOB columns the same as other columns, using regular SQL and PL/SQL functions with `CLOB` columns, and calls to the `DBMS_LOB` package with `BLOB` columns:

```
drop table tab1;

create table tab1 (c1 clob);
insert into tab1 values ('<h1>HTML Document Fragment</h1><p>Some text.');

create or replace trigger trg1
  before update on tab1
  for each row
begin
  dbms_output.put_line('Old value of CLOB column: '||:OLD.c1);
  dbms_output.put_line('Proposed new value of CLOB column: '||:NEW.c1);

-- Previously, we couldn't change the new value for a LOB.
-- Now, we can replace it, or construct a new value using SUBSTR, INSTR...
-- operations for a CLOB, or DBMS_LOB calls for a BLOB.
  :NEW.c1 := :NEW.c1 || to_clob('<hr><p>Standard footer paragraph.');

  dbms_output.put_line('Final value of CLOB column: '||:NEW.c1);
end;
/

set serveroutput on;
update tab1 set c1 = '<h1>Different Document Fragment</h1><p>Different text.';

select * from tab1;
```

INSTEAD OF Triggers on Nested Table View Columns

In the case of `INSTEAD OF` triggers on nested table view columns, the `new` and `old` qualifiers correspond to the new and old nested table elements. The parent row corresponding to this nested table element can be accessed using the `parent` qualifier. The parent correlation name is meaningful and valid only inside a nested table trigger.

Avoiding Name Conflicts with Triggers (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. Because 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:

Note: You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE new (
   field1     NUMBER,
   field2     VARCHAR2(20));
```

```
CREATE OR REPLACE 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 it is then used in the trigger body.

Detecting the DML Operation That Fired a Trigger

If more than one type of DML operation can fire a trigger (for example, ON INSERT OR DELETE OR UPDATE OF Emp_tab), the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to check which type of statement fire the trigger.

Within the code of the trigger body, you can execute blocks of code depending on the kind of DML operation fired the trigger:

```
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 the following:

```
CREATE OR REPLACE TRIGGER ...
... UPDATE OF Sal, Comm ON Emp_tab ...
BEGIN

... IF UPDATING ('SAL') THEN ... END IF;

END;
```

The code in the THEN clause runs only if the triggering UPDATE statement updates the SAL column. This way, the trigger can minimize its overhead when the column of interest is not being changed.

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, then 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.

The only exception to this is when the event under consideration is database STARTUP, SHUTDOWN, or LOGIN when the user logging in is SYSTEM. In these scenarios, only the trigger action is rolled back.

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 OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
  INSERT INTO Emp_tab@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, then Oracle Database cannot validate the statement accessing the remote database, and the compilation fails. The previous example exception statement cannot run, because the trigger does not complete compilation.

Because stored procedures are stored in a compiled form, the work-around for the previous example is as follows:

```
CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
   Insert_row_proc;
END;

CREATE OR REPLACE PROCEDURE Insert_row_proc AS
BEGIN
    INSERT INTO Emp_tab@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 triggers requires some restrictions that are not required for standard PL/SQL blocks. The following sections discuss these restrictions.

Maximum Trigger Size

The size of a trigger cannot be more than 32K.

SQL Statements Allowed 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. ROLLBACK, COMMIT, and SAVEPOINT cannot be used.For system triggers, {CREATE/ALTER/DROP} TABLE statements and ALTER...COMPILE are allowed.

> **Note:** A procedure called by a trigger cannot run the previous transaction control statements, because the procedure runs 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. If a timestamp or signature mismatch is found during execution of the trigger, then the remote procedure is not run, and the trigger is invalidated.

Trigger Restrictions on 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), then a LONG or LONG RAW column can be referenced in a SQL statement within a trigger. The maximum length for these datatypes is 32000 bytes.
- Variables cannot be declared using the LONG or LONG RAW datatypes.
- :NEW and :PARENT cannot be used with LONG or LONG RAW columns.

Trigger Restrictions on Mutating Tables

A **mutating** table is a table that is being modified by an UPDATE, DELETE, or INSERT statement, or a table that might be updated by the effects of a DELETE CASCADE constraint.

The session that issued the triggering statement cannot query or modify a mutating table. This restriction prevents a trigger from seeing an inconsistent set of data.

This restriction applies to all triggers that use the FOR EACH ROW clause. Views being modified in INSTEAD OF triggers are not considered mutating.

When a trigger encounters a mutating table, 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_tab
FOR EACH ROW
DECLARE
    n INTEGER;
BEGIN
    SELECT COUNT(*) INTO n FROM Emp_tab;
    DBMS_OUTPUT.PUT_LINE(' There are now ' || n ||
        ' employees.');
END;
```

If the following SQL statement is entered:

```
DELETE FROM Emp_tab WHERE Empno = 7499;
```

An error is returned because the table is mutating when the row is deleted:

```
ORA-04091: table SCOTT.Emp_tab is mutating, trigger/function may not see it
```

If you delete the line "FOR EACH ROW" from the trigger, it becomes a statement trigger which is not subject to this restriction, and the trigger.

If you need to update a mutating table, you could bypass these restrictions by using a temporary table, a PL/SQL table, or a package variable. For example, in place of a single AFTER row trigger that updates the original table, resulting in a mutating table error, you might 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:** *Oracle Database Concepts* for information about the interaction of triggers and integrity constraints

Because declarative referential integrity constraints are not supported between tables on different nodes of a distributed database, the mutating 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 an Oracle Net path back to the database that contains the link.

Restrictions on Mutating Tables Relaxed

The mutating error, discussed earlier in this section, still prevents the trigger from reading or modifying the table that the parent statement is modifying. However, starting in Oracle Database release 8.1, a delete against the parent table causes before/after statement triggers to be fired once. That way, you can create triggers (just not row triggers) to read and modify the parent and child tables.

This allows most foreign key constraint actions to be implemented through their obvious after-row trigger, providing the constraint is not self-referential. Update cascade, update set null, update set default, delete set default, inserting a missing parent, and maintaining a count of children can all be implemented easily. For example, this is an implementation of update cascade:

```
create table p (p1 number constraint ppk primary key);
create table f (f1 number constraint ffk references p);
create trigger pt after update on p for each row begin
  update f set f1 = :new.p1 where f1 = :old.p1;
end;
/
```

This implementation requires care for multirow updates. For example, if a table p has three rows with the values (1), (2), (3), and table f also has three rows with the values (1), (2), (3), then the following statement updates p correctly but causes problems when the trigger updates f:

```
update p set p1 = p1+1;
```

The statement first updates (1) to (2) in p, and the trigger updates (1) to (2) in f, leaving two rows of value (2) in f. Then the statement updates (2) to (3) in p, and the trigger updates both rows of value (2) to (3) in f. Finally, the statement updates (3) to (4) in p, and the trigger updates all three rows in f from (3) to (4). The relationship of the data in p and f is lost.

To avoid this problem, you must forbid multirow updates to p that change the primary key and reuse existing primary key values. It could also be solved by tracking which foreign key values have already been updated, then modifying the trigger so that no row is updated twice.

That is the only problem with this technique for foreign key updates. The trigger cannot miss rows that have been changed but not committed by another transaction, because the foreign key constraint guarantees that no matching foreign key rows are locked before the after-row trigger is called.

System Trigger Restrictions

Depending on the event, different event attribute functions are available. For example, certain DDL operations may not be allowed on DDL events. Check "Event Attribute Functions" on page 10-2 before using an event attribute function, because its effects might be undefined rather than producing an error condition.

Only committed triggers are fired. For example, if you create a trigger that should be fired after all `CREATE` events, then the trigger itself does not fire after the creation, because the correct information about this trigger was not committed at the time when the trigger on `CREATE` events was fired.

For example, if you execute the following SQL statement:

```
CREATE OR REPLACE TRIGGER Foo AFTER CREATE ON DATABASE
BEGIN null;
END;
```

Then, trigger `foo` is not fired after the creation of `foo`. Oracle Database does not fire a trigger that is not committed.

Foreign Function Callouts

All restrictions on foreign function callouts also apply.

Who Is the Trigger User?

The following statement, inside a trigger, returns the owner of the trigger, not the name of user who is updating the table:

```
SELECT Username FROM USER_USERS;
```

Privileges Needed to Work with 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, or to reference a table in another schema from a trigger in your 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. In addition, the user creating the trigger must also have EXECUTE privilege on the referenced procedures, functions, or packages.

To create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, then you can drop the trigger, but not alter it.

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger owner, not the privilege domain of the user issuing the triggering statement. This is similar to the privilege model for stored procedures.

Compiling Triggers

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 statement is entered, 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, then the trigger is still created. If a DML statement fires this trigger, then the DML statement fails. (Runtime that

trigger errors always cause the DML statement to fail.) You can use the SHOW ERRORS statement 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 for Triggers

Compiled triggers have dependencies. They become invalid if a depended-on object, such as a stored procedure or 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 following statement shows the dependencies for the triggers in the SCOTT schema:

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
    FROM ALL_DEPENDENCIES
    WHERE OWNER = 'SCOTT' and TYPE = 'TRIGGER';
```

Triggers may depend on other functions or packages. If the function or package specified in the trigger is dropped, then the trigger is marked invalid. An attempt is made to validate the trigger on occurrence of the event. If the trigger cannot be validated successfully, then it is marked VALID WITH ERRORS, and the event fails.

Note:

- There is an exception for STARTUP events: STARTUP events succeed even if the trigger fails. There are also exceptions for SHUTDOWN events and for LOGON events if you login as SYSTEM.
- Because the DBMS_AQ package is used to enqueue a message, dependency between triggers and queues cannot be maintained.

Recompiling Triggers

Use the ALTER TRIGGER statement to recompile a trigger manually. For example, the following statement recompiles the PRINT_SALARY_CHANGES trigger:

```
ALTER TRIGGER Print_salary_changes COMPILE;
```

To recompile a trigger, you must own the trigger or have the ALTER ANY TRIGGER system privilege.

Modifying Triggers

Like a stored procedure, a trigger cannot be explicitly altered: It must be replaced with a new definition. (The ALTER TRIGGER statement 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 statement, and you can rerun the CREATE TRIGGER statement.

To drop a trigger, the trigger must be in your schema, or you must have the DROP ANY TRIGGER system privilege.

Debugging Triggers

You can debug a trigger using the same facilities available for stored procedures.

See Also: "Debugging Stored Procedures" on page 7-40

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 entered 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 entered and the trigger restriction (if any) evaluates to TRUE.

Enabling Triggers

By default, a trigger is automatically enabled when it is created; however, it can later be disabled. After 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 statement 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 statement 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;
```

Disabling Triggers

You might temporarily disable a trigger if:

- An object it references is not available.
- You need to perform a large data load, and you 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 statement 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 statement 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;
```

Viewing Information About Triggers

The following data dictionary views reveal information about triggers:

- USER_TRIGGERS

- ALL_TRIGGERS
- DBA_TRIGGERS

The new column, BASE_OBJECT_TYPE, specifies whether the trigger is based on DATABASE, SCHEMA, table, or view. The old column, TABLE_NAME, is null if the base object is not table or view.

The column ACTION_TYPE specifies whether the trigger is a call type trigger or a PL/SQL trigger.

The column TRIGGER_TYPE includes two additional values: BEFORE EVENT and AFTER EVENT, applicable only to system events.

The column TRIGGERING_EVENT includes all system and DML events.

> **See Also:** *Oracle Database Reference* for a complete description of these data dictionary views

For example, assume the following statement was used to create the REORDER trigger:

> **Caution:** You may need to set up data structures for certain examples to work:

```
CREATE OR REPLACE 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 Trigger_name = 'REORDER';

TYPE             TRIGGERING_STATEMENT       TABLE_NAME
---------------- -------------------------- ------------
AFTER EACH ROW   UPDATE                     INVENTORY

SELECT Trigger_body
   FROM USER_TRIGGERS
   WHERE Trigger_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 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

- Enable building complex views that are updatable
- Track system events

This section provides an example of each of these trigger applications. These examples are not meant to be used exactly as written: They are provided to assist you in designing your own triggers.

Auditing with Triggers: Example

Triggers are commonly used to supplement the built-in auditing features of Oracle Database. Although triggers can be written to record information similar to that recorded by the `AUDIT` statement, triggers should be used only when more detailed audit information is required. For example, use triggers to provide value-based auditing for each row.

Sometimes, the `AUDIT` statement 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 Database's auditing features provide, compared to auditing defined by triggers, as shown in Table 9–1.

Table 9–1 Comparison of Built-in Auditing and Trigger-Based Auditing

Audit Feature	Description
DML and DDL Auditing	Standard auditing options permit auditing of DML and DDL statements regarding all types of schema objects and structures. Comparatively, *triggers* permit auditing of DML statements entered against tables, and DDL auditing at `SCHEMA` or `DATABASE` level.
Centralized Audit Trail	All database audit information is recorded centrally and automatically using the auditing features of Oracle Database.
Declarative Method	Auditing features enabled using the standard Oracle Database 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.

Table 9–1 (Cont.) Comparison of Built-in Auditing and Trigger-Based Auditing

Audit Feature	Description
Session and Execution time Auditing	Using the database auditing features, records can be generated once every time an audited statement is entered (`BY ACCESS`) or once for every session that enters 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, unless autonomous transactions are used, any audit information generated by a trigger is rolled back if the triggering statement is rolled back. For more information on autonomous transactions, see *Oracle Database Concepts*.
Sessions can be Audited	Connections and disconnections, as well as session activity (physical I/Os, logical I/Os, deadlocks, and so on), 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.

Choosing between `AFTER` row and `AFTER` statement triggers depends on the information being audited. For example, row triggers provide value-based auditing for each table row. 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_tab` table for each row. It requires that a "reason code" be stored in a global package variable before the update. This shows how triggers can be used to provide value-based auditing and how to use public package variables.

Note: You may need to set up the following data structures for the examples to work:

```
CREATE OR REPLACE PACKAGE Auditpackage AS
   Reason VARCHAR2(10);
PROCEDURE Set_reason(Reason VARCHAR2);
END;
CREATE TABLE Emp99 (
   Empno               NOT NULL   NUMBER(4),
   Ename               VARCHAR2(10),
   Job                 VARCHAR2(9),
   Mgr                 NUMBER(4),
   Hiredate            DATE,
   Sal                 NUMBER(7,2),
   Comm                NUMBER(7,2),
   Deptno              NUMBER(2),
   Bonus               NUMBER,
   Ssn                 NUMBER,
   Job_classification  NUMBER);

CREATE TABLE Audit_employee (
   Oldssn              NUMBER,
   Oldname             VARCHAR2(10),
   Oldjob              VARCHAR2(2),
   Oldsal              NUMBER,
   Newssn              NUMBER,
   Newname             VARCHAR2(10),
   Newjob              VARCHAR2(2),
   Newsal              NUMBER,
   Reason              VARCHAR2(10),
   User1               VARCHAR2(10),
   Systemdate          DATE);
```

```
CREATE OR REPLACE TRIGGER Audit_employee
AFTER INSERT OR DELETE OR UPDATE ON Emp99
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 preceding 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_tab 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.Ename, :old.Job_classification, :old.Sal,
   :new.Ssn, :new.Ename, :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 run:

```
CREATE OR REPLACE TRIGGER Audit_employee_reset
AFTER INSERT OR DELETE OR UPDATE ON Emp_tab
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.

This next trigger also uses triggers to do auditing. It tracks changes made to the Emp_tab table and stores this information in AUDIT_TABLE and AUDIT_TABLE_VALUES.

Note: You may need to set up the following data structures for the example to work:

```
CREATE TABLE Audit_table (
   Seq      NUMBER,
   User_at  VARCHAR2(10),
   Time_now DATE,
   Term     VARCHAR2(10),
   Job      VARCHAR2(10),
   Proc     VARCHAR2(10),
   enum     NUMBER);
CREATE SEQUENCE Audit_seq;
CREATE TABLE Audit_table_values (
   Seq      NUMBER,
   Dept     NUMBER,
   Dept1    NUMBER,
   Dept2    NUMBER);
```

```
CREATE OR REPLACE TRIGGER Audit_emp
   AFTER INSERT OR UPDATE OR DELETE ON Emp_tab
   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_tab', '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_tab', '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_tab', '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: Examples

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 3, "Maintaining Data Integrity Through Constraints"

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 Database'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 Database 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 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.

Referential Integrity Using Triggers

There are many cases where referential integrity can be enforced using triggers. Note, however, you should only use triggers when there is no declarative support for the action you are performing.

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, then 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_tab and Dept_tab 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 following example 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 OR REPLACE TRIGGER Emp_dept_check
BEFORE INSERT OR UPDATE OF Deptno ON Emp_tab
FOR EACH ROW WHEN (new.Deptno IS NOT NULL)

-- Before a row is inserted, or DEPTNO is updated in the Emp_tab
-- table, fire this trigger to verify that the new foreign
-- key value (DEPTNO) is present in the Dept_tab table.
DECLARE
   Dummy              INTEGER;  -- to be used for cursor fetch
   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 Dummy_cursor (Dn NUMBER) IS
   SELECT Deptno FROM Dept_tab
      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 Parent Table The following trigger is defined on the DEPT_TAB table to enforce the UPDATE and DELETE RESTRICT referential action on the primary key of the DEPT_TAB table:

```
CREATE OR REPLACE TRIGGER Dept_restrict
BEFORE DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, check for dependent
-- foreign key values in Emp_tab; rollback if any are found.
DECLARE
   Dummy                INTEGER;      -- to be used for cursor fetch
   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_tab 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;
```

> **Caution:** This trigger does not work with self-referential tables (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: Example The following trigger is defined on the DEPT_TAB table to enforce the UPDATE and DELETE SET NULL referential action on the primary key of the DEPT_TAB table:

```
CREATE OR REPLACE TRIGGER Dept_set_null
AFTER DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, set all corresponding
-- dependent foreign key values in Emp_tab to NULL:
BEGIN
   IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
      UPDATE Emp_tab SET Emp_tab.Deptno = NULL
         WHERE Emp_tab.Deptno = :old.Deptno;
   END IF;
END;
```

DELETE Cascade Trigger for Parent Table: Example The following trigger on the DEPT_TAB table enforces the DELETE CASCADE referential action on the primary key of the DEPT_TAB table:

```
CREATE OR REPLACE TRIGGER Dept_del_cascade
AFTER DELETE ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab, delete all
-- rows from the Emp_tab table whose DEPTNO is the same as
-- the DEPTNO being deleted from the Dept_tab table:
BEGIN
   DELETE FROM Emp_tab
      WHERE Emp_tab.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: Example The following trigger ensures that if a department number is updated in the Dept_tab table, then this change is propagated to dependent foreign keys in the Emp_tab 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 OR REPLACE PACKAGE Integritypackage AS
   Updateseq NUMBER;
END Integritypackage;

CREATE OR REPLACE PACKAGE BODY Integritypackage AS
END Integritypackage;
-- create flag col:
ALTER TABLE Emp_tab ADD Update_id NUMBER;

CREATE OR REPLACE TRIGGER Dept_cascade1 BEFORE UPDATE OF Deptno ON Dept_tab
DECLARE
   Dummy NUMBER;

-- Before updating the Dept_tab 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 OR REPLACE TRIGGER Dept_cascade2 AFTER DELETE OR UPDATE
   OF Deptno ON Dept_tab FOR EACH ROW

-- For each department number in Dept_tab that is updated,
-- cascade the update to dependent foreign keys in the
```

```
-- Emp_tab table.  Only cascade the update if the child row
-- has not already been updated by this trigger:
BEGIN
   IF UPDATING THEN
      UPDATE Emp_tab
         SET Deptno = :new.Deptno,
         Update_id = Integritypackage.Updateseq   --from 1st
         WHERE Emp_tab.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_tab, delete all
   -- rows from the Emp_tab table whose DEPTNO is the same as
   -- the DEPTNO being deleted from the Dept_tab table:
      DELETE FROM Emp_tab
      WHERE Emp_tab.Deptno = :old.Deptno;
   END IF;
END;
CREATE OR REPLACE TRIGGER Dept_cascade3 AFTER UPDATE OF Deptno ON Dept_tab
BEGIN  UPDATE Emp_tab
   SET Update_id = NULL
   WHERE Update_id = Integritypackage.Updateseq;
END;
```

Note: Because this trigger updates the `Emp_tab` 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 always work properly in your environment.

Trigger for Complex Check Constraints: Example

Triggers can enforce integrity rules other than referential integrity. For example, this trigger performs a complex check before allowing the triggering statement to run.

Note: You may need to set up the following data structures for the example to work:

```
CREATE TABLE Salgrade (
   Grade                NUMBER,
   Losal                NUMBER,
   Hisal                NUMBER,
   Job_classification   NUMBER)
```

```
CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT OR UPDATE OF Sal, Job ON Emp99
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.Ename);
   WHEN NO_DATA_FOUND THEN
      Raise_application_error(-20322,
```

```
        'Invalid Job Classification '
        ||:new.Job_classification);
END;
```

Complex Security Authorizations and Triggers: Example

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 Database. For example, a trigger can prohibit updates to salary data of the `Emp_tab` 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 run, 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.

This example shows a trigger used to enforce security.

> **Note:** You may need to set up the following data structures for the example to work:
>
> ```
> CREATE TABLE Company_holidays (Day DATE);
> ```

```
CREATE OR REPLACE TRIGGER Emp_permit_changes
BEFORE INSERT OR DELETE OR UPDATE ON Emp99
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_tab table during non-working hours');
END;
```

> **See Also:** *Oracle Database Security Guide* for details on database security features

Transparent Event Logging and Triggers

Triggers are very useful when you want to transparently perform a related change in the database following certain events.

The `REORDER` trigger example shows a trigger that reorders parts as necessary when certain conditions are met. (In other words, a triggering statement is entered, and the `PARTS_ON_HAND` value is less than the `REORDER_POINT` value.)

Derived Column Values and Triggers: Example

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 for the following reasons:

- 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.

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.

> **Note:** You may need to set up the following data structures for the example to work:
>
> ```
> ALTER TABLE Emp99 ADD(
>    Uppername   VARCHAR2(20),
>    Soundexname VARCHAR2(20));
> ```

```
CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp99

/* 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;
```

Building Complex Updatable Views Using Triggers: Example

Views are an excellent mechanism to provide logical windows over table data. However, when the view query gets complex, the system implicitly cannot translate the DML on the view into those on the underlying tables. `INSTEAD OF` triggers help solve this problem. These triggers can be defined over views, and they fire *instead* of the actual DML.

Consider a library system where books are arranged under their respective titles. The library consists of a collection of book type objects. The following example explains the schema.

```
CREATE OR REPLACE TYPE Book_t AS OBJECT
(
   Booknum   NUMBER,
   Title     VARCHAR2(20),
   Author    VARCHAR2(20),
   Available CHAR(1)
);
CREATE OR REPLACE TYPE Book_list_t AS TABLE OF Book_t;
```

Assume that the following tables exist in the relational schema:

```
Table Book_table (Booknum, Section, Title, Author, Available)
```

Booknum	Section	Title	Author	Available
121001	Classic	Iliad	Homer	Y
121002	Novel	Gone With the Wind	Mitchell M	N

Library consists of `library_table(section)`.

Section
Geography
Classic

You can define a complex view over these tables to create a logical view of the library with sections and a collection of books in each section.

```
CREATE OR REPLACE VIEW Library_view AS
SELECT i.Section, CAST (MULTISET (
   SELECT b.Booknum, b.Title, b.Author, b.Available
   FROM Book_table b
   WHERE b.Section = i.Section) AS Book_list_t) BOOKLIST
FROM Library_table i;
```

Make this view updatable by defining an `INSTEAD OF` trigger over the view.

```
CREATE OR REPLACE TRIGGER Library_trigger INSTEAD OF INSERT ON Library_view FOR
EACH ROW
   Bookvar BOOK_T;
   i       INTEGER;
BEGIN
   INSERT INTO Library_table VALUES (:NEW.Section);
   FOR i IN 1..:NEW.Booklist.COUNT LOOP
      Bookvar := Booklist(i);
      INSERT INTO book_table
         VALUES ( Bookvar.booknum, :NEW.Section, Bookvar.Title, Bookvar.Author,
bookvar.Available);
   END LOOP;
END;
/
```

The library_view is an updatable view, and any INSERTs on the view are handled by the trigger that gets fired automatically. For example:

```
INSERT INTO Library_view VALUES ('History', book_list_t(book_t(121330,
'Alexander', 'Mirth', 'Y');
```

Similarly, you can also define triggers on the nested table booklist to handle modification of the nested table element.

Tracking System Events Using Triggers

Fine-Grained Access Control Using Triggers: Example System triggers can be used to set application context. Application context is a relatively new feature that enhances your ability to implement fine-grained access control. Application context is a secure session cache, and it can be used to store session-specific attributes.

In the example that follows, procedure set_ctx sets the application context based on the user profile. The trigger setexpensectx ensures that the context is set for every user.

```
CONNECT secdemo/secdemo

CREATE OR REPLACE CONTEXT Expenses_reporting USING Secdemo.Exprep_ctx;

REM ================================================================
REM Creation of the package which implements the context:
REM ================================================================

CREATE OR REPLACE PACKAGE Exprep_ctx AS
 PROCEDURE Set_ctx;
END;

SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY Exprep_ctx IS
   PROCEDURE Set_ctx IS
      Empnum   NUMBER;
      Countrec NUMBER;
      Cc       NUMBER;
      Role     VARCHAR2(20);
   BEGIN

      -- SET emp_number:
      SELECT Employee_id INTO Empnum FROM Employee
```

```
         WHERE Last_name = SYS_CONTEXT('userenv', 'session_user');

      DBMS_SESSION.SET_CONTEXT('expenses_reporting','emp_number', Empnum);

      -- SET ROLE:
      SELECT COUNT (*) INTO Countrec FROM Cost_center WHERE Manager_id=Empnum;
      IF (countrec > 0) THEN
         DBMS_SESSION.SET_CONTEXT('expenses_reporting','exp_role','MANAGER');
      ELSE
         DBMS_SESSION.SET_CONTEXT('expenses_reporting','exp_role','EMPLOYEE');
      END IF;

      -- SET cc_number:
      SELECT Cost_center_id INTO Cc FROM Employee
         WHERE Last_name = SYS_CONTEXT('userenv','session_user');
      DBMS_SESSION.SET_CONTEXT(expenses_reporting','cc_number',Cc);
   END;
END;
```

CALL Syntax

```
CREATE OR REPLACE TRIGGER Secdemo.Setexpseetx
AFTER LOGON ON DATABASE
CALL Secdemo.Exprep_etx.Set_otx
```

Responding to System Events through Triggers

Oracle Database's system event publication lets applications subscribe to database events, just like they subscribe to messages from other applications.

See Also: Chapter 10, "Working With System Events"

Oracle Database's system events publication framework includes the following features:

- Infrastructure for publish/subscribe, by making the database an active publisher of events.
- Integration of data cartridges in the server. The system events publication can be used to notify cartridges of state changes in the server.
- Integration of fine-grained access control in the server.

By creating a trigger, you can specify a procedure that runs when an event occurs. DML events are supported on tables, and system events are supported on

DATABASE and SCHEMA. You can turn notification on and off by enabling and disabling the trigger using the ALTER TRIGGER statement.

This feature is integrated with the Advanced Queueing engine. Publish/subscribe applications use the DBMS_AQ.ENQUEUE() procedure, and other applications such as cartridges use callouts.

See Also:

- *Oracle Database SQL Reference*
- *Oracle Streams Advanced Queuing User's Guide and Reference* for details on how to subscribe to published events

How Events Are Published Through Triggers

When events are detected by the server, the trigger mechanism executes the action specified in the trigger. As part of this action, you can use the DBMS_AQ package to publish the event to a queue, so that subscribers get notifications.

Note: Only system-defined database events can be detected this way. You cannot define your own event conditions.

When an event occurs, all triggers that are enabled on that event are fired, with some exceptions:

- If the trigger is actually the target of the triggering event, it is not fired. For example, a trigger for all DROP events is not fired when it is dropped itself.
- If a trigger is not fired if it has been modified but not committed within the same transaction as the firing event. For example, recursive DDL within a system trigger might modify a trigger, which prevents the modified trigger from being fired by events within the same transaction.

More than one trigger can be created on an object. When an event fires more than one trigger, the order is not defined and you should not rely on the triggers being fired in a particular order.

Publication Context

When an event is published, certain runtime context and attributes, as specified in the parameter list, are passed to the callout procedure. A set of functions called event attribute functions are provided.

> **See Also:** "Event Attribute Functions" on page 10-2 for information on event-specific attributes

For each system event supported, event-specific attributes are identified and predefined for the event. You can choose the parameter list to be any of these attributes, along with other simple expressions. For callouts, these are passed as `IN` arguments.

Error Handling

Return status from publication callout functions for all events are ignored. For example, with `SHUTDOWN` events, the server cannot do anything with the return status.

> **See Also:** "List of Database Events" on page 10-7 for details on return status

Execution Model

Traditionally, triggers execute as the definer of the trigger. The trigger action of an event is executed as the definer of the action (as the definer of the package or function in callouts, or as owner of the trigger in queues). Because the owner of the trigger must have `EXECUTE` privileges on the underlying queues, packages, or procedure, this behavior is consistent.

10

Working With System Events

System events, like `LOGON` and `SHUTDOWN`, provide a mechanism for tracking system changes. Oracle Database lets you combine this tracking with database event notification, which provides a simple and elegant method of delivering asynchronous messaging to an application.

This chapter includes descriptions of the various events on which triggers can be created. It also provides the list of event attribute functions. Topics include the following:

- Event Attribute Functions
- List of Database Events

See Also:

- Chapter 9, "Using Triggers" for background information useful for understanding the information in this chapter. You access event information inside a trigger body.
- *Oracle 2 Day DBA* for information on Oracle Enterprise Manager. You can set up a flexible system for handling events using the Event panels in Enterprise Manager. This technique lets you detect more conditions than the system events in this chapter, and it lets you set up actions such as invoking shell scripts.

Event Attribute Functions

When a trigger is fired, you can retrieve certain attributes about the event that fired the trigger. Each attribute is retrieved by a function call. Table 10–1 describes the system-defined event attributes.

Note:

- To make these attributes available, you must first run the `CATPROC.SQL` script.
- The trigger dictionary object maintains metadata about events that will be published and their corresponding attributes.
- In earlier releases, these functions were accessed through the `SYS` package. We recommend you use these public synonyms whose names begin with `ora_`.

Table 10–1 System-Defined Event Attributes

Attribute	Type	Description	Example
`ora_client_ip_address`	`VARCHAR2`	Returns the IP address of the client in a LOGON event, when the underlying protocol is TCP/IP	`if (ora_sysevent = 'LOGON') then addr := ora_client_ip_ address; end if;`
`ora_database_name`	`VARCHAR2(50)`	Database name.	`DECLARE db_name VARCHAR2(50); BEGIN db_name := ora_database_name; END;`
`ora_des_encrypted_password`	`VARCHAR2`	The DES encrypted password of the user being created or altered.	`IF (ora_dict_obj_type = 'USER') THEN INSERT INTO event_table (ora_des_encrypted_password); END IF;`

Table 10–1 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
ora_dict_obj_name	VARCHAR(30)	Name of the dictionary object on which the DDL operation occurred.	INSERT INTO event_table ('Changed object is ' || ora_ dict_obj_name');
ora_dict_obj_name_list (name_list OUT ora_name_ list_t)	BINARY_INTEGER	Return the list of object names of objects being modified in the event.	if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_modified := ora_ dict_obj_name_list (name_list); end if;
ora_dict_obj_owner	VARCHAR(30)	Owner of the dictionary object on which the DDL operation occurred.	INSERT INTO event_table ('object owner is' || ora_dict_obj_ owner');
ora_dict_obj_owner_ list(owner_list OUT ora_ name_list_t)	BINARY_INTEGER	Returns the list of object owners of objects being modified in the event.	if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_of_modified_ objects := ora_dict_obj_owner_ list(owner_list); end if;
ora_dict_obj_type	VARCHAR(20)	Type of the dictionary object on which the DDL operation occurred.	INSERT INTO event_table ('This object is a ' || ora_dict_obj_ type);
ora_grantee(user_list OUT ora_name_list_t)	BINARY_INTEGER	Returns the grantees of a grant event in the OUT parameter; returns the number of grantees in the return value.	if (ora_sysevent = 'GRANT') then number_of_users := ora_ grantee(user_list); end if;
ora_instance_num	NUMBER	Instance number.	IF (ora_instance_num = 1) THEN INSERT INTO event_table ('1'); END IF;

Table 10–1 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
ora_is_alter_column(column_ name IN VARCHAR2)	BOOLEAN	Returns true if the specified column is altered.	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then alter_column := ora_is_ alter_column('FOO'); end if;
ora_is_creating_nested_table	BOOLEAN	Returns true if the current event is creating a nested table	if (ora_sysevent = 'CREATE' and ora_dict_obj_type = 'TABLE' and ora_is_creating_nested_table) then insert into event_tab values ('A nested table is created'); end if;
ora_is_drop_column(column_ name IN VARCHAR2)	BOOLEAN	Returns true if the specified column is dropped.	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then drop_column := ora_is_ drop_column('FOO'); end if;
ora_is_servererror	BOOLEAN	Returns TRUE if given error is on error stack, FALSE otherwise.	IF (ora_is_servererror(*error_ number*)) THEN INSERT INTO event_table ('Server error!!'); END IF;
ora_login_user	VARCHAR2(30)	Login user name.	SELECT ora_login_user FROM dual;
ora_partition_pos	BINARY_INTEGER	In an INSTEAD OF trigger for CREATE TABLE, the position within the SQL text where you could insert a PARTITION clause.	-- Retrieve ora_sql_txt into -- sql_text variable first. n := ora_partition_pos; new_stmt := substr(sql_text, 1, n-1) || ' ' || my_partition_clause || ' ' || substr(sql_text, n));

Table 10–1 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
ora_privilege_list(privilege_list OUT ora_name_ list_t)	BINARY_INTEGER	Returns the list of privileges being granted by the grantee or the list of privileges revoked from the revokees in the OUT parameter; returns the number of privileges in the return value.	if (ora_sysevent = 'GRANT' or ora_sysevent = 'REVOKE') then number_of_privileges := ora_privilege_list(priv_list); end if;
ora_revokee (user_list OUT ora_name_list_ t)	BINARY_INTEGER	Returns the revokees of a revoke event in the OUT parameter; returns the number of revokees in the return value.	if (ora_sysevent = 'REVOKE') then number_of_users := ora_ revokee(user_list);
ora_server_error	NUMBER	Given a position (1 for top of stack), it returns the error number at that position on error stack	INSERT INTO event_table ('top stack error ' || ora_server_ error(1));
ora_server_error_depth	BINARY_INTEGER	Returns the total number of error messages on the error stack.	n := ora_server_error_depth; -- This value is used with -- other functions such as -- ora_server_error
ora_server_error_msg (position in binary_integer)	VARCHAR2	Given a position (1 for top of stack), it returns the error message at that position on error stack	INSERT INTO event_table ('top stack error message' || ora_ server_error_msg(1));

Table 10–1 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
`ora_server_error_num_params (position in binary_integer)`	`BINARY_INTEGER`	Given a position (1 for top of stack), it returns the number of strings that have been substituted into the error message using a format like "%s".	`n := ora_server_error_num_ params(1);`
`ora_server_error_param (position in binary_integer, param in binary_integer)`	`VARCHAR2`	Given a position (1 for top of stack) and a parameter number, returns the matching "%s", "%d", and so on substitution value in the error message.	`-- E.g. the 2rd %s in a message -- like "Expected %s, found %s" param := ora_server_error_ param(1,2);`
`ora_sql_txt (sql_text out ora_name_list_t)`	`BINARY_INTEGER`	Returns the SQL text of the triggering statement in the OUT parameter. If the statement is long, it is broken up into multiple PL/SQL table elements. The function return value specifies how many elements are in the PL/SQL table.	`sql_text ora_name_list_t; stmt VARCHAR2(2000); ... n := ora_sql_txt(sql_text); FOR i IN 1..n LOOP  stmt := stmt || sql_text(i); END LOOP; INSERT INTO event_table ('text of triggering statement: ' || stmt);`

Table 10–1 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
ora_sysevent	VARCHAR2(20)	System event firing the trigger: Event name is same as that in the syntax.	INSERT INTO event_table (ora_ sysevent);
ora_with_grant_option	BOOLEAN	Returns true if the privileges are granted with grant option.	if (ora_sysevent = 'GRANT' and ora_with_grant_option = TRUE) then insert into event_table ('with grant option'); end if;
space_error_info(error_number OUT NUMBER, error_type OUT VARCHAR2, object_owner OUT VARCHAR2, table_space_name OUT VARCHAR2, object_name OUT VARCHAR2, sub_object_name OUT VARCHAR2)	BOOLEAN	Returns true if the error is related to an out-of-space condition, and fills in the OUT parameters with information about the object that caused the error.	if (space_error_info(eno, typ, owner, ts, obj, subobj) = TRUE) then dbms_output.put_line('The object ' || obj || ' owned by ' || owner || ' has run out of space.'); end if;

List of Database Events

System Events

System events are related to entire instances or schemas, not individual tables or rows. Triggers created on startup and shutdown events must be associated with the database instance. Triggers created on error and suspend events can be associated with either the database instance or a particular schema.

Table 10–2 contains a list of system manager events.

Table 10–2 System Manager Events

Event	When Fired?	Conditions	Restrictions	Transaction	Attribute Functions
STARTUP	When the database is opened.	None allowed	No database operations allowed in the trigger. Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name
SHUTDOWN	Just before the server starts the shutdown of an instance. This lets the cartridge shutdown completely. For abnormal instance shutdown, this event may not be fired.	None allowed	No database operations allowed in the trigger. Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name
SERVERERROR	When the error eno occurs. If no condition is given, then this event fires when any error occurs. Does not apply to ORA-1034, ORA-1403, ORA-1422, ORA-1423, and ORA-4030 conditions, because they are not true errors or are too serious to continue processing.	ERRNO = eno	Depends on the error. Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror space_error_info

Client Events

Client events are the events related to user logon/logoff, DML, and DDL operations. For example:

```
CREATE OR REPLACE TRIGGER On_Logon
  AFTER LOGON
  ON The_user.Schema
BEGIN
  Do_Something;
END;
```

The LOGON and LOGOFF events allow simple conditions on UID( ) and USER( ). All other events allow simple conditions on the type and name of the object, as well as functions like UID( ) and USER( ).

The LOGON event starts a separate transaction and commits it after firing the triggers. All other events fire the triggers in the existing user transaction.

The LOGON and LOGOFF events can operate on any objects. For all other events, the corresponding trigger cannot perform any DDL operations, such as DROP and ALTER, on the object that caused the event to be generated.

The DDL allowed inside these triggers is altering, creating, or dropping a table, creating a trigger, and compile operations.

If an event trigger becomes the target of a DDL operation (such as CREATE TRIGGER), it cannot be fired later during the same transaction

Table 10–3 contains a list of client events.

Table 10–3 Client Events

Event	When Fired?	Attribute Functions
BEFORE ALTER AFTER ALTER	When a catalog object is altered.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_password (for ALTER USER events) ora_is_alter_column, ora_is_drop_column (for ALTER TABLE events)
BEFORE DROP AFTER DROP	When a catalog object is dropped.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner
BEFORE ANALYZE AFTER ANALYZE	When an analyze statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE ASSOCIATE STATISTICS AFTER ASSOCIATE STATISTICS	When an associate statistics statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list
BEFORE AUDIT AFTER AUDIT BEFORE NOAUDIT AFTER NOAUDIT	When an audit or noaudit statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name

Table 10–3 (Cont.) Client Events (Cont.)

Event	When Fired?	Attribute Functions
BEFORE COMMENT AFTER COMMENT	When an object is commented	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE CREATE AFTER CREATE	When a catalog object is created.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)
BEFORE DDL AFTER DDL	When most SQL DDL statements are issued. Not fired for ALTER DATABASE, CREATE CONTROLFILE, CREATE DATABASE, and DDL issued through the PL/SQL procedure interface, such as creating an advanced queue.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE DISASSOCIATE STATISTICS AFTER DISASSOCIATE STATISTICS	When a disassociate statistics statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list

Table 10–3 (Cont.) Client Events (Cont.)

Event	When Fired?	Attribute Functions
BEFORE GRANT AFTER GRANT	When a grant statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_grantee ora_with_grant_option ora_privileges
BEFORE LOGOFF	At the start of a user logoff	ora_sysevent ora_login_user ora_instance_num ora_database_name
AFTER LOGON	After a successful logon of a user.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address
BEFORE RENAME AFTER RENAME	When a rename statement is issued.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type

Table 10–3 (Cont.) Client Events (Cont.)

Event	When Fired?	Attribute Functions
BEFORE REVOKE AFTER REVOKE	When a revoke statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privileges
AFTER SUSPEND	After a SQL statement is suspended because of an out-of-space condition. The trigger should correct the condition so the statement can be resumed.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror space_error_info
BEFORE TRUNCATE AFTER TRUNCATE	When an object is truncated	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

11

Using the Publish-Subscribe Model for Applications

Because the database is the most significant resource of information within the enterprise, Oracle created a publish-subscribe solution for enterprise information delivery and messaging to complement this role. Topics in this chapter include:

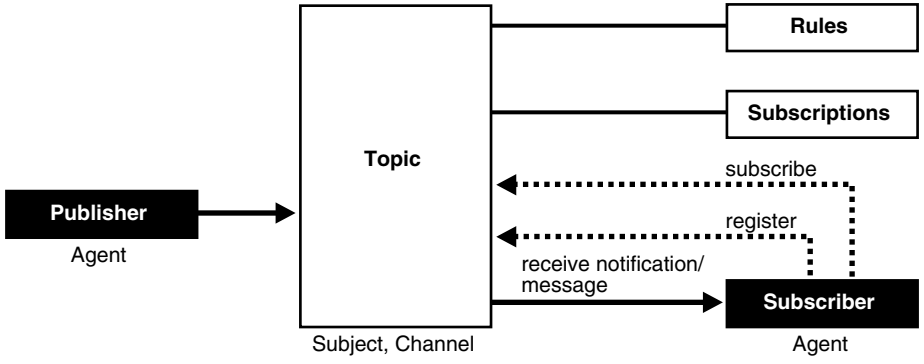
Introduction to Publish-Subscribe

Networking technologies and products enable a high degree of connectivity across a large number of computers, applications, and users. In these environments, it is important to provide asynchronous communications for the class of distributed systems that operate in a loosely-coupled and autonomous fashion, and which require operational immunity from network failures. This requirement has been filled by various middleware products that are characterized as messaging, message oriented middleware (MOM), message queuing, or publish-subscribe.

Applications that communicate through a publish and subscribe paradigm require the sending applications (publishers) to publish messages without explicitly specifying recipients or having knowledge of intended recipients. Similarly, receiving applications (subscribers) must receive only those messages that the subscriber has registered an interest in.

This decoupling between senders and recipients is usually accomplished by an intervening entity between the publisher and the subscriber, which serves as a level of indirection. This intervening entity is a queue that represents a subject or channel. Figure 11–1 illustrates publish and subscribe functionality.

Figure 11–1 Oracle Publish-Subscribe Functionality



A subscriber subscribes to a queue by expressing interest in messages enqueued to that queue and by using a subject- or content-based rule as a filter. This results in a set of rule-based subscriptions associated with a given queue.

At runtime, publishers post messages to various queues. The queue (in other words, the delivery mechanisms of the underlying infrastructure) then delivers messages that match the various subscriptions to the appropriate subscribers.

Publish-Subscribe Architecture

Oracle Database includes the following features to support database-enabled publish-subscribe messaging:

- Database Events
- Advanced Queuing
- Client Notifications

Database Events

Database events support declarative definitions for publishing database events, detection, and run-time publication of such events. This feature enables active publication of information to end-users in an event-driven manner, to complement the traditional pull-oriented approaches to accessing information.

See Also: Chapter 10, "Working With System Events"

Advanced Queuing

Oracle Advanced Queuing (AQ) supports a queue-based publish-subscribe paradigm. Database queues serve as a durable store for messages, along with capabilities to allow publish and subscribe based on queues. A rules-engine and subscription service dynamically route messages to recipients based on expressed interest. This allows decoupling of addressing between senders and receivers to complement the existing explicit sender-receiver message addressing.

See Also: *Oracle Streams Advanced Queuing User's Guide and Reference*

Client Notifications

Client notifications support asynchronous delivery of messages to interested subscribers. This enables database clients to register interest in certain queues, and it enables these clients to receive notifications when publications on such queues occur. Asynchronous delivery of messages to database clients is in contrast to the traditional polling techniques used to retrieve information.

See Also: *Oracle Call Interface Programmer's Guide*

Publish-Subscribe Concepts

This section describes various concepts related to publish-subscribe.

queue

A queue is an entity that supports the notion of named subjects of interest. Queues can be characterized as:

non-persistent queue (lightweight queue)

The underlying queue infrastructure pushes the messages published to connected clients in a lightweight, at-best-once, manner.

persistent queue

Queues serve as durable containers for messages. Messages are delivered in a deferred and reliable mode.

agent

Publishers and subscribers are internally represented as agents. There is a distinction between an agent and a client.

An **agent** is a persistent logical subscribing entity that expresses interest in a queue through a subscription. An agent has properties, such as an associated subscription, an address, and a delivery mode for messages. In this context, an agent is an electronic proxy for a publisher or subscriber.

A **client** is a transient physical entity. The attributes of a client include the physical process where the client programs run, the node name, and the client application logic. There could be several clients acting on behalf of a single agent. Also, the same client, if authorized, can act on behalf of multiple agents.

rule on a queue

A rule on a queue is specified as a conditional expression using a predefined set of operators on the message format attributes or on the message header attributes. Each queue has an associated message content format that describes the structure of the messages represented by that queue. The message format may be unstructured (`RAW`) or it may have a well-defined structure (ADT). This allows both subject- or content-based subscriptions.

subscriber

Subscribers (agents) may specify subscriptions on a queue using a rule. Subscribers are durable and are stored in a catalog.

database event publication framework

The database represents a significant source for publishing information. An event framework is proposed to allow declarative definition of database event publication. As these pre-defined events occur, the framework detects and publishes such events. This allows active delivery of information to end-users in an event-driven manner as part of the publish-subscribe capability.

registration

Registration is the process of associated delivery information by a given client, acting on behalf of an agent. There is an important distinction between the subscription and registration related to the agent/client separation.

Subscription indicates an interest in a particular queue by an agent. It does not specify where and how delivery must occur. Delivery information is a physical property that is associated with a client, and it is a transient manifestation of the logical agent (the subscriber). A specific client process acting on behalf of an agent registers delivery information by associating a host and port, indicating *where* the delivery should be done, and a callback, indicating *how* there delivery should be done.

publishing a message

Publishers publish messages to queues by using the appropriate queuing interfaces. The interfaces may depend on which model the queue is implemented on. For example, an enqueue call represents the publishing of a message.

rules engine

When a message is posted or published to a given queue, a rules engine extracts the set of candidate rules from all rules defined on that queue that match the published message.

subscription services

Corresponding to the list of candidate rules on a given queue, the set of subscribers that match the candidate rules can be evaluated. In turn, the set of agents corresponding to this subscription list can be determined and notified.

posting

The queue notifies all registered clients of the appropriate published messages. This concept is called **posting**. When the queue needs to notify all interested clients, it posts the message to all registered clients.

receive a message

A subscriber may receive messages through any of the following mechanisms:

- A client process acting on behalf of the subscriber specifies a callback using the registration mechanism. The posting mechanism then asynchronously invokes the callback when a message matches the subscriber's subscription. The message content may be passed to the callback function (non-persistent queues only).
- A client process acting on behalf of the subscriber specifies a callback using the registration mechanism. The posting mechanism then asynchronously invokes the callback function, but without the full message content. This serves as a notification to the client, which subsequently retrieves the message content in a pull fashion (persistent queues only).
- A client process acting on behalf of the subscriber simply retrieves messages from the queue in a periodic, or some other appropriate, manner. While the messages are deferred, there is no asynchronous delivery to the end-client.

Examples of a Publish-Subscribe Mechanism

Note: You may need to set up data structures, similar to the following, for certain examples to work:

```
CONNECT system/manager
DROP USER pubsub CASCADE;
CREATE USER pubsub IDENTIFIED BY pubsub;
GRANT CONNECT, RESOURCE TO pubsub;
GRANT EXECUTE ON DBMS_AQ to pubsub;
GRANT EXECUTE ON DBMS_AQADM to pubsub;
GRANT AQ_ADMINISTRATOR_ROLE TO pubsub;
CONNECT pubsub/pubsub
```

Scenario: This example shows how system events, client notification, and AQ work together to implement publish-subscribe.

- Create under the user schema, `pubsub`, with all objects necessary to support a publish-subscribe mechanism. In this particular code, the Agent `snoop` subscribe to messages that are published at logon events. Note that the user `pubsub` needs `AQ_ADMINISTRATOR_ROLE` privileges to use AQ functionality.

```
Rem ----------------------------------------------------
REM create queue table for persistent multiple consumers:
```

```
Rem ------------------------------------------------------

CONNECT pubsub/pubsub;

Rem  Create or replace a queue table
BEGIN
DBMS_AQADM.CREATE_QUEUE_TABLE(
   Queue_table        =>  'Pubsub.Raw_msg_table',
   Multiple_consumers =>   TRUE,
   Queue_payload_type =>  'RAW',
   Compatible         =>  '8.1');
END;
/
Rem ------------------------------------------------------
Rem  Create a persistent queue for publishing messages:
Rem ------------------------------------------------------

Rem  Create a queue for logon events
begin
BEGIN
   DBMS_AQADM.CREATE_QUEUE(
         Queue_name     =>   'Pubsub.Logon',
         Queue_table    =>   'Pubsub.Raw_msg_table',
         Comment        =>   'Q for error triggers');
END;
/

Rem ------------------------------------------------------
Rem  Start the queue:
Rem ------------------------------------------------------

BEGIN
   DBMS_AQADM.START_QUEUE('pubsub.logon');
END;
/

Rem ------------------------------------------------------
Rem  define new_enqueue for convenience:
Rem ------------------------------------------------------

CREATE OR REPLACE PROCEDURE New_enqueue(
               Queue_name      IN VARCHAR2,
               Payload         IN RAW ,
               Correlation     IN VARCHAR2 := NULL,
               Exception_queue IN VARCHAR2 := NULL)
```

```
AS

Enq_ct     DBMS_AQ.Enqueue_options_t;
Msg_prop   DBMS_AQ.Message_properties_t;
Enq_msgid  RAW(16);
Userdata   RAW(1000);

BEGIN
   Msg_prop.Exception_queue := Exception_queue;
   Msg_prop.Correlation := Correlation;
   Userdata := Payload;

DBMS_AQ.ENQUEUE(Queue_name, Enq_ct, Msg_prop, Userdata, Enq_msgid);
END;
/

Rem ----------------------------------------------------
Rem  add subscriber with rule based on current user name,
Rem  using correlation_id
Rem ----------------------------------------------------


DECLARE
Subscriber Sys.Aq$_agent;
BEGIN
   Subscriber := sys.aq$_agent('SNOOP', NULL, NULL);
DBMS_AQADM.ADD_SUBSCRIBER(
    Queue_name         => 'Pubsub.logon',
    Subscriber         => subscriber,
    Rule               => 'CORRID = ''SCOTT'' ');
END;
/

Rem ----------------------------------------------------
Rem  create a trigger on logon on database:
Rem ----------------------------------------------------


Rem  create trigger on after logon:
CREATE OR REPLACE TRIGGER pubsub.Systrig2
   AFTER LOGON
   ON DATABASE
   BEGIN
      New_enqueue('Pubsub.Logon', HEXTORAW('9999'), Dbms_standard.login_user);
   END;
```

```
/
```

- After subscriptions are created, the next step is for the client to register for notification using callback functions. This is done using the Oracle Call Interface (OCI). The following code performs necessary steps for registration. The initial steps of allocating and initializing session handles are omitted here for sake of clarity.

```
ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;

/* callback function for notification of logon of user 'scott' on database: */

ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *pay;
ub4 payl;
dvoid *desc;
ub4 mode;
{
    printf("Notification : User Scott Logged on\n");
}

int main()
{
    OCISession *authp = (OCISession *) 0;
    OCISubscription *subscrhpSnoop = (OCISubscription *)0;

    /*************************************************
       Initialize OCI Process/Environment
       Initialize Server Contexts
       Connect to Server
       Set Service Context
    *************************************************/

    /* Registration Code Begins */

    /* Each call to initSubscriptionHn allocates
           and Initialises a Registration Handle */


    initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle */
        "ADMIN:PUBSUB.SNOOP", /* subscription name */
                  /* <agent_name>:<queue_name> */
        (dvoid*)notifySnoop); /* callback function */
```

```
     /*****************************************************
       The Client Process does not need a live Session for Callbacks
       End Session and Detach from Server
     *****************************************************/

    OCISessionEnd ( svchp,  errhp, authp, (ub4) OCI_DEFAULT);

    /* detach from server */
    OCIServerDetach( srvhp, errhp, OCI_DEFAULT);

    while (1)     /* wait for callback */
        sleep(1);

}

void initSubscriptionHn (subscrhp,
subscriptionName,
func)

OCISubscription **subscrhp;
char* subscriptionName;
dvoid * func;
{

    /* allocate subscription handle: */

    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)subscrhp,
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

    /* set subscription name in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) subscriptionName,
        (ub4) strlen((char *)subscriptionName),
        (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

    /* set callback function in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) func, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
```

```
        (dvoid *) 0, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

    /* set namespace in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) &namespace, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

    checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,

        OCI_DEFAULT));
}
```

If user `SCOTT` logs on to the database, the client is notified, and the call back function `notifySnoop` is called.

Part IV

Developing Specialized Applications

This part deals with application development scenarios that not every developer faces. Oracle Database provides a range of features to help with each kind of application.

This part contains:

12

Using Regular Expressions With Oracle Database

This chapter introduces regular expression support for Oracle Database. This chapter covers the following topics:

- What are Regular Expressions?
- Oracle Database Regular Expression Support
- Oracle Database SQL Functions for Regular Expressions
- Metacharacters Supported in Regular Expressions
- Constructing Regular Expressions

See Also:

- *Oracle Database SQL Reference* for additional details on Oracle Database SQL functions for regular expressions
- *Oracle Database Globalization Support Guide* for details on using SQL regular expression functions in a multilingual environment
- *Mastering Regular Expressions* published by O'Reilly & Associates, Inc.

What are Regular Expressions?

Regular expressions specify patterns to search for in string data using standardized syntax conventions. A regular expression can specify complex patterns of character sequences. For example, the following regular expression:

```
a(b|c)d
```

searches for the pattern: 'a', followed by either 'b' or 'c', then followed by 'd'. This regular expression matches both 'abd' and 'acd'.

A regular expression is specified using two types of characters:

- Metacharacters—operators that specify algorithms for performing the search.
- Literals—the actual characters to search for.

Examples of regular expression syntax are given later in this chapter.

Oracle Database Regular Expression Support

Oracle Database implements regular expression support compliant with the POSIX Extended Regular Expression (ERE) specification.

Regular expression support is implemented with a set of Oracle Database SQL functions that allow you to search and manipulate string data. You can use these functions in any environment where Oracle Database SQL is used. See "Oracle Database SQL Functions for Regular Expressions" later in this chapter for more information.

Oracle Database supports a set of common metacharacters used in regular expressions. The behavior of supported metacharacters and related features is described in "Metacharacters Supported in Regular Expressions" on page 12-4.

> **Note:** The interpretation of metacharacters differs between tools that support regular expressions in the industry. If you are porting regular expressions from another environment to Oracle Database, ensure that the regular expression syntax is supported and the behavior is what you expect.

Oracle Database SQL Functions for Regular Expressions

The database provides a set of SQL functions that allow you to search and manipulate strings using regular expressions. You can use these functions on any

datatype that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2.

A regular expression must be enclosed or wrapped between single quotes. Doing so, ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

Table 12–1 gives a brief description of each regular expression function.

Note: As with all text literals used in SQL functions, regular expressions must be enclosed or wrapped between single quotes. If your regular expression includes the single quote character, enter two single quotation marks to represent one single quotation mark within your expression.

Table 12–1 SQL Regular Expression Functions

SQL Function	Description
REGEXP_LIKE	This function searches a character column for a pattern. Use this function in the WHERE clause of a query to return rows matching the regular expression you specify. See the *Oracle Database SQL Reference* for syntax details on the REGEXP_LIKE function.
REGEXP_REPLACE	This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern you specify. See the *Oracle Database SQL Reference* for syntax details on the REGEXP_REPLACE function.
REGEXP_INSTR	This function searches a string for a given occurrence of a regular expression pattern. You specify which occurrence you want to find and the start position to search from. This function returns an integer indicating the position in the string where the match is found. See the *Oracle Database SQL Reference* for syntax details on the REGEXP_INSTR function.
REGEXP_SUBSTR	This function returns the actual substring matching the regular expression pattern you specify. See the *Oracle Database SQL Reference* for syntax details on the REGEXP_SUBSTR function.

Metacharacters Supported in Regular Expressions

Table 12–2 lists the metacharacters supported for use in regular expressions passed to SQL regular expression functions. Details on the matching behavior of these metacharacters is given in "Constructing Regular Expressions" on page 12-5.

Table 12–2 Metacharacters Supported in Regular Expressions

Metacharacter Syntax	Operator Name	Description
`.`	Any Character — Dot	Matches any character
`+`	One or More — Plus Quantifier	Matches one or more occurrences of the preceding subexpression
`?`	Zero or One — Question Mark Quantifier	Matches zero or one occurrence of the preceding subexpression
`*`	Zero or More — Star Quantifier	Matches zero or more occurrences of the preceding subexpression
`{`*m*`}`	Interval—Exact Count	Matches exactly *m* occurrences of the preceding subexpression
`{`*m*`,}`	Interval—At Least Count	Matches at least *m* occurrences of the preceding subexpression
`{`*m*`,`*n*`}`	Interval—Between Count	Matches at least *m*, but not more than *n* occurrences of the preceding subexpression
`[ ... ]`	Matching Character List	Matches any character in list `...`
`[^ ... ]`	Non-Matching Character List	Matches any character not in list `...`
`|`	Or	`'a|b'` matches character 'a' or 'b'.
`( ... )`	Subexpression or Grouping	Treat expression `...` as a unit. The subexpression can be a string of literals or a complex expression containing operators.
`\`*n*	Backreference	Matches the n^{th} preceding subexpression, where *n* is an integer from 1 to 9.
`\`	Escape Character	Treat the subsequent metacharacter in the expression as a literal.
`^`	Beginning of Line Anchor	Match the subsequent expression only when it occurs at the beginning of a line.

Table 12–2 (Cont.) Metacharacters Supported in Regular Expressions

Metacharacter Syntax	Operator Name	Description
`$`	End of Line Anchor	Match the preceding expression only when it occurs at the end of a line.
`[:class:]`	POSIX Character Class	Match any character belonging to the specified character *`class`*. Can be used inside any list expression.
`[.element.]`	POSIX Collating Sequence	Specifies a collating sequence to use in the regular expression. The *`element`* you use must be a defined collating sequence, in the current locale.
`[=character=]`	POSIX Character Equivalence Class	Match characters having the same base character as the *`character`* you specify.

Constructing Regular Expressions

This section discusses construction of regular expressions.

Basic String Matching with Regular Expressions

The simplest match that you can perform with regular expressions is the basic string match. For this type of match, the regular expression is a string of literals with no metacharacters. For example, to find the sequence 'abc', you specify the regular expression:

```
abc
```

Regular Expression Operations on Subexpressions

As mentioned earlier, regular expressions are constructed using metacharacters and literals. Metacharacters that operate on a single literal, such as '+' and '?' can also operate on a sequence of literals or on a whole expression. To do so, you use the grouping operator to enclose the sequence or subexpression. See "Subexpression" on page 12-10 for more information on grouping.

Regular Expression Operator and Metacharacter Usage

This section gives usage examples for each supported metacharacter or regular expression operator.

Match Any Character—Dot

The *dot* operator `'.'` matches any single character in the current character set. For example, to find the sequence—'a', followed by any character, followed by 'c'—use the expression:

```
a.c
```

This expression matches all of the following sequences:

```
abc
adc
a1c
a&c
```

The expression does not match:

```
abb
```

One or More—Plus

The one or more operator `'+'` matches one or more occurrences of the preceding expression. For example, to find one or more occurrences of the character 'a', you use the regular expression:

```
a+
```

This expression matches all of the following:

```
a
aa
aaa
```

The expression does not match:

```
bbb
```

Zero or One—Question Mark Operator

The question mark matches zero or one—and only one—occurrence of the preceding character or subexpression. You can think of this operator as specifying an expression that is optional in the source text.

For example, to find—'a', optionally followed by 'b', then followed by 'c'—you use the following regular expression:

```
ab?c
```

This expression matches:

```
abc
ac
```

The expression does not match:

```
adc
abbc
```

Zero or More—Star

The zero or more operator `'*'`, matches zero or more occurrences of the preceding character or subexpression. For example, to find—'a', followed by zero or more occurrences of 'b', then followed by 'c'—use the regular expression:

```
ab*c
```

This expression matches all of the following sequences:

```
ac
abc
abbc
abbbbc
```

The expression does not match:

```
adc
```

Interval—Exact Count

The exact-count interval operator is specified with a single digit enclosed in braces. You use this operator to search for an exact number of occurrences of the preceding character or subexpression.

For example, to find where 'a' occurs exactly 5 times, you specify the regular expression:

```
a{5}
```

This expression matches:

```
aaaaa
```

The expression does not match:

```
aaaa
```

Interval—At Least Count

You use the at-least-count interval operator to search for a specified number of occurrences, or more, of the preceding character or subexpression. For example, to find where 'a' occurs at least 3 times, you use the regular expression:

```
a{3,}
```

This expression matches all of the following:

```
aaa
aaaaa
```

The expression does not match:

```
aa
```

Interval—Between Count

You use the between-count interval operator to search for a number of occurrences within a specified range. For example, to find where 'a' occurs at least 3 times and no more than 5 times, you use the following regular expression:

```
a{3,5}
```

This expression matches all of the following sequences:

```
aaa
aaaa
aaaaa
```

The expression does not match:

```
aa
```

Matching Character List

You use the matching character list to search for an occurrence of any character in a list. For example, to find either 'a', 'b', or 'c' use the following regular expression:

```
[abc]
```

This expression matches the first character in each of the following strings:

```
at
bet
cot
```

The expression does not match:

```
def
```

The following regular expression operators are allowed within the character list, any other metacharacters included in a character list lose their special meaning (are treated as literals):

- Range operator `'-'`
- POSIX character class `[: :]`
- POSIX collating sequence `[. .]`
- POSIX character equivalence class `[= =]`

Non-Matching Character List

Use the non-matching character list to specify characters that you do not want to match. Characters that are not in the non-matching character list are returned as a match. For example, to exclude the characters 'a', 'b', and 'c' from your search results, use the following regular expression:

```
[^abc]
```

This expression matches characters 'd' and 'g' in the following strings:

```
abcdef
ghi
```

The expression does not match:

```
abc
```

As with the matching character list, the following regular expression operators are allowed within the non-matching character list (any other metacharacters included in a character list are ignored):

- Range operator `'-'`
- POSIX character class `[: :]`
- POSIX collating sequence `[. .]`

- POSIX character equivalence class `[= =]`

For example, the following regular expression excludes any character between 'a' and 'i' from the search result:

```
[^a-i]
```

This expression matches the characters 'j' and 'l' in the following strings:

```
hijk
lmn
```

The expression does not match the characters:

```
abcdefghi
```

Or

Use the Or operator `'|'` to specify an alternate expression. For example to match 'a' or 'b', use the following regular expression:

```
a|b
```

Subexpression

You can use the subexpression operator to group characters that you want to find as a string or to create a complex expression. For example, to find the optional string 'abc', followed by 'def', use the following regular expression:

```
(abc)?def
```

This expression matches strings 'abcdef' and 'def' in the following strings:

```
abcdefghi
defghi
```

The expression does not match the string:

```
ghi
```

Backreference

The backreference lets you search for a repeated expression. You specify a backreference with `'\n'`, where *n* is an integer from 1 to 9 indicating the n^{th} preceding subexpression in your regular expression.

For example, to find a repeated occurrence of either string 'abc' or 'def', use the following regular expression:

```
(abc|def)\1
```

This expression matches the following strings:

```
abcabc
defdef
```

The expression does not match the following strings:

```
abcdef
abc
```

The backreference counts subexpressions from left to right starting with the opening parenthesis of each preceding subexpression.

The backreference lets you search for a repeated string without knowing the actual string ahead of time. For example, the regular expression:

```
^(.*)\1$
```

matches a line consisting of two adjacent appearances of the same string.

Escape Character

Use the escape character `'\'` to search for a character that is normally treated as a metacharacter. For example to search for the '+' character, use the following regular expression:

```
\+
```

This expression matches the plus character '+' in the following string:

```
abc+def
```

The expression does not match any characters in the string:

```
abcdef
```

Beginning of Line Anchor

Use the beginning of line anchor `^` to search for an expression that occurs only at the beginning of a line. For example, to find an occurrence of the string `def` at the beginning of a line, use the expression:

```
^def
```

This expression matches `def` in the string:

```
defghi
```

The expression does not match `def` in the following string:

```
abcdef
```

End of Line Anchor

The end of line anchor metacharacter `'$'` lets you search for an expression that occurs only at the end of a line. For example, to find an occurrence of `def` that occurs at the end of a line, use the following expression:

```
def$
```

This expression matches `def` in the string:

```
abcdef
```

The expression does not match `def` in the following string:

```
defghi
```

POSIX Character Class

The POSIX character class operator lets you search for an expression within a character list that is a member of a specific POSIX Character Class. You can use this operator to search for characters with specific formatting such as uppercase characters, or you can search for special characters such as digits or punctuation characters. The full set of POSIX character classes is supported.

To use this operator, specify the expression using the syntax `[:`*`class`*`:]` where *`class`* is the name of the POSIX character class to search for. For example, to search for one or more consecutive uppercase characters, use the following regular expression:

```
[[:upper:]]+
```

This expression matches 'DEF' in the string:

```
abcDEFghi
```

The expression does not return a match for the following string:

```
abcdefghi
```

Note that the character class must occur within a character list, so the character class is always nested within the brackets for the character list in the regular expression.

> **See Also:** *Mastering Regular Expressions* published by O'Reilly & Associates, Inc. for more information on POSIX character classes

POSIX Collating Sequence

The POSIX collating sequence element operator `[. .]` lets you use a collating sequence in your regular expression. The element you specify must be a defined collating sequence in the current locale.

This operator lets you use a multicharacter collating sequence in your regular expression where only one character would otherwise be allowed. For example, you can use this operator to ensure that the collating sequence 'ch', when defined in a locale such as Spanish, is treated as one character in operations that depend on the ordering of characters.

To use the collating sequence operator, specify `[.`*`element`*`.]` where *`element`* is the collating sequence you want to find. You can use any collating sequence that is defined in the current locale including single-character elements as well as multicharacter elements.

For example, to find the collating sequence 'ch', use the following regular expression:

```
[[.ch.]]
```

This expression matches the sequence 'ch' in the following string:

```
chabc
```

The expression does not match the following string:

```
cdefg
```

You can use the collating sequence operator in any regular expression where collation is needed. For example, to specify the range from 'a' to 'ch', you can use the following expression:

```
[a-[.ch.]]
```

POSIX Character Equivalence Class

Use the POSIX character equivalence class operator to search for characters in the current locale that are equivalent. For example, to find the Spanish character 'ñ' as well as 'n'.

To use this operator, specify `[=character=]`, to find all characters that are members of the same character equivalence class as the specified *`character`*.

For example, the following regular expression could be used to search for characters equivalent to 'n' in a Spanish locale:

```
[[=n=]]
```

This expression matches both 'N' and 'ñ' in the following string:

```
El Niño
```

Note:

- The character equivalence class must occur within a character list, so the character equivalence class is always nested within the brackets for the character list in the regular expression.
- Usage of character equivalents depends on how canonical rules are defined for your database locale. See the *Oracle Database Globalization Support Guide* for more information on linguistic sorting and string searching.

13

Developing Web Applications with PL/SQL

If you think that only new languages such as Java and JavaScript can do network operations and produce dynamic Web content, think again. PL/SQL has a number of features that you can use to Web-enable your database and make your back-office data interactive and accessible to intranet users or your customers.

This chapter discusses the following topics:

PL/SQL Web Applications

Web applications written in PL/SQL are typically sets of stored procedures that interact with Web browsers through the HTTP protocol. A set of interlinked dynamic HTML pages forms the user interface of a web application.

When a Web browser user visits a Web page, follows a hypertext link, or presses a `Submit` button on an HTML form, a URL is sent to the Web (HTTP) server, which causes the database server to run a stored procedure. Information the user provides in an HTML form is encoded in the URL. The URL also encodes information to identify which procedure, in which database, to call. The Web server passes this information along to the database as a set of parameters for the stored procedure.

The stored procedure that is invoked from a URL calls subprograms from the PL/SQL Web Toolkit. Typically, some of these subprograms, such as `Htp.Print`, prepare an HTML page that is displayed in the Web browser as a response to the user.

This process *dynamically* generates Web pages. Code running inside the database server produces HTML on the fly, so the generated Web page can vary depending on the database contents and the input parameters.

Dynamic generation of HTML is to be distinguished from *dynamic HTML* (DHTML). With DHTML, code in JavaScript or some other scripting language is downloaded to the browser along with HTML code, and this script code is *processed by the browser*.

DHTML can be coded by hand, so that it is static code, or it can itself be generated by program. That is, the generation of Web pages using a database can be combined with downloading JavaScript or other DHTML script code. A PL/SQL Web application can dynamically create complex DHTML that would be tedious to produce manually. A typical stored procedure might print some header information, issue a database query, and loop through the result set, formatting the data in an HTML list or table.

This program flow is very similar to a Perl script that operates on a text file. CGI scripts and programs running on a Web server are often used to dynamically produce Web pages, but they are usually not the best choice for interacting with a database. Using PL/SQL stored procedures has the advantage of providing all the power and flexibility of database processing: it supports DML statements, dynamic SQL, and cursors. It also eliminates the memory overhead of forking a new CGI process to treat each URL.

A Web browser-based application can be implemented entirely in PL/SQL using the following Oracle Database components:

- **PL/SQL Gateway** – a set of Web-server features that let you use PL/SQL stored procedures to process Web-browser (HTTP) requests and generate responses. A browser request in this context is a URL that represents a PL/SQL procedure call with actual parameter values. PL/SQL Gateway translates the URL, calls the stored procedure with the parameters, and returns output (typically HTML) from the procedure to the client Web browser.

  In Oracle Database 10*g*, **mod_plsql** is a *plug-in* to Oracle HTTP Server that implements PL/SQL Gateway (mod_plsql is also supplied as part of Oracle Application Server).

- **PL/SQL Web Toolkit** – a set of PL/SQL packages that application programmers can use to develop stored procedures that are called by PL/SQL Gateway at runtime. In response to a Web-browser request (URL), such a procedure accesses Oracle Database according to the browser-user input, updating or retrieving database information. It then generates an HTTP response to the browser, typically in the form of a file download or a web page (HTML) to be displayed. The Toolkit API provides ways for a stored procedure to obtain information about an HTTP request, generate HTTP headers such as content-type and mime-type, set browser cookies, and generate HTML pages.

PL/SQL Gateway

PL/SQL Gateway provides support for deploying PL/SQL-based database applications on the World-Wide Web. It is part of Oracle HTTP Server (OHS), which ships with Oracle Application Server and Oracle Database.

As part of the Oracle HTTP Server, it is the job of PL/SQL Gateway to *interpret* a URL sent by a Web browser to a Web server, *call* the appropriate PL/SQL subprograms to treat the browser request, then *return* the generated response to the browser. Typically, PL/SQL Gateway responds to a Web-browser HTTP request by constructing an HTML page to display. There are additional uses for the gateway, however. Here are two:

- Transfer files from a client machine to or from Oracle Database. You can upload and download text files or binary files. See "Uploading and Downloading Files With PL/SQL Gateway".
- Perform custom user authentication in Web applications. See "Custom Authentication With PL/SQL Gateway".

Configuring mod_plsql

As a plug-in to Oracle HTTP Server, mod_plsql causes stored procedures to be executed in response to HTTP requests.

For each URL that is processed, mod_plsql either uses a database session from its connection pool, or creates a new session on the fly and pools it. In order that mod_plsql can invoke the appropriate database PL/SQL procedure in a URL-processing session, you must first configure a virtual path and associate that path with a **Database Access Descriptor** (DAD).

A DAD is a named set of configuration values that specify the information necessary to create a session for a specific database and a specific database user/password. This includes the database service name and the Globalization Support setting (language, currency symbol, and so on) for the session.

See Also:

- *mod_plsql User's Guide*
- *Oracle HTTP Server Administrator's Guide* for information on mod_plsql configuration parameters
- "Using Caching with PL/SQL Web Applications" in *Oracle Application Server 10g Performance Guide* for information on caching dynamically generated HTML pages to improve performance

Uploading and Downloading Files With PL/SQL Gateway

You can use PL/SQL Gateway to transfer files from a client machine to or from Oracle Database. You can upload and download text files or binary files.

Uploading Files to the Database

To upload files, you must first define a document repository using the DAD configuration, and specify how to upload the content: as a `BLOB` or `LONG RAW` value. To initiate uploading, you define and submit a `multipart/form-data` form, following the RFC 1867 specification.

After you successfully upload a file, the procedure specified in the `ACTION` attribute of the `multipart/form-data` form is invoked. This invocation is similar to that of any regular PL/SQL Gateway procedure. Subsequently, you can download files that you have uploaded, delete uploaded files from the database, and read or write their attributes.

Downloading Files From the Database

You can download a file from the database in several alternative ways:

- Define a PL/SQL procedure that calls `wpg_docload.download_file(file_name)` to download file *file_name*.
- Define a virtual path for document downloads in the DAD configuration, and associate a user-defined procedure with that path. When PL/SQL Gateway detects the virtual path it automatically invokes the user-defined procedure, which in turn must call `wpg_docload.download_file(file_name)` to download file *file_name*.
- Use the *Direct BLOB Download* mechanism to download a `BLOB` from any database table. You do this by calling a PL/SQL procedure that streams the standard HTTP headers, such as `mime-type` and `content-length`, and then invokes `wpg_docload.download_file(blob_name)` to download `BLOB` *blob_name*.

See Also: RFC 1867, "Form-Based File Upload in HTML" (IETF)

Custom Authentication With PL/SQL Gateway

To authenticate individual Web-browser users for database purposes, it would be cumbersome to provide a different Database Access Descriptor (DAD) and URL for each user. Typically, a single URL, corresponding to a *single database user* (schema), is used for *all browser users*.

This means that authentication of browser users is typically not performed by the database, but by the Web *application*. It is the application that determines if a given Web-browser user should have access to the database. It can also determine which database schema (hence which URL) to use for a given user. You use the `OWA_CUSTOM` package of the PL/SQL Web Toolkit to perform such custom user authentication in Web applications.

For example, suppose you have a purchasing application that is accessed by multiple third-party vendors. Instead of creating a separate schema for each vendor, load the application into a common schema where all users log in. Using `OWA_CUSTOM`, your application can authenticate each user, in any way it needs to. With `OWA_CUSTOM`, authentication is done *only* by the application, not by the database.

Custom authentication cannot be combined with dynamic username/password authentication; it needs to have a static username/password stored in the DAD configuration file. PL/SQL Gateway uses this DAD username/password to log in to the database.

Once mod_plsql is logged in, authentication control is passed back to the application, by calling an application-level PL/SQL hook. This callback function is implemented by the application developers. The value returned by the callback function determines whether authentication succeeds or fails: `TRUE` means success; `FALSE` means failure.

Depending on what kind of custom authentication is desired, you can place the authentication function in different locations:

- **Global OWA** invokes the same authentication function for all users and all procedures.
- **Custom OWA** invokes a different authentication function for each user and for each procedure.
- **Individual-Package OWA** invokes the same authentication function for all users, but only for anonymous procedures or procedures in a specific package.

  For example, using Custom OWA, an authorization function might verify that a user has logged in as user `guest` with password `welcome`, or it might check the user's IP address to determine access.

See Also: *mod_plsql User's Guide*

PL/SQL Web Toolkit

To develop the stored procedures that are executed by PL/SQL Gateway at runtime, you use PL/SQL Web Toolkit: a set of PL/SQL packages that can be used to obtain information about an HTTP request; specify HTTP response headers, such as `cookies`, `content-type`, and `mime-type`, for HTTP headers; set cookies; and generate standard HTML tags for creating HTML pages.

Commonly used PL/SQL Toolkit packages are listed in Table 13–1.

Table 13–1 Some Packages in PL/SQL Toolkit

Package	Description
`htp`	`htp` (Hypertext Procedures) package – Procedures that generate HTML tags. For instance, the procedure `htp.anchor` generates the HTML anchor tag, `<A>`.
`htf`	`htf` (Hypertext Functions) package– Function versions of the procedures in the `htp` package. The function versions do not directly generate output in a Web page. Instead, they pass their output as return values to the statements that invoke them. Use these functions when you need to nest function calls.

Table 13–1 (Cont.) Some Packages in PL/SQL Toolkit

Package	Description
owa_cache	Functions and procedures that enable the PL/SQL Gateway *cache* feature, to improve the performance of your PL/SQL Web application. You can use this package to enable expires-based and validation-based caching using the PL/SQL Gateway file system.
owa_cookie	Subprograms that send and retrieve HTTP cookies to and from a client Web browser. Cookies are strings a browser uses to maintain state between HTTP calls. State can be maintained throughout a client session or longer if a cookie expiration date is included.
owa_custom	The authorize function used by cookies. See "Uploading and Downloading Files With PL/SQL Gateway".
owa_image	Subprograms that obtain the coordinates where a user clicked an image. Use this package when you have an image map whose destination links invoke a PL/SQL Gateway.
owa_opt_lock	Subprograms that impose database optimistic locking strategies, to prevent lost updates. Lost updates can otherwise occur if a user selects, and then attempts to update, a row whose values have been changed in the meantime by another user.
owa_pattern	Subprograms that perform string matching and string manipulation with regular expression functionality.
owa_sec	Subprograms used by the PL/SQL Gateway for authenticating requests.
owa_text	Subprograms used by package owa_pattern for manipulating strings. You can also use them directly.
owa_util	Utility subprograms: ■ Dynamic SQL utilities to produce pages with dynamically generated SQL code. ■ HTML utilities to retrieve the values of CGI environment variables and perform URL redirects. ■ Date utilities for correct date-handling. Date values are simple strings in HTML, but must be properly treated as an Oracle Database datatype.

Table 13–1 (Cont.) Some Packages in PL/SQL Toolkit

Package	Description
wpg_docload	Subprograms that download documents from a document repository that you define using the DAD configuration. See "Uploading and Downloading Files With PL/SQL Gateway".

See Also: *PL/SQL Packages and Types Reference*

Generating HTML Output from PL/SQL

Traditionally, PL/SQL Web applications have used function calls to generate each HTML tag for output, using the PL/SQL Web toolkit packages that come with Oracle Database:

```
owa_util.mime_header('text/html');

htp.htmlOpen;
htp.headOpen;
htp.title('Title of the HTML File');
htp.headClose;

htp.bodyOpen( cattributes => 'TEXT="#000000" BGCOLOR="#FFFFFF"');
htp.header(1, 'Heading in the HTML File');
htp.para;
htp.print('Some text in the HTML file.');
htp.bodyClose;

htp.htmlClose;
```

You can learn the API calls corresponding to each tag, or just use some of the basic ones like `HTP.PRINT` to print the text and tags together:

```
htp.print('<html>');
htp.print('<head>');
htp.print('<meta http-equiv="Content-Type" content="text/html">');
htp.print('<title>Title of the HTML File</title>');
htp.print('</head>');

htp.print('<body TEXT="#000000" BGCOLOR="#FFFFFF">');
htp.print('<h1>Heading in the HTML File</h1>');
htp.print('<p>Some text in the HTML file.');
htp.print('</body>');
```

```
htp.print('</html>');
```

This chapter introduces an additional method, PL/SQL server pages, that lets you build on your knowledge of HTML tags, rather than learning a new set of function calls.

In an application written as a set of PL/SQL server pages, you can still use functions from the PL/SQL Web toolkit to simplify the processing involved in displaying tables, storing persistent data (cookies), and working with CGI protocol internals.

Passing Parameters to a PL/SQL Web Application

To be useful in a wide variety of situations, a Web application must be interactive enough to allow user choices. To keep the attention of impatient Web surfers, you should streamline the interaction so that users can specify these choices very simply, without a lot of decision-making or data entry.

The main methods of passing parameters to PL/SQL Web applications are:

- Using HTML form tags. The user fills in a form on one Web page, and all the data and choices are transmitted to a stored procedure when the user clicks the `Submit` button on the page.
- Hard-coded in the URL. The user clicks on a link, and a set of predefined parameters are transmitted to a stored procedure. Typically, you would include separate links on your Web page for all the choices that the user might want.

Passing List and Dropdown List Parameters from an HTML Form

List boxes and dropdown lists are implemented using the same HTML tag (`<SELECT>`).

Use a list box for a large number of choices, where the user might have to scroll to see them all, or to allow multiple selections. List boxes are good for showing items in alphabetical order, so that users can find an item quickly without reading all the choices.

Use a dropdown list for a small number of choices, or where screen space is limited, or for choices in an unusual order. The dropdown captures the first-time user's attention and makes them read the items. If you keep the choices and order consistent, users can memorize the motion of selecting an item from the dropdown list, allowing them to make selections quickly as they gain experience.

Passing Radio Button and Checkbox Parameters from an HTML Form

Radio buttons pass either a null value (if none of the radio buttons in a group is checked), or the value specified on the radio button that is checked.

To specify a default value for a set of radio buttons, you can include the `CHECKED` attribute in one of the `INPUT` tags, or include a `DEFAULT` clause on the parameter within the stored procedure. When setting up a group of radio buttons, be sure to include a choice that indicates "no preference", because once the user selects a radio button, they can still select a different one, but they cannot clear the selection completely. For example, include a "Don't Care" or "Don't Know" selection along with "Yes" and "No" choices, in case someone makes a selection and then realizes it was wrong.

Checkboxes need special handling, because your stored procedure might receive a null value, a single value, or multiple values:

All the checkboxes with the same `NAME` attribute make up a checkbox group. If none of the checkboxes in a group is checked, the stored procedure receives a null value for the corresponding parameter.

If one checkbox in a group is checked, the stored procedure receives a single `VARCHAR2` parameter.

If more than one checkbox in a group is checked, the stored procedure receives a parameter with the PL/SQL type `TABLE OF VARCHAR2`. You must declare a type like this, or use a predefined one like `OWA_UTIL.IDENT_ARR`. To retrieve the values, use a loop:

```
CREATE OR REPLACE PROCEDURE handle_checkboxes ( checkboxes owa_util.ident_arr )
AS
BEGIN
  ...
  FOR i IN 1..checkboxes.count
  LOOP
    htp.print('<p>Checkbox value: ' || checkboxes(i));
  END LOOP;
  ...
END;
/
show errors;
```

Passing Entry Field Parameters from an HTML Form

Entry fields require the most validation, because a user might enter data in the wrong format, out of range, and so on. If possible, validate the data on the client

side using dynamic HTML or Java, and format it correctly for the user or prompt them to enter it again.

For example:

- You might prevent the user from entering alphabetic characters in a numeric entry field, or from entering characters once a length limit is reached.
- You might silently remove spaces and dashes from a credit card number if the stored procedure expects the value in that format.
- You might inform the user immediately when they type a number that is too large, so that they can retype it.

Because you cannot always rely on such validation to succeed, code the stored procedures to deal with these cases anyway. Rather than forcing the user to use the `Back` button when they enter wrong data, display a single page with an error message and the original form with all the other values filled in.

For sensitive information such as passwords, a special form of the entry field, `<INPUT TYPE=PASSWORD>`, hides the text as it is typed in.

For example, the following procedure accepts two strings as input. The first time it is called, the user sees a simple form prompting for the input values. When the user submits the information, the same procedure is called again to check if the input is correct. If the input is OK, the procedure processes it. If not, the procedure prompts for new input, filling in the original values for the user.

```
-- Store a name and associated zip code in the database.
CREATE OR REPLACE PROCEDURE associate_name_with_zipcode
(
  name VARCHAR2 DEFAULT NULL,
  zip VARCHAR2 DEFAULT NULL
)
AS
  booktitle VARCHAR2(256);
BEGIN
-- Both entry fields must contain a value. The zip code must be 6 characters.
-- (In a real program you would perform more extensive checking.)
  IF name IS NOT NULL AND zip IS NOT NULL AND length(zip) = 6 THEN
    store_name_and_zipcode(name, zip);
    htp.print('<p>The person ' || name || ' has the zip code ' || zip || '.');
-- If the input was OK, we stop here and the user does not see the form again.
    RETURN;
  END IF;

-- If some data was entered, but it is not correct, show the error message.
```

```
  IF (name IS NULL AND zip IS NOT NULL)
    OR (name IS NOT NULL AND zip IS NULL)
    OR (zip IS NOT NULL AND length(zip) != 6)
  THEN
    htp.print('<p><b>Please re-enter the data. Fill in all fields, and use a
6-digit zip code.</b>');
  END IF;

-- If the user has not entered any data, or entered bad data, prompt for
-- input values.

-- Make the form call the same procedure to check the input values.
  htp.formOpen( 'scott.associate_name_with_zipcode', 'GET');
  htp.print('<p>Enter your name:</td>');
  htp.print('<td valign=center><input type=text name=name value="' || name ||
'">');
  htp.print('<p>Enter your zip code:</td>');
  htp.print('<td valign=center><input type=text name=zip value="' || zip ||
'">');
  htp.formSubmit(NULL, 'Submit');
  htp.formClose;
END;
/
show errors;
```

Passing Hidden Parameters from an HTML Form

One technique for passing information through a sequence of stored procedures, without requiring the user to specify the same choices each time, is to include hidden parameters in the form that calls a stored procedure. The first stored procedure places information, such as a user name, into the HTML form that it generates. The value of the hidden parameter is passed to the next stored procedure, as if the user had entered it through a radio button or entry field.

Other techniques for passing information from one stored procedure to another include:

- Sending a "cookie" containing the persistent information to the browser. The browser then sends this same information back to the server when accessing other Web pages from the same site. Cookies are set and retrieved through the HTTP headers that are transferred between the browser and the Web server before the HTML text of each Web page.

- Storing the information in the database itself, where later stored procedures can retrieve it. This technique involves some extra overhead on the database server, and you must still find a way to keep track of each user as multiple users access the server at the same time.

Uploading a File from an HTML Form

You can use an HTML form to choose a file on a client system, and transfer it to the server. A stored procedure can insert the file into the database as a `CLOB`, `BLOB`, or other type that can hold large amounts of data.

The PL/SQL Web toolkit and the PL/SQL Gateways like mod_plsql have the notion of a "document table" that holds uploaded files.

See Also: *mod_plsql User's Guide*

Submitting a Completed HTML Form

By default, an HTML form must have a `Submit` button, which transmits the data from the form to a stored procedure or CGI program. You can label this button with text of your choice, such as "Search", "Register", and so on.

You can have multiple forms on the same page, each with its own form elements and `Submit` button. You can even have forms consisting entirely of hidden parameters, where the user makes no choice other than clicking the button.

Using JavaScript or other scripting languages, you can do away with the Submit button and have the form submitted in response to some other action, such as selecting from a dropdown list. This technique is best when the user only makes a single selection, and the confirmation step of the `Submit` button is not essential.

Handling Missing Input from an HTML Form

When an HTML form is submitted, your stored procedure receives null parameters for any form elements that are not filled in. For example, null parameters can result from an empty entry field, a set of checkboxes, radio buttons, or list items with none checked, or a `VALUE` parameter of "" (empty quotation marks).

Regardless of any validation you do on the client side, always code stored procedures to handle the possibility that some parameters are null:

- Use a `DEFAULT` clause in all parameter declarations, to prevent an exception when the stored procedure is called with a missing form parameter. You can set the default to zero for numeric values (when that makes sense), and use

DEFAULT NULL when you want to check whether or not the user actually specifies a value.

- Before using an input parameter value that has a DEFAULT NULL declaration, check if it is null.
- Make the procedure generate sensible results even when not all input parameters are specified. You might leave some sections out of a report, or display a text string or image in a report to indicate where parameters were not specified.
- Provide a way to fill in the missing values and run the stored procedure again, directly from the results page. For example, you could include a link that calls the same stored procedure with an additional parameter, or display the original form with its values filled in as part of the output.

Maintaining State Information Between Web Pages

Web applications are particularly concerned with the idea of **state**, the set of data that is current at a particular moment in time. It is easy to lose state information when switching from one Web page to another, which might result in asking the user to make the same choices over and over.

You can pass state information between dynamic Web pages using HTML forms. The information is passed as a set of name-value pairs, which are turned into stored procedure parameters for you.

If the user has to make multiple selections, or one selection from many choices, or it is important to avoid an accidental selection, use an HTML form. After the user makes and reviews all the choices, they confirm the choices with the Submit button. Subsequent pages can use forms with hidden parameters (<INPUT TYPE=HIDDEN> tags) to pass these choices from one page to the next.

If the user is only considering one or two choices, or the decision points are scattered throughout the Web page, you can save the user from hunting around for the Submit button by representing actions as hyperlinks and including any necessary name-value pairs in the query string (the part following the ? within a URL).

An alternative way to main state information is to use Oracle Application Server and its mod_ose module. This approach lets you store state information in package variables that remain available as a user moves around a Web site.

> **See Also:** the Oracle Application Server documentation set at http://otn.oracle.com/documentation/

Performing Network Operations within PL/SQL Stored Procedures

While built-in PL/SQL features are focused on traditional database operations and programming logic, Oracle Database provides packages that open up Internet computing to PL/SQL programmers.

Sending E-Mail from PL/SQL

You can send e-mail from a PL/SQL program or stored procedure using the `UTL_SMTP` package. You can find details about this package in the *PL/SQL Packages and Types Reference*.

The following code example illustrates how the SMTP package might be used by an application to send email. The application connects to an SMTP server at port 25 and sends a simple text message.

```
PROCEDURE send_test_message
IS
    mailhost    VARCHAR2(64) := 'mailhost.fictional-domain.com';
    sender      VARCHAR2(64) := 'me@fictional-domain.com';
    recipient   VARCHAR2(64) := 'you@fictional-domain.com';
    mail_conn  utl_smtp.connection;
BEGIN
    mail_conn := utl_smtp.open_connection(mailhost, 25);
    utl_smtp.helo(mail_conn, mailhost);
    utl_smtp.mail(mail_conn, sender);
    utl_smtp.rcpt(mail_conn, recipient);
-- If we had the message in a single string, we could collapse
-- open_data(), write_data(), and close_data() into a single call to data().
    utl_smtp.open_data(mail_conn);
    utl_smtp.write_data(mail_conn, 'This is a test message.' || chr(13));
    utl_smtp.write_data(mail_conn, 'This is line 2.' || chr(13));
    utl_smtp.close_data(mail_conn);
    utl_smtp.quit(mail_conn);
    EXCEPTION
        WHEN OTHERS THEN
           -- Insert error-handling code here
           NULL;
END;
```

Getting a Host Name or Address from PL/SQL

You can determine the hostname of the local machine, or the IP address of a given hostname from a PL/SQL program or stored procedure using the UTL_INADDR package. You can find details about this package in the *PL/SQL Packages and Types Reference*. You use the results in calls to the UTL_TCP package.

Working with TCP/IP Connections from PL/SQL

You can open TCP/IP connections to machines on the network, and read or write to the corresponding sockets, using the UTL_TCP package. You can find details about this package in the *PL/SQL Packages and Types Reference*.

Retrieving the Contents of an HTTP URL from PL/SQL

You can retrieve the contents of an HTTP URL using the UTL_HTTP package. The contents are typically in the form of HTML-tagged text, but may be plain text, a JPEG image, or any sort of file that is downloadable from a Web server. You can find details about this package in the *PL/SQL Packages and Types Reference*.

The UTL_HTTP package lets you:

- Control the details of the HTTP session, including header lines, cookies, redirects, proxy servers, IDs and passwords for protected sites, and CGI parameters through the GET or POST methods.
- Speed up multiple accesses to the same Web site using HTTP 1.1 persistent connections.
- Construct and interpret URLs for use with UTL_HTTP through the ESCAPE and UNESCAPE functions in the UTL_URL package.

Typically, developers have used Java or Perl to perform these operations; this package lets you do them with PL/SQL.

```
CREATE OR REPLACE PROCEDURE show_url
(
    url      IN VARCHAR2,
    username IN VARCHAR2 DEFAULT NULL,
    password IN VARCHAR2 DEFAULT NULL
) AS
    req       utl_http.req;
    resp      utl_http.resp;
    name      VARCHAR2(256);
    value     VARCHAR2(1024);
    data      VARCHAR2(255);
```

```
    my_scheme VARCHAR2(256);
    my_realm  VARCHAR2(256);
    my_proxy  BOOLEAN;
BEGIN
-- When going through a firewall, pass requests through this host.
-- Specify sites inside the firewall that don't need the proxy host.
  utl_http.set_proxy('proxy.my-company.com', 'corp.my-company.com');

-- Ask UTL_HTTP not to raise an exception for 4xx and 5xx status codes,
-- rather than just returning the text of the error page.
  utl_http.set_response_error_check(FALSE);

-- Begin retrieving this Web page.
  req := utl_http.begin_request(url);

-- Identify ourselves. Some sites serve special pages for particular browsers.
  utl_http.set_header(req, 'User-Agent', 'Mozilla/4.0');

-- Specify a user ID and password for pages that require them.
  IF (username IS NOT NULL) THEN
    utl_http.set_authentication(req, username, password);
  END IF;

  BEGIN
-- Start receiving the HTML text.
    resp := utl_http.get_response(req);

-- Show the status codes and reason phrase of the response.
    dbms_output.put_line('HTTP response status code: ' || resp.status_code);
    dbms_output.put_line('HTTP response reason phrase: ' || resp.reason_phrase);

-- Look for client-side error and report it.
    IF (resp.status_code >= 400) AND (resp.status_code <= 499) THEN

-- Detect whether the page is password protected, and we didn't supply
-- the right authorization.
      IF (resp.status_code = utl_http.HTTP_UNAUTHORIZED) THEN
        utl_http.get_authentication(resp, my_scheme, my_realm, my_proxy);
        IF (my_proxy) THEN
          dbms_output.put_line('Web proxy server is protected.');
          dbms_output.put('Please supply the required ' || my_scheme ||
            ' authentication username/password for realm ' || my_realm ||
            ' for the proxy server.');
        ELSE
          dbms_output.put_line('Web page ' || url || ' is protected.');
```

```
          dbms_output.put('Please supplied the required ' || my_scheme ||
            ' authentication username/password for realm ' || my_realm ||
            ' for the Web page.');
        END IF;
      ELSE
        dbms_output.put_line('Check the URL.');
      END IF;

      utl_http.end_response(resp);
      RETURN;

-- Look for server-side error and report it.
    ELSIF (resp.status_code >= 500) AND (resp.status_code <= 599) THEN

      dbms_output.put_line('Check if the Web site is up.');
      utl_http.end_response(resp);
      RETURN;

    END IF;

-- The HTTP header lines contain information about cookies, character sets,
-- and other data that client and server can use to customize each session.
    FOR i IN 1..utl_http.get_header_count(resp) LOOP
      utl_http.get_header(resp, i, name, value);
      dbms_output.put_line(name || ': ' || value);
    END LOOP;

-- Keep reading lines until no more are left and an exception is raised.
    LOOP
      utl_http.read_line(resp, value);
      dbms_output.put_line(value);
    END LOOP;
  EXCEPTION
    WHEN utl_http.end_of_body THEN
    utl_http.end_response(resp);
  END;

END;
/
SET serveroutput ON
-- The following URLs illustrate the use of this procedure,
-- but these pages do not actually exist. To test, substitute
-- URLs from your own Web server.
exec show_url('http://www.oracle.com/no-such-page.html')
exec show_url('http://www.oracle.com/protected-page.html')
```

```
exec show_url('http://www.oracle.com/protected-page.html', 'scott', 'tiger')
```

Working with Tables, Image Maps, Cookies, and CGI Variables from PL/SQL

Packages for all of these functions are supplied with Oracle8*i* and higher. You use these packages in combination with the mod_plsql plug-in of Oracle HTTP Server (OHS). You can format the results of a query in an HTML table, produce an image map, set and get HTTP cookies, check the values of CGI variables, and combine other typical Web operations with a PL/SQL program.

Documentation for these packages is not part of the database documentation library. The location of the documentation depends on the particular application server you are running. To get started with these packages, look at their procedure names and parameters using the SQL*Plus `DESCRIBE` command:

```
DESCRIBE HTP;
DESCRIBE HTF;
DESCRIBE OWA_UTIL;
```

Embedding PL/SQL Code in Web Pages (PL/SQL Server Pages)

To include dynamic content, including the results of SQL queries, inside Web pages, you can use server-side scripting through *PL/SQL Server Pages* (PSP). You can author the Web pages in a script-friendly HTML authoring tool, and drop the pieces of PL/SQL code into place. For cutting-edge Web pages, you might find this technique more convenient than using the HTP and HTF packages to write out HTML content line by line.

Because the processing is done on the server -- in this case, the database server rather than the Web server -- the browser receives a plain HTML page with no special script tags, and you can support all browsers and browser levels equally. It also makes network traffic efficient by minimizing the number of server round-trips.

Embedding the PL/SQL code in the HTML page that you create lets you write content quickly and follow a rapid, iterative development process. You maintain central control of the software, with only a Web browser required on the client machine.

The steps to implement a Web-based solution using PL/SQL server pages are:

- Choosing a Software Configuration

- Writing the Code and Content for the PL/SQL Server Page
- Loading the PL/SQL Server Page into the Database as a Stored Procedure

Choosing a Software Configuration

To develop and deploy PL/SQL Server Pages, you need Oracle Database version 8.1.6 or later, together with the mod_plsql plug-in of Oracle HTTP Server (OHS).

Choosing Between PSP and the PL/SQL Web Toolkit

You can produce the same results in different ways:

- By writing an HTML page with embedded PL/SQL code and compiling it as a PL/SQL server page. You may call procedures from the PL/SQL Web Toolkit, but not to generate every line of HTML output.
- By writing a complete stored procedure that produces HTML by calling the `HTP` and `OWA_*` packages in the PL/SQL Web Toolkit.

The key factors in choosing between these techniques are:

- What source are you using as a starting point?
  - If you have a large body of HTML, and want to include dynamic content or make it the front end of a database application, use PSP.
  - If you have a large body of PL/SQL code that produces formatted output, you may find it more convenient to produce HTML tags by changing your print statements to call the HTP package of the PL/SQL Web Toolkit.
- What is the fastest and most convenient authoring environment for your group?
  - If most work is done using HTML authoring tools, use PSP.
  - If you use authoring tools that produce PL/SQL code, such as the page-building wizards in Oracle Application Server Portal, then it might be less convenient to use PSP.

How PSP Relates to Other Scripting Solutions

Because any kind of tags can be passed unchanged to the browser through a PL/SQL server page, you can include JavaScript or other client-side script code in a PL/SQL server page.

You cannot mix PL/SQL server pages with other server-side script features, such as server-side includes. In many cases, you can get the same results by using the corresponding PSP features.

PSP uses the same script tag syntax as Java Server Pages (JSP), to make it easy to switch back and forth.

PSP uses syntax similar to that of Active Server Pages (ASP), although the syntax is not identical and you must typically translate from VBScript or JScript to PL/SQL. The best candidates for migration are pages that use the Active Data Object (ADO) interface to do database operations.

Writing the Code and Content for the PL/SQL Server Page

You can start with an existing Web page, or with an existing stored procedure. Either way, with a few additions and changes you can create dynamic Web pages that perform database operations and display the results.

The Format of the PSP File

The file for a PL/SQL server page must have the extension `.psp`.

It can contain whatever content you like, with text and tags interspersed with PSP directives, declarations, and scriptlets:

- In the simplest case, it is nothing more than an HTML file. Compiling it as a PL/SQL server page produces a stored procedure that outputs the exact same HTML file.
- In the most complex case, it is a PL/SQL procedure that generates all the content of the Web page, including the tags for title, body, and headings.
- In the typical case, it is a mix of HTML (providing the static parts of the page) and PL/SQL (filling in the dynamic content).

The order and placement of the PSP directives and declarations is not significant in most cases -- only when another file is being included. For ease of maintenance, we recommend placing the directives and declarations together near the beginning of the file.

The following sections discuss the way to produce various results using the PSP scripting elements. If you are familiar with dynamic HTML and want to start coding right away, you can jump forward to Syntax of PL/SQL Server Page Elements on page 13-27 and "Examples of PL/SQL Server Pages" on page 13-31.

Specifying the Scripting Language

To identify a file as a PL/SQL Server Page, include a `<%@ page language="PL/SQL" %>` directive somewhere in the file. This directive is for compatibility with other scripting environments.

Accepting User Input

User input comes encoded in the URL that retrieves the HTML page. You can generate the URL by hard-coding it in an HTML link, or by calling your page as the "action" of an HTML form. Your page receives the input as parameters to a PL/SQL stored procedure.

To set up parameter passing for a PL/SQL server page, include a `<%@ plsql parameter="`*`varname`*`" %>` directive. By default, parameters are of type `VARCHAR2`. To use a different type, include a `type="`*`typename`*`"` attribute within the directive. To set a default value, so that the parameter becomes optional, include a `default="`*`expression`*`"` attribute in the directive. The values for this attribute are substituted directly into a PL/SQL statement, so any strings must be single-quoted, and you can use special values such as `null`.

Displaying HTML

The PL/SQL parts of the page are enclosed within special delimiters. All other content is passed along verbatim -- including any whitespace -- to the browser. To display text or HTML tags, write it as you would a typical Web page. You do not need to call any output function.

Sometimes you might want to display one line of output or another, or change the value of an attribute, based on some condition. You can include `IF/THEN` logic and variable substitution inside the PSP delimiters, as shown in subsequent sections.

Returning XML, Text, or Other Document Types

By default, the PL/SQL gateway transmits files as HTML documents, so that the browser formats them according to the HTML tags. If you want the browser to interpret the document as XML, plain text (with no formatting), or some other document type, include a `<%@ page contentType="`*`MIMEtype`*`" %>` directive. (The attribute name is case-sensitive, so be sure to capitalize it as `contentType`.) Specify `text/html`, `text/xml`, `text/plain`, `image/jpeg`, or some other MIME type that the browser or other client program recognizes. Users may have to configure their browsers to recognize some MIME types.

Typically, a PL/SQL server page is intended to be displayed in a Web browser. It could also be retrieved and interpreted by a program that can make HTTP requests, such as a Java or Perl application.

Returning Pages Containing Different Character Sets

By default, the PL/SQL gateway transmits files using the character set defined by the Web gateway. To convert the data to a different character set for displaying in a

browser, include a <%@ page charset="*encoding*" %> directive. Specify Shift_JIS, Big5, UTF-8, or another encoding that the browser or other client program recognizes.

You must also configure the character set setting in the database accessor descriptor (DAD) of the Web gateway. Users may have to select the same encoding in their browsers to see the data displayed properly.

For example, a database in Japan might have a database character set that uses the EUC encoding, while the Web browsers are set up to display Shift_JIS encoding.

Handling Script Errors

Any errors in HTML tagging are handled by the browser. The PSP loading process does not check for them.

If you make a syntax error in the PL/SQL code, the loader stops and you must fix the error before continuing. Note that any previous version of the stored procedure can be erased when you attempt to replace it and the script contains a syntax error. You might want to use one database for prototyping and debugging, then load the final stored procedure into a different database for production. You can switch databases using a command-line flag, without changing any source code.

To handle database errors that occur when the script runs, you can include PL/SQL exception-handling code within a PSP file, and have any unhandled exceptions bring up a special page. The page for unhandled exceptions is another PL/SQL server page with extension .psp. The error procedure does not receive any parameters, so to determine the cause of the error, it can call the SQLCODE and SQLERRM functions.

You can also display a standard HTML page without any scripting when an error occurs, but you must still give it the extension .psp and load it into the database as a stored procedure.

Naming the PL/SQL Stored Procedure in a PSP Script

Each top-level PL/SQL server page corresponds to a stored procedure within the server. By default, the procedure is given the same name as the original file, with the .psp extension removed. To name the procedure something else, include a include a <%@ plsql procedure="*procname*" %> directive.

Including the Contents of Other Files in a PSP Script

You can set up an include mechanism to pull in the contents of other files, typically containing either static HTML content or more PL/SQL scripting code. Include a

<%@ include file="*filename*" %> directive at the point where the other file's content should appear. Because the files are processed at the point where you load the stored procedure into the database, the substitution is done only once, not whenever the page is served.

You can use any names and extensions for the included files. If the included files contain PL/SQL scripting code, they do not need their own set of directives to identify the procedure name, character set, and so on.

When specifying the names of files to the PSP loader, you must include the names of all included files also. Specify the names of included files before the names of any `.psp` files.

You can use this feature to pull in the same content, such as a navigation banner, into many different files. Or, you can use it as a macro capability to include the same section of script code in more than one place in a page.

Declaring Variables in a PSP Script

If you need to use global variables within the script, you can include a declaration block inside the delimiters <%! %>. All the usual PL/SQL syntax is allowed within the block. The delimiters server as shorthand, letting you omit the `DECLARE` keyword. All the declarations are available to the code later on in the file.

You can specify multiple declaration blocks; internally, they are all merged into a single block when the PSP file is made into a stored procedure.

You can also use explicit `DECLARE` blocks within the <% %> delimiters that are explained later. These declarations are only visible to the following `BEGIN/END` block.

Specifying Executable Statements in a PSP Script

You can include any PL/SQL statements within the delimiters <% %>. The statements can be complete, or clauses of a compound statement, such as the `IF` part of an `IF-THEN-ELSE` statement. Any variables declared within `DECLARE` blocks are only visible to the following `BEGIN/END` block.

Substituting an Expression Result in a PSP Script

To include a value that depends upon the result of a PL/SQL expression, include the expression within the delimiters <%= %>. Because the result is always substituted in the middle of text or tags, it must be a string value or be able to be cast to a string. For any types that cannot be implicitly cast, such as `DATE`, pass the value to the PL/SQL `TO_CHAR` function.

The content between the <%= %> delimiters is processed by the HTP.PRN function, which trims any leading or trailing whitespace and requires that you quote any literal strings.

Conventions for Quoting and Escaping Strings in a PSP Script

When values specified in PSP attributes are used for PL/SQL operations, they are passed exactly as you specify them in the PSP file. If PL/SQL requires a single-quoted string, you must specify the string with the single quotes around it -- and surround the whole thing with double quotes.

You can also nest single-quoted strings inside single quotes. In this case, you must *escape* the nested single quotes by specifying the sequence \'.

Most characters and character sequences can be included in a PSP file without being changed by the PSP loader. To include the sequence %>, specify the escape sequence %\>. To include the sequence <%, specify the escape sequence <\%.

Including Comments in a PSP Script

To put a comment in the HTML portion of a PL/SQL server page, for the benefit of people reading the PSP source code, use the syntax:

```
<%-- Comment text --%>
```

These comments do not appear in the HTML output from the PSP.

To create a comment that is visible in the HTML output, place the comment in the HTML portion and use the regular HTML comment syntax:

```
<!-- Comment text -->
```

To include a comment inside a PL/SQL block within a PSP, you can use the normal PL/SQL comment syntax.

For example, here is part of a PSP file showing several kinds of comments:

```
<p>Today we introduce our new model XP-10.
<%--
This is the project with code name "Secret Project".
People viewing the HTML page will not see this comment.
--%>
<!--
Some pictures of the XP-10.
People viewing the HTML page will see this comment.
-->
<%
```

```
for image_file in (select pathname, width, height, description
  from image_library where model_num = 'XP-10')
-- Comments interspersed with PL/SQL statements.
-- People viewing the HTML page will not see this comment.
loop
%>
<img src="<%= image_file.pathname %>" width=<% image_file.width %>
height=<% image_file.height %> alt="<% image_file.description %>">
<br>
<%
end loop;
%>
```

Retrieving a Result Set from a Query in a PSP Script

If your background is in HTML design, here are a few examples of retrieving data from the database and displaying it.

To display the results of a query that returns multiple rows, you can iterate through each row of the result set, printing the appropriate columns using HTML list or table tags:

```
<% FOR item IN (SELECT * FROM some_table) LOOP %>
  <TR>
  <TD><%= item.col1 %></TD>
  <TD><%= item.col2 %></TD>
  </TR>
<% END LOOP; %>
```

If you want to print out an entire database table in one operation, you can call the `OWA_UTIL.TABLEPRINT` or `OWA_UTIL.CELLSPRINT` procedures from the PL/SQL Web toolkit:

```
<% OWA_UTIL.TABLEPRINT(CTABLE => 'some_table', CATTRIBUTES => 'border=2',
CCOLUMNS => 'col1, col2', CCLAUSES => 'WHERE col1 > 5'); %>

htp.tableOpen('border=2');
owa_util.cellsprint( 'select col1, col2 from some_table where col1 > 5');
htp.tableClose;
```

Coding Tips for PSP Scripts

To share procedures, constants, and types across different PL/SQL server pages, compile them into a separate package in the database using a plain PL/SQL source file. Although you can reference packaged procedures, constants, and types from PSP scripts, the PSP scripts can only produce standalone procedures, not packages.

To make things easier to maintain, keep all your directives and declarations together near the beginning of a PL/SQL server page.

Syntax of PL/SQL Server Page Elements

You can find examples of many of these elements in "Examples of PL/SQL Server Pages" on page 13-31.

Page Directive

Specifies characteristics of the PL/SQL server page:

- What scripting language it uses.
- What type of information (MIME type) it produces.
- What code to run to handle all uncaught exceptions. This might be an HTML file with a friendly message, renamed to a `.psp` file. You must specify this same file name in the `loadpsp` command that compiles the main PSP file. You must specify exactly the same name in both the `errorPage` directive and in the `loadpsp` command, including any relative path name such as `../include/`.

Note that the attribute names `contentType` and `errorPage` are case-sensitive.

Syntax

```
<%@ page [language="PL/SQL"] [contentType="content type string"] charset="encoding" [errorPage="file.psp"] %>
```

Procedure Directive

Specifies the name of the stored procedure produced by the PSP file. By default, the name is the filename without the `.psp` extension.

Syntax

```
<%@ plsql procedure="procedure name" %>
```

Parameter Directive

Specifies the name, and optionally the type and default, for each parameter expected by the PSP stored procedure. The parameters are passed using name-value pairs, typically from an HTML form. To specify a default value of a character type, use single quotes around the value, inside the double quotes required by the directive. For example:

```
<%@ parameter="username" type="varchar2" default="'nobody'" %>
```

Syntax

```
<%@ plsql parameter="parameter name" [type="PL/SQL type"] [default="value"] %>
```

Include Directive

Specifies the name of a file to be included at a specific point in the PSP file. The file must have an extension other than `.psp`. It can contain HTML, PSP script elements, or a combination of both. The name resolution and file inclusion happens when the PSP file is loaded into the database as a stored procedure, so any changes to the file after that are not reflected when the stored procedure is run.

You must specify exactly the same name in both the include directive and in the `loadpsp` command, including any relative path name such as `../include/`.

Syntax

```
<%@ include file="path name" %>
```

Declaration Block

Declares a set of PL/SQL variables that are visible throughout the page, not just within the next `BEGIN/END` block. This element typically spans multiple lines, with individual PL/SQL variable declarations ended by semicolons.

Syntax

```
<%! PL/SQL declaration;
    [ PL/SQL declaration; ] ... %>
```

Code Block (Scriptlet)

Executes a set of PL/SQL statements when the stored procedure is run. This element typically spans multiple lines, with individual PL/SQL statements ended

by semicolons. The statements can include complete blocks, or can be the bracketing parts of `IF/THEN/ELSE` or `BEGIN/END` blocks. When a code block is split into multiple scriptlets, you can put HTML or other directives in the middle, and those pieces are conditionally executed when the stored procedure is run.

Syntax

```
<% PL/SQL statement;
   [ PL/SQL statement; ] ... %>
```

Expression Block

Specifies a single PL/SQL expression, such as a string, arithmetic expression, function call, or combination of those things. The result is substituted as a string at that spot in the HTML page that is produced by the stored procedure. You do not need to end the PL/SQL expression with a semicolon.

Syntax

```
<%= PL/SQL expression %>
```

Loading the PL/SQL Server Page into the Database as a Stored Procedure

You load one or more PSP files into the database as stored procedures. Each `.psp` file corresponds to one stored procedure. The pages are compiled and loaded in one step, to speed up the development cycle:

```
loadpsp [ -replace ] -user username/password[@connect_string]
    [ include_file_name ... ] [ error_file_name ] psp_file_name ...
```

To do a "create and replace" on the stored procedures, include the `-replace` flag.

The loader logs on to the database using the specified user name, password, and connect string. The stored procedures are created in the corresponding schema.

Include the names of all the include files (whose names do not have the `.psp` extension) before the names of the PL/SQL server pages (whose names have the `.psp` extension). Also include the name of the file specified in the `errorPage` attribute of the `page` directive. These filenames on the loadpsp command line must match exactly the names specified within the PSP `include` and `page` directives, including any relative path name such as `../include/`.

For example:

```
loadpsp -replace -user scott/tiger@orcl banner.inc error.psp display_order.psp
```

In this example:

- The stored procedure is created in the database `orcl`. The database is accessed as user scott with password tiger, both to create the stored procedure and when the stored procedure is executed.
- `banner.inc` is a file containing boilerplate text and script code, that is included by the `.psp` file. The inclusion happens when the PSP is loaded into the database, not when the stored procedure is executed.
- `error.psp` is a file containing code, text, or both that is processed when an unhandled exception occurs, to present a friendly page rather than an internal error message.
- `display_order.psp` contains the main code and text for the Web page. By default, the corresponding stored procedure is named `DISPLAY_ORDER`.

Running a PL/SQL Server Page Through a URL

Once the PL/SQL server page has been turned into a stored procedure, you can run it by retrieving an HTTP URL through a Web browser or other Internet-aware client program. The virtual path in the URL depends on the way the Web gateway is configured.

The parameters to the stored procedure are passed through either the POST method or the GET method of the HTTP protocol. With the POST method, the parameters are passed directly from an HTML form and are not visible in the URL. With the GET method, the parameters are passed as name-value pairs in the query string of the URL, separated by `&` characters, with most non-alphanumeric characters in encoded format (such as `%20` for a space). You can use the GET method to call a PSP page from an HTML form, or you can use a hard-coded HTML link to call the stored procedure with a given set of parameters.

Sample PSP URLs

Using `METHOD=GET`, the URL might look something like this:

`http://`*`sitename`*`/`*`schemaname`*`/`*`pspname`*`?`*`parmname1=value1&parmname2=value2`*

Using `METHOD=POST`, the URL does not show the parameters:

`http://`*`sitename`*`/`*`schemaname`*`/`*`pspname`*

The METHOD=GET format is more convenient for debugging and allows visitors to pass exactly the same parameters when they return to the page through a bookmark.

The METHOD=POST format allows a larger volume of parameter data, and is suitable for passing sensitive information that should not be displayed in the URL. (URLs linger on in the browser's history list and in the HTTP headers that are passed to the next-visited page.) It is not practical to bookmark pages that are called this way.

Examples of PL/SQL Server Pages

This section shows how you might start with a very simple PL/SQL server page, and produce progressively more complicated versions as you gain more confidence.

As you go through each step, you can use the procedures in "Loading the PL/SQL Server Page into the Database as a Stored Procedure" on page 13-29 and "Running a PL/SQL Server Page Through a URL" on page 13-30 to compile the PSP files and try them in a browser.

Sample Table

In this example, we use a very small table representing a product catalog. It holds the name of an item, the price, and URLs for a description and picture of the item.

```
Name       Type
---------- -------------
PRODUCT    VARCHAR2(100)
PRICE      NUMBER(7,2)
URL        VARCHAR2(200)
PICTURE    VARCHAR2(200)

Guitar
455.5
http://auction.fictional_site.com/guitar.htm
http://auction.fictional_site.com/guitar.jpg


Brown shoe
79.95
http://retail.fictional_site.com/loafers.htm
http://retail.fictional_site.com/shoe.gif

Radio
9.95
http://promo.fictional_site.com/freegift.htm
```

```
http://promo.fictional_site.com/alarmclock.jpg
```

Dumping the Sample Table

For your own debugging, you might want to display the complete contents of an SQL table. You can do this with a single call to `OWA_UTIL.TABLEPRINT`. In subsequent iterations, we use other techniques to get more control over the presentation.

```
<%@ plsql procedure="show_catalog_simple" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Complete Dump)</TITLE></HEAD>
<BODY>
<%
declare
dummy boolean;
begin
dummy := owa_util.tableprint('catalog','border');
end;
%>
</BODY>
</HTML>
```

Printing the Sample Table using a Loop

Next, we loop through the items in the table and explicitly print just the pieces we want.

- We could adjust the `SELECT` statement to retrieve only a subset of the rows or columns.
- We could change the HTML or the location of the expressions to change the appearance of each item, or the order in which the columns are shown.
- At this early stage, we pick a very simple presentation, a set of list items, to avoid any problems from mismatched or unclosed table tags.

```
<%@ plsql procedure="show_catalog_raw" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Raw Form)</TITLE></HEAD>
<BODY>
<UL>
<% for item in (select * from catalog order by price desc) loop %>
<LI>
```

```
Item = <%= item.product %><BR>
price = <%= item.price %><BR>
URL = <I><%= item.url %></I><BR>
picture = <I><%= item.picture %></I>
<% end loop; %>
</UL>
</BODY>
</HTML>
```

Once the previous simple example is working, we can display the contents in a more usable format.

- We use some HTML tags around certain values for emphasis.
- Instead of printing the URLs for the description and picture, we plug them into link and image tags so that the reader can see the picture and follow the link.

```
<%@ plsql procedure="show_catalog_pretty" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Better Form)</TITLE></HEAD>
<BODY>
<UL>
<% for item in (select * from catalog order by price desc) loop %>
<LI>
Item = <A HREF="<%= item.url %>"><%= item.product %></A><BR>
price = <BIG><%= item.price %></BIG><BR>
<IMG SRC="<%= item.picture %>">
<% end loop; %>
</UL>
</BODY>
</HTML>
```

Allowing a User Selection

We have a dynamic page, but from a user point of view it may still be dull. The results are always the same unless you update the catalog table.

- To liven up the page, we can make it accept a minimum price, and present only the items that are more expensive. (Your customers' buying criteria may vary.)
- When the page is displayed in a browser, by default the minimum price is 100 units of the appropriate currency. Later, we will see how to allow the user to pick a minimum price.

```
<%@ plsql procedure="show_catalog_partial" %>
```

```
<%@ plsql parameter="minprice" default="100" %>

<HTML>
<HEAD><TITLE>Show Items Greater Than Specified Price</TITLE></HEAD>
<BODY>
<P>This report shows the items whose price is greater than <%= minprice %>.
<UL>
<% for item in (select * from catalog where price > minprice order by price
desc) loop %>
<LI>
Item = <A HREF="<%= item.url %>"><%= item.product %></A><BR>
price = <BIG><%= item.price %></BIG><BR>
<IMG SRC="<%= item.picture %>">
<% end loop; %>
</UL>
</BODY>
</HTML>
```

This technique of filtering results is fine for some applications, such as search results, where users might worry about being overwhelmed by choices. But in a retail situation, you might want to use an alternative technique so that customers can still choose to purchase other items.

- Instead of filtering the results through a `WHERE` clause, we can retrieve the entire result set, then take different actions for different returned rows.
- We can change the HTML to highlight the output that meets their criteria. In this case, we use the background color for an HTML table row. We could also insert a special icon, increase the font size, or use some other technique to call attention to the most important rows.
- At this point, where we want to present a specific user experience, it becomes worth the trouble to lay out the results in an HTML table.

```
<%@ plsql procedure="show_catalog_highlighted" %>
<%@ plsql parameter="minprice" default="100" %>
<%! color varchar2(7); %>

<HTML>
<HEAD><TITLE>Show Items Greater Than Specified Price</TITLE></HEAD>
<BODY>
<P>This report shows all items, highlighting those whose price is
 greater than <%= minprice %>.
<TABLE BORDER>
<TR>
<TH>Product</TH>
```

```
<TH>Price</TH>
<TH>Picture</TH>
</TR>
<%
for item in (select * from catalog order by price desc) loop
  if item.price > minprice then
    color := '#CCCCFF';
  else
    color := '#CCCCCC';
  end if;
%>
<TR BGCOLOR="<%= color %>">
<TD><A HREF="<%= item.url %>"><%= item.product %></A></TD>
<TD><BIG><%= item.price %></BIG></TD>
<TD><IMG SRC="<%= item.picture %>"></TD>
</TR>
<% end loop; %>
</TABLE>
</BODY>
</HTML>
```

Sample HTML Form to Call a PL/SQL Server Page

Here is a bare-bones HTML form that allows someone to enter a price, and then calls the `SHOW_CATALOG_PARTIAL` stored procedure passing the entered value as the `MINPRICE` parameter.

To avoid coding the entire URL of the stored procedure in the `ACTION=` attribute of the form, we can make the form a PSP file so that it goes in the same directory as the PSP file it calls. Even though this HTML file has no PL/SQL code, we can give it a `.psp` extension and load it as a stored procedure into the database. When the stored procedure is run, it just displays the HTML exactly as it appears in the file.

```
<html>
<body>
<form method="POST" action="show_catalog_partial">
<p>Enter the minimum price you want to pay:
<input type="text" name="minprice">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Note: An HTML form is different from other forms you might produce with tools and programming languages. It is part of an HTML file, delimited by <FORM> and </FORM> tags, where someone can make choices and enter data, then transmit those choices to a server-side program using the CGI protocol.

To produce a complete application using PSP, you might need to learn the syntax of <INPUT>, <SELECT>, and other HTML tags related to forms.

Including JavaScript in a PSP File

To produce an elaborate HTML file, perhaps including dynamic content such as JavaScript, you can simplify the source code by implementing it as a PSP. This technique avoids having to deal with nested quotation marks, escape characters, concatenated literals and variables, and indentation of the embedded content.

For example, here is how an HTML file containing JavaScript might be generated using a PSP:

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="graph" %>
<%!
-- Begin with a date that does not exist in the audit table
last_timestamp date := sysdate + 1;
%>
<html>
<head>
<title>Usage Statistics</title>
<script language="JavaScript">
<!--
d=document

// Draw a horizontal graph line using a graphic that is stretched
// by a scaling factor.
function graph(howmuch)
{
  preamble = "<img src='/images/graph_line.gif' height='8' width='"
  climax = howmuch * 4;
  denouement = "'> (" + howmuch + ")\n"
  d.write( preamble + climax + denouement )
}
```

```
// -->
</script>
</head>
<body text="#000000" bgcolor="#FFFFFF">
<h1>Usage Statistics</h1>

<table border=1>

<%
-- For each day, count how many times each procedure was called.
for item in (select trunc(time_stamp) t, count(*) n, procname p
  from audit_table group by trunc(time_stamp), procname
  order by trunc(time_stamp) desc, procname)
loop
-- At the start of each day's data, print the date.
  if item.t != last_timestamp then
    htp.print('<tr><td colspan=2><font size="+2">');
    htp.print(htf.bold(item.t));
    htp.print('</font></td></tr>');
    last_timestamp := item.t;
end if;
%>

<tr><td><%= item.p %></a>:
<td>
<!-- Render an image of variable width to represent the data value. -->
<script language="JavaScript">
<!--
graph(<%= item.n %>)
// -->
</script>
</td>
</tr>

<% end loop; %>

</table>

</body>
</html>
```

Coding this procedure as a regular PL/SQL stored procedure would result in convoluted lines with doubled apostrophes, such as this:

```
htp.print('preamble =
          "<img src=''/images/graph_line.gif'' height=''8'' width=''"');
```

Debugging PL/SQL Server Page Problems

As you begin experimenting with PSP, and as you adapt your first simple pages into more elaborate ones, keep these guidelines in mind when you encounter problems:

- The first step is to get all the PL/SQL syntax and PSP directive syntax right. If you make a mistake here, the file does not compile.
  - Make sure you use semicolons to terminate lines where required.
  - If a value must be quoted, quote it. You might need to enclose a single-quoted value (needed by PL/SQL) inside double quotes (needed by PSP).
  - Mistakes in the PSP directives are usually reported through PL/SQL syntax messages. Check that your directives use the right syntax, that directives are closed properly, and that you are using the right element (declaration, expression, or code block) depending on what goes inside it.
  - PSP attribute names are case-sensitive. Most are specified in all lowercase; `contentType` and `errorPage` must be specified as mixed-case.
- The next step is to run the PSP file by requesting its URL in a Web browser. At this point, you might get an error that the file is not found.
  - Make sure you are requesting the right virtual path, depending on the way the Web gateway is configured. Typically, the path includes the hostname, optionally a port number, the schema name, and the name of the stored procedure (with no `.psp` extension).
  - Remember, if you use the `-replace` option when compiling the file, the old version of the stored procedure is erased. So, after a failed compilation, you must fix the error or the page is not available. You might want to test new scripts in a separate schema until they are ready, then load them into the production schema.
  - If you copied the file from another file, remember to change any procedure name directives in the source to match the new file name.
  - Once you get one file-not-found error, make sure to request the latest version of the page the next time. The error page might be cached by the

browser. You might need to press Shift-Reload in the browser to bypass its cache.

- When the PSP script is run, and the results come back to the browser, use standard debugging techniques to check for and correct wrong output. The tricky part is to set up the interface between different HTML forms, scripts, and CGI programs so that all the right values are passed into your page. The page might return an error because of a parameter mismatch.
  - To see exactly what is being passed to your page, use `METHOD=GET` in the calling form so that the parameters are visible in the URL.
  - Make sure that the form or CGI program that calls your page passes the correct number of parameters, and that the names specified by the `NAME=` attributes on the form match the parameter names in the PSP file. If the form includes any hidden input fields, or uses the `NAME=` attribute on the `Submit` or `Reset` buttons, the PSP file must declare equivalent parameters.
  - Make sure that the parameters can be cast from string into the correct PL/SQL types. For example, do not include alphabetic characters if the parameter in the PSP file is declared as a `NUMBER`.
  - Make sure that the query string of the URL consists of name-value pairs, separated by equals signs, especially if you are passing parameters by constructing a hard-coded link to the page.
  - If you are passing a lot of parameter data, such as large strings, you might exceed the volume that can be passed with `METHOD=GET`. You can switch to `METHOD=POST` in the calling form without changing your PSP file.
  - Although the `loadpsp` command reports line numbers correctly when there is a syntax error in your source file, line numbers reported for runtime errors refer to a transformed version of the source and do not match the line numbers in the original source. When you encounter errors like these, that produce an error trace instead of the expected Web page, you will need to locate the error through exception handlers and by printing debug output.

Putting an Application using PL/SQL Server Pages into Production

When you start developing an application with PSP, you may spend most of your time getting the logic correct in the script. Before putting the application into production, consider other issues such as usability and download speed:

- Pages can be rendered faster in the browser if the `HEIGHT=` and `WIDTH=` attributes are specified for all images. You might standardize on picture sizes, or

store the height and width of images in the database along with the data or URL.

- For viewers who turn off graphics, or who use alternative browsers that read the text out loud, include a description of significant images using the `ALT=` attribute. You might store the description in the database along with the image.
- Although an HTML table provides a good way to display data, a large table can make your application seem slow. Often, the reader sees a blank page until the entire table is downloaded. If the amount of data in an HTML table is large, consider splitting the output into multiple tables.
- If you set text, font, or background colors, test your application with different combinations of browser color settings:
  - Test what happens if you override just the foreground color in the browser, or just the background color, or both.
  - Generally, if you set one color (such as the foreground text color), you should set all the colors through the `<BODY>` tag, to avoid hard-to-read combinations like white text on a white background.
  - If you use a background image, specify a similar background color to provide proper contrast for viewers who do not load graphics.
  - If the information conveyed by different colors is crucial, consider using some other method instead of or in addition to the color change. For example, you might put a graphic icon next to special items in a table. Some of your viewers may see your page on a monochrome screen, or on browsers that cannot represent different colors. (Such browsers might fit in a shirt pocket and use a stylus for input.)
- Providing context information prevents users from getting lost. Include a descriptive `<TITLE>` tag for your page. If the user is partway through a procedure, indicate which step is represented by your page. Provide links to logical points to continue with the procedure, return to a previous step, or cancel the procedure completely. Many pages might use a standard set of links that you embed using the include directive.
- In any entry fields, users might enter incorrect values. Where possible, use select lists to present a set of choices. Validate any text entered in a field before passing it to SQL. The earlier you can validate, the better; a JavaScript routine can detect incorrect data and prompt the user to correct it before they press the `Submit` button and make a call to the database.

- Browsers tend to be lenient when displaying incorrect HTML. But what looks OK in one browser might look bad or might not display at all in another browser.
  - Pay attention to HTML rules for quotation marks, closing tags, and especially for anything to do with tables.
  - Minimize the dependence on tags that are only supported by a single browser. Sometimes you can provide an extra bonus using such tags, but your application should still be usable with other browsers.
  - You can check the validity, and even in some cases the usability, of your HTML for free at many sites on the World Wide Web.

Enabling PL/SQL Web Applications for XML

You might find that a PL/SQL Web application needs to accept data in XML format, or produce tagged output that is XML rather than HTML.

When displaying output, you can set the MIME type of the Web page to `text/xml` so that an XML-enabled browser or other Web client software can render it as XML.

You can also use a number of built-in features like the `XMLTYPE` type, `DBMS_XMLQUERY` and `DBMS_XMLSAVE` packages, and `SYS_XMLGEN` and `SYS_XMLAGG` functions within your application. For information about these features, see *Oracle XML Developer's Kit Programmer's Guide*.

14

Porting Non-Oracle Applications to Oracle Database 10*g*

Often, a programming project requires adapting existing code rather than writing new code. When that code comes from some other database platform, it is important to understand the Oracle Database features that are designed to make porting easy.

Topics include the following:

- Performing Natural Joins and Inner Joins
- Migrating a Schema and Data from Another Database System
- Performing Several Comparisons within a Query

Performing Natural Joins and Inner Joins

When porting queries from other database systems to Oracle Database, you can code Oracle Database queries using ANSI-compliant notation for joins. For example:

```
SELECT * FROM a NATURAL JOIN b;
SELECT * FROM a JOIN b USING (c1);
SELECT * FROM a JOIN b USING (c1) WHERE c2 > 100;
SELECT * FROM a NATURAL JOIN b INNER JOIN c;
```

The standard notation makes the relations between the tables explicit, and saves you from coding equality tests for join conditions in the `WHERE` clause. Support for full outer joins also eliminates the need for complex workarounds to do those queries.

Because different vendors support varying amounts of standard join syntax, and some vendors introduce their own syntax extensions, you might still need to rewrite some join queries.

See *Oracle Database SQL Reference* for full syntax of the `SELECT` statement and the support for join notation.

Migrating a Schema and Data from Another Database System

Oracle provides a free product called the Oracle Migration Workbench that can convert a schema (including data, triggers, and stored procedures) from other database products to Oracle Database. Although the product runs on Windows, it can transfer data from databases on other operating systems to Oracle Database running on any operating system.

By using this product, you can avoid having to write your own applications to convert your legacy data when switching to Oracle Database. Related technology lets you convert certain kinds of source code, for example to migrate Visual Basic code to Java.

For the current set of supported databases, see the OTN Web site:

`http://otn.oracle.com/`

Performing Several Comparisons within a Query

When you want to choose from many different conditions within a query, you can use:

- The SQL statement CASE:

```
SELECT CASE
  WHEN day IN
    ('Monday','Tuesday','Wednesday','Thursday','Friday')
    THEN 'weekday'
  WHEN day in
    ('Saturday','Sunday')
    THEN 'weekend'
  ELSE 'unknown day' END
FROM DUAL;
```

  This technique helps performance when you can replace a call to a PL/SQL function with a test done directly in SQL.

  Oracle Database supports the SQL-92 notation for searched case, simple case, NULLIF, and COALESCE.

- The SQL function DECODE:

```
SELECT DECODE (day,
  'Monday', 'weekday',
  'Tuesday', 'weekday',
  'Wednesday', 'weekday',
  'Thursday', 'weekday',
  'Friday', 'weekday',
  'Saturday', 'weekend',
  'Sunday', 'weekend,
  'unknown day')
INTO day_category FROM DUAL;
```

  This construct lets you test a variable against a number of different alternatives, and return a different value in each case. The final value is used when none of the alternatives is matched.

The CASE technique is more portable, and is preferable for new code.

15

Using Flashback Features

This chapter discusses the following flashback topics:

- Overview of Flashback Features
- Database Administration Tasks Before Using Flashback Features
- Using Flashback Query (SELECT ... AS OF)
- Using the DBMS_FLASHBACK Package
- Using ORA_ROWSCN
- Using Flashback Version Query
- Using Flashback Transaction Query
- Flashback Tips

See Also:

- *Oracle Database Backup and Recovery Advanced User's Guide* and *Oracle Database Administrator's Guide* for information on flashback features designed for database administration tasks, such as Oracle Flashback Database and Oracle Flashback Table
- *Oracle Database SQL Reference* for the syntax of SQL extensions for flashback features

Overview of Flashback Features

Oracle Database has a group of features, known collectively as **flashback**, that provide ways to view past states of database objects, or to return database objects to a previous state, without using traditional point-in-time recovery.

Flashback features of the database can be used to:

- Perform queries that return past data.
- Perform queries that return metadata showing a detailed history of changes to the database.
- Recover tables or individual rows to a previous point in time.

Flashback features use the *Automatic Undo Management* system to obtain metadata and historical data for transactions. They rely on **undo data**: records of the effects of individual transactions. Undo data is persistent and survives a database malfunction or shutdown. Using flashback features, you employ undo data to query past data or recover from logical corruptions. Besides your use of it in flashback operations, undo data is used by Oracle Database to do the following:

- rollback active transactions
- recover terminated transactions using database or process recovery
- provide read consistency for SQL queries

> **See Also:** *Oracle Database Concepts* for more information about flashback features and automatic undo management

Application Development Features

In application development, flashback features can be used to report on historical data or undo erroneous changes. Flashback features that allow you to do this include:

- *Oracle Flashback Query* - retrieve data for a time in the past that you specify using the `AS OF` clause of the `SELECT` statement.
- *Oracle Flashback Version Query* - retrieve metadata and historical data for a specific time interval. You can view all the rows of a table that ever existed during a given time interval. Metadata about the different versions of rows includes start and end time, type of change operation, and identity of the transaction that created the row version. You use the `VERSIONS BETWEEN` clause of the `SELECT` statement to create a Flashback Version Query.

- *Oracle Flashback Transaction Query* - retrieve metadata and historical data for a given transaction, or for all transactions within a given time interval. You can also obtain the SQL code to undo the changes to particular rows affected by a transaction. You typically use Flashback Transaction Query in conjunction with a Flashback Version Query that provides the transaction IDs for the rows of interest. To perform a Flashback Transaction Query, you select from the `FLASHBACK_TRANSACTION_QUERY` view.
- *`DBMS_FLASHBACK` package* – set the clock back to a time in the past, to examine data current at that time.

Database Administration Features

You can use the `DBMS_FLASHBACK` package, Flashback Query, Flashback Version Query, and Flashback Transaction Query for application development or interactively, as a database user or administrator.

Other flashback features are typically used only in database administration tasks:

- *Oracle Flashback Table* - recover a table to its state at a previous point in time. You can restore table data while the database is on line, undoing changes to only the specified table.
- *Oracle Flashback Drop* - recover a dropped table. This reverses the effects of a `DROP TABLE` statement.
- *Oracle Flashback Database* – quickly return the database to an earlier point in time, by undoing all of the changes that have taken place since then. This is fast, because you do not have to restore database backups.

Flashback Database, Flashback Table, and Flashback Drop are primarily provided as data recovery mechanisms and are therefore documented elsewhere. The other flashback features, while valuable in data recovery scenarios, are also used in contexts such as application development. They are therefore the focus of this chapter.

See Also:

- *Oracle Database Backup and Recovery Advanced User's Guide*
- *Oracle Database Administrator's Guide*, Chapter "Using Flashback Drop and Managing the Recycle Bin", for information on the Flashback Drop feature
- *Oracle Database Administrator's Guide*, Chapter "Recovering Tables Using the Flashback Table Feature", for information on the Flashback Table feature
- *Oracle Database Administrator's Guide*, Chapter "Auditing Table Changes Using Flashback Transaction Query", for information on Automatic Undo Management

Database Administration Tasks Before Using Flashback Features

Before you can use flashback features in your application, the following administrative tasks must be performed to configure your database. Consult with your database administrator to perform these tasks.

- Create an undo tablespace with enough space to keep the required data for flashback operations. The more often the data is updated, the more space is required. Calculating the space requirements is usually performed by a database administrator; you can find the calculation formula in the *Oracle Database Administrator's Guide*.
- Enable Automatic Undo Management – see *Oracle Database Administrator's Guide*. In particular, this involves setting the following database configuration parameters: `UNDO_MANAGEMENT`, `UNDO_TABLESPACE`, and `UNDO_RETENTION`.
- Set the `RETENTION GUARANTEE` clause for the undo tablespace, to *ensure* that unexpired undo is not discarded – `UNDO_RETENTION` is not, by itself, a strict guarantee. If the system is under space pressure, then unexpired undo may be overwritten with freshly generated undo; `RETENTION GUARANTEE` prevents this.
- Grant flashback privileges to users, roles, or applications that need to use flashback features, as follows:
  - *`DBMS_FLASHBACK` package* – Grant `EXECUTE` privilege on `DBMS_FLASHBACK` to provide access to the features in this package.

- *Flashback Query* and *Flashback Version Query* – Grant `FLASHBACK` and `SELECT` privileges on specific objects to be accessed during queries, or grant the `FLASHBACK ANY TABLE` privilege to allow queries on *all* tables.
- *Flashback Transaction Query* – Grant the `SELECT ANY TRANSACTION` privilege.
- *Execution of undo SQL code* – Grant `SELECT`, `UPDATE`, `DELETE`, and `INSERT` privileges for specific tables, as appropriate, to permit execution of undo SQL code retrieved by a Flashback Transaction Query.

- To enable flashback operations on specific `LOB` columns of a table, use the `ALTER TABLE` command with the `RETENTION` option. Because undo data for `LOB` columns can be voluminous, you must define which `LOB` columns to use with flashback operations.

  See Also:

  - *Oracle Database Backup and Recovery Advanced User's Guide* and *Oracle Database Administrator's Guide* for information on DBA tasks such as setting up automatic undo management and granting privileges
  - *Oracle Database Application Developer's Guide - Large Objects* for information on LOB storage and the `RETENTION` parameter

Using Flashback Query (SELECT ... AS OF)

You perform a Flashback Query using a `SELECT` statement with an `AS OF` clause. You use a Flashback Query to retrieve data as it existed at some time in the past. The query explicitly references a past time using a timestamp or SCN. It returns committed data that was current at that point in time.

Potential uses of Flashback Query include:

- Recovering lost data or undoing incorrect, committed changes. For example, if you mistakenly delete or update rows, and then commit them, you can immediately repair the mistake.
- Comparing current data with the corresponding data at some time in the past. For example, you might run a daily report that shows the change in data from yesterday. You can compare individual rows of table data, or find intersections or unions of sets of rows.
- Checking the state of transactional data at a particular time. For example, you could verify the account balance of a certain day.

- Simplifying application design, by removing the need to store some kinds of temporal data. Using a Flashback Query, you can retrieve past data directly from the database.
- Applying packaged applications, such as report generation tools, to past data.
- Providing self-service error correction for an application, enabling users to undo and correct their errors.

> **See Also:** *Oracle Database SQL Reference* for details on the syntax of the `SELECT... AS OF` statement

Examining Past Data: Example

This example uses a Flashback Query to examine the state of a table at a previous time. Suppose, for instance, that a DBA discovers at 12:30 PM that data for employee `JOHN` had been deleted from the `employee` table, and the DBA knows that at 9:30AM the data for `JOHN` was correctly stored in the database. The DBA can use a Flashback Query to examine the contents of the table at 9:30, to find out what data had been lost. If appropriate, the DBA can then re-insert the lost data in the database.

The following query retrieves the state of the employee record for `JOHN` at 9:30AM, April 4, 2003:

```
SELECT * FROM employee AS OF TIMESTAMP
   TO_TIMESTAMP('2003-04-04 09:30:00', 'YYYY-MM-DD HH:MI:SS')
   WHERE name = 'JOHN';
```

This update then restores John's information to the `employee` table:

```
INSERT INTO employee
    (SELECT * FROM employee AS OF TIMESTAMP
     TO_TIMESTAMP('2003-04-04 09:30:00', 'YYYY-MM-DD HH:MI:SS')
     WHERE name = 'JOHN');
```

Tips for Using Flashback Query

Keep the following in mind when using a Flashback Query (`SELECT ... AS OF`):

- You can specify or omit the `AS OF` clause for each table, and specify different times for different tables. Use an `AS OF` clause in a query to perform DDL operations (such as creating and truncating tables) or DML operations (such as inserting and deleting) in the same session as the query.

- To use the results of a Flashback Query in a DDL or DML statement that affects the current state of the database, use an AS OF clause inside an INSERT or CREATE TABLE AS SELECT statement.
- When choosing whether to use a timestamp or an SCN in Flashback Query, remember that Oracle Database uses SCNs internally and maps these to timestamps at a granularity of 3 seconds. If a possible 3-second error (maximum) is important to a Flashback Query in your application, use an SCN instead of a timestamp. See "Flashback Tips – General".
- You can create a view that refers to past data by using the AS OF clause in the SELECT statement that defines the view. If you specify a relative time by subtracting from SYSDATE, the past time is *recalculated* for each query. For example:

```
CREATE VIEW hour_ago AS
  SELECT * FROM employee AS OF
    TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE);
```

  Note that shortly after a change in daylight savings time, SYSDATE - 1 might refer to either 23 or 25 hours ago, not 24.
- You can use the AS OF clause in self-joins, or in set operations such as INTERSECT and MINUS, in order to extract or compare data from two different times. You can store the results by preceding a Flashback Query with a CREATE TABLE AS SELECT or INSERT INTO TABLE SELECT statement. For example, this query re-inserts into table employee the rows that were present there an hour ago:

```
INSERT INTO employee
   (SELECT * FROM employee AS OF
        TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE))
   MINUS SELECT * FROM employee);
```

Using the DBMS_FLASHBACK Package

The DBMS_FLASHBACK package generally provides the same functionality as Flashback Query, but Flashback Query can sometimes be more convenient to use.

The DBMS_FLASHBACK package acts as a time machine: you can turn back the clock, carry out normal queries as if you were at that time in the past, then return to the present. Because you can use the DBMS_FLASHBACK package to perform queries on past data without special clauses such as AS OF or VERSIONS BETWEEN, you can

reuse existing PL/SQL code, without change, to interrogate the database at times in the past.

You must have the EXECUTE privilege on the DBMS_FLASHBACK package.

To use the DBMS_FLASHBACK package in your PL/SQL code:

1. Call DBMS_FLASHBACK.ENABLE_AT_TIME or DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER to turn back the clock to a given time in the past. After this, all queries retrieve data that was current at the specified time.
2. Perform normal queries (that is, without any special flashback-feature syntax, such as AS OF). The database is automatically queried at the specified past time. Perform *only queries*; do not try to perform DDL or DML operations.
3. Call DBMS_FLASHBACK.DISABLE to return to the present. (You must call DISABLE before calling ENABLE... again for a different time. You cannot nest ENABLE /DISABLE pairs.)

You can use a cursor to store the results of queries into the past. To do this, open the cursor before calling DBMS_FLASHBACK.DISABLE. After storing the results and then calling DISABLE, you can do the following:

- Perform INSERT or UPDATE operations, to modify the current database state using the stored results from the past.
- Compare current data with the past data: After calling DISABLE, open a second cursor. Fetch from the first cursor to retrieve past data; fetch from the second cursor to retrieve current data. You can store the past data in a temporary table, and then use set operators such as MINUS or UNION to contrast or combine the past and current data.

You can call DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER at any time to obtain the current System Change Number (SCN). Note that the *current* SCN is always returned; this takes no account of previous calls to DBMS_FLASHBACK.ENABLE*.

See Also:

- *PL/SQL Packages and Types Reference* for details about the DBMS_FLASHBACK package
- *Oracle Database Reference* and *Oracle Database Recovery Manager Reference* for information about SCNs

Using ORA_ROWSCN

ORA_ROWSCN is a pseudocolumn of any table that is not fixed or external. It represents the SCN of the *most recent change* to a given row; that is, the latest COMMIT operation for the row. For example:

```
SQL> SELECT ora_rowscn, name, salary FROM employee WHERE empno = 7788;

ORA_ROWSCN    NAME    SALARY
----------    ----    ------
    202553    Fudd      3000
```

The latest COMMIT operation for the row took place at approximately SCN 202553. (You can use function SCN_TO_TIMESTAMP to convert an SCN, like ORA_ROWSCN, to the corresponding TIMESTAMP value.)

ORA_SCN is in fact a conservative upper bound of the latest commit time: the actual commit SCN can be somewhat earlier. ORA_SCN is more precise (closer to the actual commit SCN) for a row-dependent table (created using CREATE TABLE with the ROWDEPENDENCIES clause).

Noteworthy uses of ORA_ROWSCN in application development include concurrency control and client cache invalidation. To see how you might use it in concurrency control, consider the following scenario.

Your application examines a row of data, and records the corresponding ORA_ROWSCN as 202553. Later, the application needs to update the row, but only if its record of the data is still accurate. That is, this particular update operation depends, logically, on the row not having been changed. The operation is therefore made conditional on the ORA_ROWSCN being still 202553. Here is an equivalent interactive command:

```
SQL> UPDATE employee SET salary = salary + 100
     WHERE empno = 7788 AND ora_rowscn = 202553;

0 rows updated.
```

The conditional update fails in this case, because the ORA_ROWSCN is no longer 202553. This means that some user or another application changed the row and performed a COMMIT more recently than the recorded ORA_ROWSCN.

Your application queries again to obtain the new row data and ORA_ROWSCN. Suppose that the ORA_ROWSCN is now 415639. The application tries the conditional update again, using the new ORA_ROWSCN. This time, the update succeeds, and it is committed. Here is an interactive equivalent:

```
SQL> UPDATE employee SET salary = salary + 100
     WHERE empno = 7788 AND ora_rowscn = 415639;

1 row updated.

SQL> COMMIT;

Commit complete.

SQL> SELECT ora_rowscn, name, salary FROM employee WHERE empno = 7788;

ORA_ROWSCN    NAME    SALARY
----------    ----    ------
    465461    Fudd      3100
```

The SCN corresponding to the new `COMMIT` is `465461`.

Besides using `ORA_ROWSCN` in an `UPDATE` statement `WHERE` clause, you can use it in a `DELETE` statement `WHERE` clause or the `AS OF` clause of a Flashback Query.

See Also:

- *Oracle Database SQL Reference*

Using Flashback Version Query

You use a Flashback Version Query to retrieve the different versions of specific rows that existed during a given time interval. A new row version is created whenever a `COMMIT` statement is executed.

You specify a Flashback Version Query using the `VERSIONS BETWEEN` clause of the `SELECT` statement. Here is the syntax:

```
VERSIONS {BETWEEN {SCN | TIMESTAMP} start AND end}
```

where *start* and *end* are expressions representing the start and end of the time interval to be queried, respectively. The interval is *closed* at both ends: the upper and lower limits specified (*start* and *end*) are both included in the time interval.

The Flashback Version Query returns a table with a *row for each version* of the row that existed at any time during the time interval you specify. Each row in the table includes pseudocolumns of metadata about the row version, described in Table 15–1. This information can reveal when and how a particular change (perhaps erroneous) occurred to your database.

Table 15–1 Flashback Version Query Row Data Pseudocolumns

Pseudocolumn Name	Description
VERSIONS_STARTSCN, VERSIONS_STARTTIME	Starting System Change Number (SCN) or TIMESTAMP when the row version was created. This identifies the time when the data first took on the values reflected in the row version. You can use this to identify the past target time for a Flashback Table or Flashback Query operation. If this is NULL, then the row version was created before the lower time bound of the query BETWEEN clause.
VERSIONS_ENDSCN, VERSIONS_ENDTIME	SCN or TIMESTAMP when the row version expired. This identifies the row expiration time. If this is NULL, then either the row version was still current at the time of the query or the row corresponds to a DELETE operation.
VERSIONS_XID	Identifier of the transaction that created the row version.
VERSIONS_OPERATION	Operation performed by the transaction: I for insertion, D for deletion, or U for update. The version is that of the row that was inserted, deleted, or updated; that is, the row *after* an INSERT operation, the row *before* a DELETE operation, or the row affected by an UPDATE operation. *Note:* For user updates of an index key, a Flashback Version Query may treat an UPDATE operation as two operations, DELETE plus INSERT, represented as two version rows with a D followed by an I VERSIONS_OPERATION.

A given row version is valid starting at its time VERSIONS_START* up to, but not including, its time VERSIONS_END*. That is, it is valid for any time *t* such that VERSIONS_START* <= *t* < VERSIONS_END*. For example, the following output indicates that the salary was 10243 from September 9, 2002, included, to November 25, 2003, not included.

```
VERSIONS_START_TIME     VERSIONS_END_TIME     SALARY
-------------------     -----------------     ------
09-SEP-2003             25-NOV-2003           10243
```

Here is a typical Flashback Version Query:

```
SELECT versions_startscn, versions_starttime,
       versions_endscn, versions_endtime,
       versions_xid, versions_operation,
       name, salary
  FROM employee
```

```
VERSIONS BETWEEN TIMESTAMP
    TO_TIMESTAMP('2003-07-18 14:00:00', 'YYYY-MM-DD HH24:MI:SS')
AND TO_TIMESTAMP('2003-07-18 17:00:00', 'YYYY-MM-DD HH24:MI:SS')
WHERE name = 'JOE';
```

Pseudocolumn VERSIONS_XID provides a unique identifier for the transaction that put the data in that state. You can use this value in connection with a Flashback Transaction Query to locate metadata about this transaction in the FLASHBACK_TRANSACTION_QUERY view, including the SQL required to undo the row change and the user responsible for the change – see "Using Flashback Transaction Query" on page 15-12.

> **See Also:** *Oracle Database SQL Reference* for information on the Flashback Version Query pseudocolumns and the syntax of the VERSIONS clause

Using Flashback Transaction Query

A Flashback Transaction Query is a query on the view FLASHBACK_TRANSACTION_QUERY. You use a Flashback Transaction Query to obtain transaction information, including SQL code that you can use to undo each of the changes made by the transaction.

> **See Also:** *Oracle Database Backup and Recovery Advanced User's Guide.* and *Oracle Database Administrator's Guide* for information on how a DBA can use the Flashback Table feature to restore an entire table, rather than individual rows

As an example, the following statement queries the FLASHBACK_TRANSACTION_QUERY view for transaction information, including the transaction ID, the operation, the operation start and end SCNs, the user responsible for the operation, and the SQL code to undo the operation:

```
SELECT xid, operation, start_scn,commit_scn, logon_user, undo_sql
     FROM flashback_transaction_query
     WHERE xid = HEXTORAW('000200030000002D');
```

As another example, the following query uses a Flashback Version Query as a subquery to associate each row version with the LOGON_USER responsible for the row data change.

```
SELECT xid, logon_user FROM flashback_transaction_query
     WHERE xid IN (SELECT versions_xid FROM employee VERSIONS BETWEEN TIMESTAMP
```

```
      TO_TIMESTAMP('2003-07-18 14:00:00', 'YYYY-MM-DD HH24:MI:SS') AND
      TO_TIMESTAMP('2003-07-18 17:00:00', 'YYYY-MM-DD HH24:MI:SS'));
```

Flashback Transaction Query and Flashback Version Query: Example

This example demonstrates the use of a Flashback Transaction Query in conjunction with a Flashback Version Query. The example assumes simple variations of the `employee` and `departments` tables in the sample `hr` schema.

In this example, a DBA carries out the following series of actions in SQL*Plus:

```
connect hr/hr
CREATE TABLE emp
   (empno number primary key, empname varchar2(16), salary number);
INSERT INTO emp VALUES (111, 'Mike', 555);
COMMIT;

CREATE TABLE dept (deptno number, deptname varchar2(32));
INSERT INTO dept VALUES (10, 'Accounting');
COMMIT;
```

At this point, `emp` and `dept` have one row each. In terms of row versions, each table has one version of one row. Next, suppose that an erroneous transaction deletes employee id `111` from table `emp`:

```
UPDATE emp SET salary = salary + 100 where empno = 111;
INSERT INTO dept VALUES (20, 'Finance');
DELETE FROM emp WHERE empno = 111;
COMMIT;
```

Subsequently, a new transaction reinserts employee id `111` with a new employee name into the `emp` table.

```
INSERT INTO emp VALUES (111, 'Tom', 777);
UPDATE emp SET salary = salary + 100 WHERE empno = 111;
UPDATE emp SET salary = salary + 50 WHERE empno = 111;
COMMIT;
```

At this point, the DBA detects the application error and needs to diagnose the problem. The DBA issues the following query to retrieve versions of the rows in the `emp` table that correspond to empno `111`. The query uses Flashback Version Query pseudocolumns.

```
connect dba_name/password
SELECT versions_xid XID, versions_startscn START_SCN,
  versions_endscn END_SCN, versions_operation OPERATION,
```

```
  empname, salary FROM hr.emp
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
  where empno = 111;

XID              START_SCN  END_SCN   OPERATION  EMPNAME    SALARY
---------------- ---------- --------- ---------- ---------- ----------
0004000700000058 113855               I          Tom        927
000200030000002D 113564               D          Mike       555
000200030000002E 112670     113564    I          Mike       555
3 rows selected
```

The results table reads chronologically, from bottom to top. The third row corresponds to the version of the row in `emp` that was originally inserted in the table when the table was created. The second row corresponds to the row in `emp` that was deleted by the erroneous transaction. The first row corresponds to the version of the row in `emp` that was reinserted with a new employee name.

The DBA identifies transaction `000200030000002D` as the erroneous transaction and issues the following Flashback Transaction Query to audit all changes made by this transaction:

```
SELECT  xid, start_scn START, commit_scn COMMIT,
        operation OP, logon_user USER,
        undo_sql FROM flashback_transaction_query
        WHERE xid = HEXTORAW('000200030000002D');

XID               START   COMMIT  OP       USER   UNDO_SQL
----------------  -----   ------  --       ----   --------------------------
000200030000002D  195243  195244  DELETE   HR     insert into "HR"."EMP"
("EMPNO","EMPNAME","SALARY") values ('111','Mike','655');

000200030000002D  195243  195244  INSERT   HR     delete from "HR"."DEPT"
where ROWID = 'AAAKD4AABAAAJ3BAAB';

000200030000002D  195243  195244  UPDATE   HR     update "HR"."EMP"
set "SALARY" = '555' where ROWID = 'AAAKD2AABAAAJ29AAA';

000200030000002D  195243  113565  BEGIN  HR

4 rows selected
```

The rightmost column (`undo_sql`) contains the SQL code that will undo the corresponding change operation. The DBA can execute this code to undo the changes made by that transaction. The `USER` column (`logon_user`) shows the user responsible for the transaction.

A DBA might also be interested in knowing all changes made in a certain time window. In our scenario, the DBA performs the following query to view the details of all transactions that executed since the erroneous transaction identified earlier (including the erroneous transaction itself):

```
SELECT xid, start_scn, commit_scn, operation, table_name, table_owner
  FROM flashback_transaction_query
  WHERE table_owner = 'HR' AND
        start_timestamp >=
          TO_TIMESTAMP ('2002-04-16 11:00:00','YYYY-MM-DD HH:MI:SS');

XID               START_SCN  COMMIT_SCN  OPERATION  TABLE_NAME  TABLE_OWNER
----------------  ---------  ----------  ---------  ----------  -----------
0004000700000058  195245     195246      UPDATE     EMP         HR
0004000700000058  195245     195246      UPDATE     EMP         HR
0004000700000058  195245     195246      INSERT     EMP         HR
000200030000002D  195243     195244      DELETE     EMP         HR
000200030000002D  195243     195244      INSERT     DEPT        HR
000200030000002D  195243     195244      UPDATE     EMP         HR

6 rows selected
```

Flashback Tips

The following tips and restrictions apply to using flashback features.

Flashback Tips – Performance

- For better performance, generate statistics on all tables involved in a Flashback Query by using the `DBMS_STATS` package, and keep the statistics current. Flashback Query always uses the cost-based optimizer, which relies on these statistics.
- The performance of a query into the past depends on how much undo data must be accessed. For better performance, use queries to select small sets of past data using indexes, not to scan entire tables. If you must do a full table scan, consider adding a parallel hint to the query.
- The performance cost in I/O is the cost of paging in data and undo blocks that are not already in the buffer cache. The performance cost in CPU use is the cost of applying undo information to affected data blocks. When operating on changes in the recent past, flashback features essentially CPU bound.

- Use index structures for Flashback Version Query: the database keeps undo data for index changes as well as data changes. Performance of index lookup-based Flashback Version Query is an order of magnitude faster than the full table scans that are otherwise needed.
- In a Flashback Transaction Query, the type of the `xid` column is `RAW(8)`. To take advantage of the index built on the `xid` column, use the `HEXTORAW` conversion function: `HEXTORAW(xid)`.
- Flashback Query against a materialized view does not take advantage of query rewrite optimizations.

See Also: *Oracle Database Performance Tuning Guide*

Flashback Tips – General

- Use the `DBMS_FLASHBACK` package or other flashback features? Use `ENABLE/DISABLE` calls to the `DBMS_FLASHBACK` package around SQL code that you do not control, or when you want to use the same past time for several consecutive queries. Use Flashback Query, Flashback Version Query, or Flashback Transaction Query for SQL that you write, for convenience. A Flashback Query, for example, is flexible enough to do comparisons and store results in a single query.
- To obtain an SCN to use later with a flashback feature, use `DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER`.
- You can compute or retrieve a past time to use in a query by using a function return value as a timestamp or SCN argument. For example, you can perform date and time calculations by adding or subtracting an `INTERVAL` value to the value of the `SYSTIMESTAMP` function.
- You can query locally or *remotely* (Flashback Query, Flashback Version Query, or Flashback Transaction Query). for example here is a remote Flashback Query:

```
(SELECT * FROM employee@some_remote_host AS OF
        TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE);
```

- To ensure database consistency, always perform a `COMMIT` or `ROLLBACK` operation before querying past data.
- Remember that all flashback processing is done using the current session settings, such as national language and character set, not the settings that were in effect at the time being queried.

- Some DDLs that alter the structure of a table, such as drop/modify column, move table, drop partition, and truncate table/partition, invalidate any existing undo data for the table. It is not possible to retrieve data from a point earlier than the time such DDLs were executed. Trying such a query results in error ORA-1466. This restriction does not apply to DDL operations that alter the storage attributes of a table, such as `PCTFREE`, `INITTRANS` and `MAXTRANS`.
- Use an SCN to query past data at a precise time. If you use a timestamp, the actual time queried might be up to 3 seconds earlier than the time you specify. Internally, Oracle Database uses SCNs; these are mapped to timestamps at a granularity of every 3 seconds.

  For example, assume that the SCN values 1000 and 1005 are mapped to the times 8:41 and 8:46 AM respectively. A query for a time between 8:41:00 and 8:45:59 AM is mapped to SCN 1000; a Flashback Query for 8:46 AM is mapped to SCN 1005.

  Due to this time-to-SCN mapping, if you specify a time that is slightly after a DDL operation (such as a table creation) the database might actually use an SCN that is just before the DDL operation. This can result in error ORA-1466.
- You cannot retrieve past data from a `V$` view in the data dictionary. Performing a query on such a view always returns the current data. You can, however, perform queries on past data in other views of the data dictionary, such as `USER_TABLES`.

16

Using Oracle XA with Transaction Monitors

This chapter describes how to use the Oracle XA library, which is typically used in applications that work with transaction monitors. The XA features are most useful in applications where transactions interact with more than one database.

The Oracle XA library is an external interface that allows global transactions to be coordinated by a transaction manager other than that of Oracle Database. This allows inclusion of non-Oracle Database 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.

The chapter includes the following topics:

See Also:

- *X/Open CAE Specification - Distributed Transaction Processing: The XA Specification* for a general overview of XA, including basic architecture. 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.
- *Oracle Call Interface Programmer's Guide* for background and reference information about the Oracle XA library
- The Oracle Database platform-specific documentation for information on library linking filenames
- File `README.doc`, located in a directory specified in the Oracle Database platform-specific documentation for a description of changes, bugs, and restrictions in the Oracle XA library for your platform. Various XA files can be found at `%ORACLE_HOME%/rdbms/demo`.

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 16–1 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, Oracle Database 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.

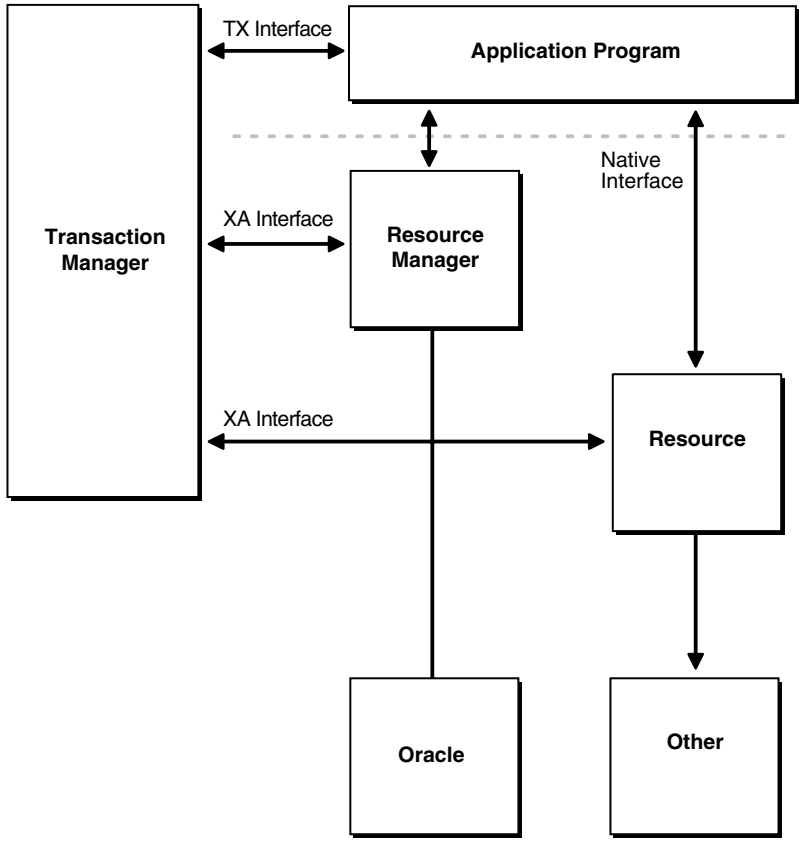
A transaction manager (TM) provides an application program interface (API) for specifying the boundaries of the transaction and manages the commit and recovery procedures.

Normally, Oracle Database acts as its own TM and manages its own commit and recovery. However, using a standards-based TM allows Oracle Database 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 Oracle Database 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 through the transaction manager through an interface called TX. The AP itself does not directly use the XA interface

Figure 16–1 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.

Required Public Information

As a resource manager, Oracle Database is required to publish the following information.

XA Feature	Oracle Database Details
`xa_switch_t` structures	The Oracle Database `xa_switch_t` structure name for static registration is `xaosw`. The Oracle Database `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 manager	The Oracle Database 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 "Defining the xa_open String" on page 16-9.
libraries	Libraries needed to link applications using Oracle XA have platform-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`, then be sure to link with `$ORACLE_HOME/lib/xaonsl.o`.
requirements	None. The functionality to support XA is part of both Standard Edition and Enterprise Edition.

XA and the Two-Phase Commit Protocol

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, then 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: TM can issue a commit for an RM only if all RMs have replied affirmatively to phase one.

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, then 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 Database (or any other resource manager) through the Oracle XA library interface.

Support for Dynamic and Static Registration

Oracle Database supports both dynamic and static registration. In dynamic registration, the RM executes an application callback before starting any work. In

static registration, you must call `xa_start` for each RM before starting any work, even if some RMs are not involved.

To use dynamic registration, both client and server must be at Oracle Database 8.0 or later. Otherwise, you can only use static registration.

Oracle XA Library Interface Subroutines

The Oracle XA library subroutines allow a TM to instruct Oracle Database what to do about transactions. Generally, the TM must "open" the resource (using `xa_open`). Typically, this results 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

Available XA Library subroutines are described in Table 16–1.

Table 16–1 XA Library Subroutines

XA Subroutine	Description
`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.

Table 16–1 (Cont.) XA Library Subroutines

XA Subroutine	Description
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

Oracle Database's XA interface includes some additional functions:

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 Database 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 Database. 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.

Developing and Installing Applications That Use the XA Libraries

This section discusses developing and installing Oracle XA applications:

Responsibilities of the DBA or System Administrator

The responsibilities of the DBA or system administrator are

1. Define the open string, with help from the application developer.

   This is described in "Defining the xa_open String" on page 16-9.

2. Make sure the `DBA_PENDING_TRANSACTIONS` view exists on the database.

   - For Oracle Database Release 8.0:

     Grant the select privilege to the `DBA_PENDING_TRANSACTIONS` view for all Oracle Database *user*(s) specified in the `xa_open` string.

   - For Oracle Database 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 Database user `SYS`. Grant the `SELECT` privilege to the `V$XATRANS$` view for all Oracle Database accounts which will be used by Oracle XA library applications.

     See Also: Your Oracle Database platform-specific documentation for the location of the `XAVIEW.SQL` script

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 starts the process that connects to Oracle Database. 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 Also:** "Defining the xa_open String" on page 16-9 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 responsibilities of the application developer are:

1. Define the open string with help from the DBA or system administrator.

   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 Also:** "Interfacing XA with Precompilers and OCIs" on page 16-16

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

```
Oracle_XA{+required_fields...} [+optional_fields...]
```

where *required_fields* is a set of any of the following:

`Acc=P//`

`Acc=P/`*`user`*`/`*`password`*

`SesTm=`*`session_time_limit`*

and where *optional_fields* is a set of any of the following:

`NoLocal=true | false`

`RAC_FAILOVER=true | false`

`DB=`*`db_name`*

`LogDir=`*`log_dir`*

`MaxCur=`*`maximum_#_of_open_cursors`*

`Objects=true | false`

`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 plus 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
```

Syntax Element	Description
`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 *Oracle Database Administrator's Guide.*
`user`	A valid Oracle Database 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`*

Syntax Element	Description
`SesTm`	Specifies the maximum length of time a transaction can be inactive before it is automatically aborted by the system.

Syntax Element	Description
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. For example, SesTM=15 indicates that the session idle time limit is 15 seconds. Entering a value of 0 is strongly *discouraged*. It might tie up resources for a long time if something goes wrong. Also, if a child process has SesTM=0, the SesTM setting is not effective after the parent process is terminated.

Optional Fields

This section describes optional fields.

NoLocal=true | false

Syntax Element	Description
NoLocal	Specifies whether local transactions are allowed. The default value is false.
true | false	If the application needs to disallow local transactions, then set the value to true.

RAC_FAILOVER=true | false

Syntax Element	Description
RAC_FAILOVER	Specifies whether XA_RECOVER waits until instance recovery finishes. The default value is false (no wait).
true | false	To force XA_RECOVER to wait until SMON finishes cache recovery, identifies in-doubt transactions, and adds them to the table DBA_2PC_PENDING, set the value to true.

DB=*db_name*

Syntax Element	Description
DB	Specifies the database name.
db_name	Indicates the name used by Oracle Database precompilers to identify the database. Application programs that use only the default database for the Oracle Database 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 Oracle Net clause in the open string. The Oracle Net clause is described later in this section. (Oracle Net was formerly known as SQL*Net and Net8.) 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*

Syntax Element	Description
LogDir	Specifies the directory on a local machine where the Oracle XA library error and tracing information may be logged.
log_dir	Indicates the path name 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.

`Loose_Coupling=true | false`

> **See Also:** "Transaction Branches" on page 16-30 for a complete explanation

`Objects=true | false`

Syntax Element	Description
`Objects`	Specifies whether the application is initialized in object mode. The default value is `false`.
`true	false`	If the application needs to use certain API calls that require object mode, such as `OCIAssignRawbytes()`, then set the value to `true`.

`MaxCur=`*`maximum_#_of_open_cursors`*

Syntax Element	Description
`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:** *Pro*C/C++ Programmer's Guide*

SqlNet=*db_link*

Syntax Element	Description
SqlNet	Specifies the Oracle Net database link. (Oracle Net was formerly known as SQL*Net and Net8.)
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 Database instance. To use the Oracle Net string without actually accessing a remote database, use the Pipe driver.

For example:

```
SqlNet=localsid1
```

localsid1 is an alias defined in the tnsnames.ora file.

Make sure that all databases to be accessed with a Oracle Net database link have an entry in /etc/oratab.

SesWt=*session_wait_limit*

Syntax Element	Description
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 Database waits before XA_RETRY is returned.

```
Threads=true | false
```

Syntax Element	Description
`Threads`	Specifies whether the application is multithreaded. The default value is `false`.
`true	false`	If the application is multithreaded, then the setting is `true`.

Interfacing XA with Precompilers and OCIs

This section describes how to use the Oracle XA library with precompilers and Oracle Call Interface (OCI).

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 for each 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_tab 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_Tab SET Sal=4500 WHERE Empno=7788;
EXEC SQL AT MANAGERS UPDATE Emp_Tab SET Mgr=7566 WHERE Empno=7788;
EXEC SQL UPDATE Emp_Tab 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 Database 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...
```

Caution: Do not use 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

Oracle Call Interface 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.

In applications that need to pass the environment handle to OCI functions, you can also call `xaoEnv()` to find that handle.

Because an application server can have multiple concurrent open Oracle Database resource managers, it should call the function `xaoSvcCtx`() with the correct arguments to obtain the correct service context.

Release 7.3

If `DB=`*db_name* is not present in the open string, then execute:

```
sqlld2(lda, NULL, 0);
```

This obtains 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));
```

This obtains the `lda` for this resource manager.

Release 8.0

If `DB=`*db_name* is not present in the open string, then execute:

```
xaoSvcCtx(NULL);
```

Alternatively, if `DB=`*db_name* is present in the open string, then execute:

```
xaoSvcCtx(db_name);
```

This gets the server context for this resource manager.

In the same way, you can execute:

```
xaoEnv(NULL);
```

or:

```
xaoEnv(db_name);
```

depending upon the open string, to get the environment handle.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about using the `OCISvcCtx`

Transaction Control using XA

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 that commit or roll back transactions. Rather, they are controlled by an API accepted by the TM that starts and stops transactions. You call the API that is defined by the transaction manager, not the XA functions listed in the following table.

The transaction managers typically control the transactions through the TX interface. This interface includes the functions described in Table 16–2.

Table 16–2 TX Interface Functions

TX Function	Description
`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 Oracle Database 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 executes 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, then the application uses the default Oracle Net driver. Thus, you must 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 WORK RELEASE` (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 before ending a transaction. In general, `release_cursor=no` should be used. Use `release_cursor=yes` only when you are certain that a statement will be executed only once.

Table 16–3 lists the TPM functions that replace regular Oracle Database commands when migrating precompiler or OCI applications to TPM applications.

Table 16–3 TPM Replacement Commands

Regular Oracle Database 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)

Table 16–3 (Cont.) TPM Replacement Commands

Regular Oracle Database Commands	TPM Functions
`SET TRANSACTION READ ONLY`	Not allowed

XA Library Thread Safety

If you use a transaction monitor that supports threads, then the Oracle XA library lets you 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 nonthreaded system, each process could be considered a thread of control, because 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 Oracle Database, 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.

Specifying Threading in the Open String

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 are created by the transaction monitor, and the application does 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 Database-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 on Threading in XA

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.
- If one thread in a process connects to Oracle Database through XA, all other threads in the process that connect to Oracle Database must also connect through XA. You cannot connect through `EXEC SQL` in one thread and through XA in another thread.

Troubleshooting XA Applications

This section discusses how to find information in case of problems or system failure. It also discusses trace files and recovery of pending transactions.

XA 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 Database 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, then 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 Database 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 Database 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 following directory (provided$ORACLE_HOME can be located):
  - $ORACLE_HOME/rdbms/trace, if the operating system is Windows
  - $ORACLE_HOME/rdbms/log, otherwise
- 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 here:

The example, xa_NULL04021992.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_Finance12151991.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 Database 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 such circumstances, and about how to decide whether the in-doubt transaction should be committed or rolled back, see the TPM documentation.

Oracle Database SYS Account Tables

There are four tables under the Oracle Database `SYS` account that contain transactions generated by regular Oracle Database 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 following SQL statement 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, then 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.

XA Issues and Restrictions

Database Links

Oracle XA applications can access other Oracle Database instances through database links, with the following restrictions:

- Use the shared server (formerly known as Multi-Threaded Server) configuration.

  This means that the transaction processing monitors (TPMs) use shared servers to open the connection to Oracle Database. The operating system network connection required for the database link is opened by the dispatcher, instead of the Oracle Database 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, Net8, or Oracle Net. (SQL*Net and Net8 are former names of Oracle Net.)

- The other database being accessed should be another Oracle Database instance.

Assuming that these restrictions are satisfied, Oracle Database allows such links and propagates the transaction protocol (prepare, rollback, and commit) to the other Oracle Database instances.

Caution: If these restrictions are not satisfied, then when you use database links within an XA transaction, it creates an O/S network connection in the Oracle Database instance that is connected to the TPM server process. Because 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 receive an ORA#24777 error.

If using the shared 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 Real Application Clusters

You can recover failed transactions from any instance of Oracle Real Application Clusters. You can also heuristically commit in-doubt transactions from any instance. An XA recover call gives a list of all prepared transactions for all instances.

A sequence of operations like the following must be performed on the same instance:

1. `xa_start`
2. `xa_commit`
3. SQL operations
4. `xa_end`
5. `xs_prepare`
6. `xa_commit` or `xs_rollback`

Under normal circumstances, an `xa_prepare`, `xa_rollback`, or `xa_commit` operation performed on a branch must be performed on the same instance that created the branch. This restriction affects load balancing. If load balancing is enabled, it is possible for sequences like the following to be performed on two

different nodes. For this reason, load balancing must not be used for the XA connection.

Node 1

1. `xa_start`
2. SQL operations
3. `xa_end(SUSPEND)`

Node 2

1. `xa_start(RESUME)`
2. SQL operations
3. `xa_end`

If an error occurs, `xa_recover` must be performed before any other XA operations. You should open the XA connection with `xa_open` using option `RAC_FAILOVER=true`.

> **Note:** The operation `xa_recover` cannot be used for switchover in normal operation.

Global uniqueness of transaction IDs (XIDs) is not guaranteed; the TM must maintain the global uniqueness of XIDs. According to the XA specification, the RM must accept XIDs from the TM. However, XA on RAC cannot determine if a given XID is unique throughout the cluster.

Suppose, for example, there is a Tx(1).Br(1) on Node 1, and another Tx(1).Br(1) on Node 2. Each of these can start and execute SQL, even though the XID is not unique.

> **See Also:** X/OPEN CAE Specification Distributed Transaction Processing: The XA Specification, ISBN 1 872630 24 3, XOPEN Document Number XO/CAE/91/300

SQL-Based Restrictions

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

Database-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 Database 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), then 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`.

Transaction Branches

Oracle Database transaction branches within the same global transaction can be coupled tightly or loosely. If they are tightly coupled, then they share locks. However, if the branches are on different instances when running Oracle Real Application Clusters, then they are loosely coupled, and they do not share locks.

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. Oracle Database 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 is 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 described in Table 16–4.

Table 16–4 Tightly and Loosely Coupled Transaction 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

Oracle Database 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 16-27

Maximum Connections for Each Thread

The maximum number of `xa_opens` for each thread is 32.

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 Oracle Database version 8.X. Grant the `SELECT` privilege on `SYS.DBA_PENDING_TRANSACTIONS` to all users that connect to Oracle Database through the XA interface.

Library Compatibility

The XA library supplied with Oracle Database Release 7.3 can be used with Oracle Database Release 8.0 or later. You must use the Release 7.2 XA library with Oracle Database Release 7.2. You can use the 8.0 library with Oracle Database Release 7.3. There is only one case of backward compatibility: an XA application that uses Release 8.0 OCI works with Oracle Database Release 7.3, 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.

Oracle Database version 8.*X* does not support 7.1.6 XA calls (although it does support 7.3 XA calls). Thus, 7.1.6 XA calls need to relink Tuxedo applications with XA libraries of Oracle Database version 8.*X*.

Link Order for Libraries

When building a TP-monitor XA application, ensure that the TP-monitors libraries (that define the symbols `ax_reg` and `ax_unreg`) are placed in the link line before Oracle Database's client shared library. If your platform does not support shared libraries or if your linker is not sensitive to ordering of libraries in the link line, use Oracle Database's non-shared client library. These link restrictions are applicable only when using XA's dynamic registration (Oracle XA switch `xaoswd`).

Changes to Oracle XA Support

XA Changes from Release 8.0 to Release 8.1

There are no changes for Release 8.1.

XA 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

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 sessions 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 are cancelled when a transaction is suspended. This means that a savepoint taken by the application in a service is 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 both the XA application and Oracle Database are Version 8.

See Also: "Extensions to the XA Interface" on page 16-7

Loosely Coupled Transaction Branches Are Supported

The database server for Oracle Database version 8 and later supports both loosely and tightly coupled transaction branches in a single Oracle Database instance. The server for Oracle Database version 7 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 means that 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 Database Version 8. It is, however, still necessary for Version 7.3 applications.

> **See Also:** "Responsibilities of the DBA or System Administrator" on page 16-8

XA Library Use with Oracle Real Application Clusters Option on All Platforms

It was not possible with Version 7 to use the Oracle XA library together with the Oracle Real Application Clusters 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 Real Application Clusters option, then you can run the Oracle XA library.

Transaction Recovery for Oracle Real Application Clusters Has Been Improved

All transactions can be recovered from any instance of Oracle Real Application Clusters. Use the `xa_recover` call to provide a snapshot of the pending transactions.

Both Global and Local Transactions Are Possible

It is possible to have both global and local transactions within the same XA connection. Local transactions are transactions that are completely coordinated by Oracle Database. For example, the following update belongs to a local transaction.

```
CONNECT scott/tiger;
UPDATE Emp_tab 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, Oracle Database acts as a subordinate and processes the XA commands issued by the transaction manager. The following update belongs to a global transaction.

```
xa_open(oracle_xa+acc=p/SCOTT/TIGER+sestm=10", 1, TMNOFLAGS);
```

```
                              /* Transaction manager opens */
                              /* connection to Oracle Database */
tpbegin();                    /* begin global transaction, the transaction*/
                              /* manager issues XA commands to Oracle Database */
                              /* to start a global transaction */
UPDATE Emp_tab SET Sal = Sal + 1;
                              /* Update is performed in the */
                              /* global transaction*/
tpcommit();                   /* commit global transaction, */
                              /* the transaction manager issues XA commands*/
                              /* to Oracle Database to commit */
                              /* the global transaction */
```

The database server for Oracle Database version 7 forbids a local transaction from being started in an XA connection. The following update would return an `ORA-2041` error code.

```
xa_open("oracle_xa+acc=p/SCOTT/TIGER+sestm=10" , 1, TMNOFLAGS);
                              /* Transaction manager opens */
                              /*connection to Oracle Database */
UPDATE Emp_tab SET Sal = Sal + 1; /* Oracle Databse version 7 returns error */
```

Oracle Database version 8.*X* and later allows local transactions to be started in an XA connection. The only restriction is that the local transaction must be committed or rolled back before starting a global transaction in the connection.

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 be set to false when connected to Oracle Database version 7. If set to true, then global transaction branches are loosely coupled; in other words, locks are not shared between branches.

- `SesWt`

  This parameter value indicates the time-out limit when waiting for a transaction branch that is being used by another session. If Oracle Database cannot switch to the transaction branch within `SesWt` seconds, then `XA_RETRY` is returned.

Two parameters have been made obsolete and should only be used when connected to Oracle Database Release 7.3.

- `GPWD`

The group password is not used by Oracle Database version 8.*X* or later. A session that is logged in with the same user name as the session that created a transaction branch is allowed to switch to the transaction branch.

- `SesCacheSz`

  This parameter is not used by Oracle Database version 8.*X* or later because session caching has been eliminated.

Index

Symbols

Numerics

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

Z