Oracle9*i*

Application Developer's Guide - Fundamentals

Release 1 (9.0.1)

July 2001

Part No. A88876-02

Oracle9*i* Application Developer’s Guide - Fundamentals, Release 1 (9.0.1)

Part No. A88876-02

Primary Author: John Russell

Contributing Authors: T. Brooksfuller, T. Burroughs, M. Cowan, R. Moran, 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, S. Vemuri, W. Wang, D. Wong, A. Yalamanchi, Q. Yu

Graphic Designer: V. Moore

Contents

Send Us Your Comments

Oracle9*i* Application Developer's Guide - Fundamentals, Release 1 (9.0.1)

Part No. A88876-02

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

Application Developer's Guide - Fundamentals describes features of application development for the Oracle9i Database. Information in this guide applies to features that work the same on all platforms, and does not include system-specific information.

This preface contains these topics:

Audience

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

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

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

Duties of an Application Developer

Activities that are typically required of an application developer include:

- Programming in SQL.
- Interfacing to SQL through other languages, such as PL/SQL, Java, or C/C++.
- Setting up interactions and mappings between multiple language environments.
- Compiling and binding applications.
- Designing part or all of a schema.
- Writing code to fit into an existing schema.
- Interfacing with the database administrator to make sure that the schema is implemented and can be recreated, 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 checking and cleanup code.
- Some degree of performance tuning. The database administrator might help here.
- Database administration, if you need to maintain your own development or test system.
- Debugging and interpreting error messages.
- Making your application available over the network, particularly over the Internet or company intranet.

- Building in security features to prevent tampering or unauthorized access to data.
- Designing the class structure, if your application is object-oriented.
- Modelling the entities represented by the database, and their relationships and interactions. You might need to construct diagrams to communicate this information to others.
- Setting up interactions with other software products, such as transaction monitors.

Organization

This document contains:

Part I: Introduction

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

Chapter 1, "Understanding the Oracle Programmatic Environments" outlines the strengths of the languages, development environments, and APIs that Oracle 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, "Managing Schema Objects" explains how to manage objects such as tables, views, numeric sequences, and synonyms. It also discusses performance enhancements to data retrieval through the use of indexes and clusters.

Chapter 3, "Selecting a Datatype" 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 (`ROWID`s).

Chapter 4, "Maintaining Data Integrity Through Constraints" explains how to use constraints to move error-checking logic out of your application and into the database.

Chapter 5, "Selecting an Index Strategy" and Chapter 6, "Speeding Up Index Access with Index-Organized Tables" explain how to speed up queries.

Chapter 7, "How Oracle Processes SQL Statements" explains SQL topics such as commits, cursors, and locking that you can take advantage of in your applications.

Chapter 8, "Coding Dynamic SQL Statements" describes dynamic SQL, compares native dynamic SQL to the DBMS_SQL package, and explains when to use dynamic SQL.

Chapter 9, "Using Procedures and Packages" explains how to store reusable procedures in the database, and how to group procedures into packages.
Chapter 10, "Calling External Procedures" explains how to code the bodies of computationally intensive procedures in languages other than PL/SQL.

Part III: Application Security

Chapter 11, "Database Security Overview for Application Developers" provides background information that you will need before addressing security issues in your applications.

Chapter 12, "Implementing Application Security Policies" explains the major security mechanisms you can use in applications: application context, fine-grained access control, and virtual private database.

Chapter 13, "Proxy Authentication" explains how to tie together authentication done by the web server or application server, with the security mechanisms of the database server.

Chapter 14, "Data Encryption Using DBMS_OBFUSCATION_TOOLKIT" explains how to secure data so that even if an unauthorized person can see the data, they cannot decode it.

Part IV: 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 15, "Using Triggers" explains how to make the database do special processing before, after, or instead of running SQL statements. You can use triggers for things like validating or transforming data, or logging database access.
Chapter 16, "Working With System Events" explains how to retrieve information, such as the user ID and database name, about the event that fires a trigger.

Chapter 17, "Using the Publish-Subscribe Model for Applications" introduces the Oracle model for asynchronous communication, also known as messaging or queuing.

Part IV: Developing Specialized Applications

Chapter 18, "Developing Web Applications with PL/SQL" 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 19, "Porting Non-Oracle Applications to Oracle9i" lists some features and techniques you can use to make applications originally written for another database system run on Oracle9*i*.

Chapter 20, "Working with Transaction Monitors with Oracle XA" describes how to connect Oracle 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 this high-level programming language, which is Oracle Corporation'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 Server.

Oracle Corporation also provides the Pro* series of precompilers, which allow you to embed SQL and PL/SQL in your application programs. If you write 3GL application programs in 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++ Precompiler Programmer's Guide*.

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

For SQL information, see the *Oracle9i SQL Reference* and *Oracle9i Database Administrator's Guide*. For basic Oracle concepts, see *Oracle9i Database Concepts*.

For developing applications that manipulate XML data, see *Oracle9i Application Developer's Guide - XML* and *Oracle9i XML Reference*.

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 *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

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

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

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

`http://technet.oracle.com/membership/index.htm`

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://technet.oracle.com/docs/index.htm`

or to the documentation search engine at:

`http://tahiti.oracle.com/`

This search engine has a number of features that you might find useful, such as searching for examples, looking up SQL and PL/SQL syntax, and formatting large numbers of search results into a "virtual book".

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.	*Oracle9i 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.
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, 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, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`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`*

Convention	Meaning	Example

`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

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

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

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

What’s New in Application Development?

The following sections describe the new features for application development in Oracle9*i*:

- Oracle9i New Features in Application Development

Oracle9i New Features in Application Development

- **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.

  > **See Also:** 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 *Oracle9i Database Migration* 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 7-41

- **Flashback Query**

  Table data can be queried as it existed at a point in time. This lets applications query, compare, or recover past data without involving the DBA and without an expensive recovery operation. The current table data remains available to other applications throughout.

  > **See Also:** "Querying Data at a Point in Time (Flashback Query)" on page 7-43

- **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, and helps the optimizer find common code that can be optimized.

  > **See Also:** "Tip: Referencing the Same Subquery Multiple Times" on page 2-8

- **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, to account for anomalies when clocks shift forward or backward. 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 3-2
  - "Representing Date and Time Data" on page 3-10

- **Better Integration of LOB Datatypes**

  You can operate on LOB types much like other similar types. 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, particularly when converting from `LONG` to LOB types.

  See Also:

  - "Representing Large Data Types" on page 3-22
  - "How Oracle Converts Datatypes" on page 3-33
  - "Representing Character Data" on page 3-6

- **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 byte lengths, or character lengths where the number of bytes is computed for you. 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 3-6

- **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 problems after the operation is complete.

> **See Also:** "Overview of Bulk Binds" on page 9-18

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

  Briefly describe new feature.

  > **See Also:** Chapter 18, "Developing Web Applications with PL/SQL" on page 18-1

- **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 9-22

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

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

  > **See Also:** "Overview of OCI and OCCI" on page 1-26

- **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. These application roles, those enabled by PL/SQL packages, are called secure application roles.

  > **See Also:** Creating Secure Application Roles on page 11-14

- **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.

See Also: Step 2. Create an Application Context on page 12-18

- **Dedicated External Procedure Agents**

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

  See Also:

  - "Loading External C Procedures" on page 10-6
  - "Publishing External Procedures" on page 10-8

Part I

Introduction To Oracle9i Application Development

This part contains the following chapter:

1

Understanding the Oracle Programmatic Environments

This chapter presents brief introductions to these application development systems:

Overview of Developing an Oracle 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. Offloading 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 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**.

The easiest kinds of web or thin-client applications to develop are **stateless**. 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), PL/SQL offers seamless SQL access, tight integration with the Oracle server and 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 applications can access 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 multi-tier Internet or intranet environment without modifying a single line of code.

For more information, see *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, 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 - 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;
```

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 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

PL/SQL supports all the SQL datatypes. Combined with the direct access that SQL provides, these shared datatypes integrate PL/SQL with the Oracle 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 for execution. This can drastically reduce the communication overhead between your application and Oracle.

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 to 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 other tools, 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 to develop scalable applications.

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

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 for your manipulation of large objects.
- DBMS_ROWID is used for employing ROWIDs.
- UTL_RAW is for the RAW facility.
- UTL_REF is for work with REFs.

Built-In Packages for Server Management

- DBMS_SESSION is for session management by DBAs.
- DBMS_SYSTEM is used to set events for debugging.
- DBMS_SPACE and DBMS_SHARED_POOL obtain 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, JDBC, and SQLJ

Oracle can store Java classes and execute them inside the database, as stored procedures and triggers. These classes can manipulate data, but cannot display GUI elements such as AWT or Swing components. Running inside the database allows these Java classes to be called many times and manipulate large amounts of data, without the processing and network overhead that comes with running on the client machine.

Oracle also supports client-side Java standards such as JDBC and SQLJ.

Built-In Java Libraries

Oracle9*i* includes the core JDK libraries such as java.lang, java.io, and so on.

Other libraries include the Java-based CORBA ORB, EJB (Enterprise Java Beans), Java-based XML Parser, and Java Web Server.

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 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 Stored 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 only have 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 non-Oracle databases are also supported.
- Stored procedures and triggers can be reused in different applications as well as different geographic sites.

Overview of Oracle JDBC

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

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.

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:

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. 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 an Oracle database server, thus using native and Java methods.

The OCI driver provides the highest compatibility between the different Oracle versions, from 7 to 9*i*. It also supports all installed Net8 and 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 the Oracle Web Application Server which is a collection of middleware services and tools that supports access from and to applications from browsers and CORBA (Common Object Request Broker Architecture) clients.

JDBC Server Driver

The JDBC server driver is a Type 2 driver that runs inside the database server and therefore reduces 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, Java stored procedures, CORBA objects, and EJB (Enterprise Java Beans). 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.

Extensions of JDBC

Among the Oracle extensions to the JDBC 1.22 standard are:

- 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

Sample Program for the JDBC Thin Driver

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.dnlddriver.OracleDriver());

    // Connect to the local database
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:dnldthin:@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();
  }
}
```

An Oracle 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 1-14) interoperates, allowing dynamic SQL statements in JDBC to be used with static SQL statements in SQLJ. A SQLJ iterator class corresponds to the JDBC result set. For more information on JDBC, see *Oracle9i JDBC Developer's Guide and Reference.*

Overview of Oracle SQLJ

SQLJ is:

- A language specification for embedding static SQL statements in Java source code which has been agreed to by a consortium of database companies, including Oracle, and by Sun, author of Java. The specification has been accepted by ANSI as a software standard.
- A software tool developed by Oracle to the standard, with extensions to the standard to support Oracle features. That tool is the subject of this brief overview.

SQLJ Tool

The Oracle software tool SQLJ has two parts: a translator and a runtime. You execute on any Java VM with a JDBC driver and a SQLJ runtime library.

A SQLJ source file is a Java source file containing embedded static SQL statements. The SQLJ translator is 100% Pure Java and is portable to any standard JDK 1.1 or higher VM.

The Oracle SQLJ implementation runs in three steps:

- Translates SQLJ source to Java code with calls to the SQLJ runtime. 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 using the Java compiler.
- Customizes for the target database. SQLJ generates "profile" files with Oracle-specific customizing.

Oracle9*i* supports SQLJ stored procedures, functions, and triggers which execute in a Java VM integrated with the data server. SQLJ is integrated with Oracle's JDeveloper. Source-level debugging support is available in JDeveloper.

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

```
String name;
#sql  { SELECT ename INTO :name FROM emp WHERE empno=67890 };
System.out.println("Name is " + name + ", employee number = " + empno);
```

Each host variable (or qualified name or complex Java host expression) is preceded by a colon (:). Other SQLJ statements are declarative (declares Java types) and allow you to declare an iterator (a construct related to a database cursor) for queries that retrieve many values:

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

Benefits of SQLJ

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

In particular, Oracle's implementation of 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 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 presuming 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 JDeveloper, a graphical IDE that provides SQLJ translation, Java compilation, profile customizing, and debugging at the source code level, all in one step.
- Provides an SQL Checker module for verification of syntax and semantics at translate-time.
- Includes Oracle type extensions. Datatypes supported are LOBs, ROWIDs, REF CURSORs, VARRAYs, nested tables, user-defined object types, as well as other datatypes such as RAW and NUMBER.

Comparison of SQLJ with JDBC

JDBC provides a complete dynamic SQL interface from Java to databases. SQLJ fills a complementary role.

JDBC provides fine-grained control of the execution of dynamic SQL from Java, while SQLJ provides a higher level static 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 programs allow direct embedding of Java bind expressions within SQL statements. JDBC requires a separate get and / or set call statement for each bind variable and specifies the binding by position number.
- 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 provides simplified rules for calling SQL stored procedures and functions. The JDBC specification requires a generic call to a stored procedure (or function), *fun*, to have the following syntax (we show SQL92 and Oracle escape syntaxes, which are both 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
  ```

  SQLJ provides simplified notations:

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

Here are similarities:

- SQLJ source files can contain JDBC calls. SQLJ and JDBC are interoperable.
- Oracle’s JPublisher tool 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 Example for Object Types

Here is a simple use of user-defined objects and object refs taken from *Oracle9i SQLJ Developer's Guide and Reference,* where more information on SQLJ is available:

The following items are created using the SQL script below:

- Two object types, `PERSON` and `ADDRESS`
- A typed table for `PERSON` objects
- An `EMPLOYEE` table that includes an `ADDRESS` column and two columns of `PERSON` references

```
SET ECHO ON;
/
/*** Clean up in preparation ***/
DROP TABLE EMPLOYEES
/
DROP TABLE PERSONS
/
DROP TYPE PERSON FORCE
/
DROP TYPE ADDRESS FORCE
/
/*** Create address UDT ***/
CREATE TYPE address AS OBJECT
(
  street        VARCHAR(60),
  city          VARCHAR(30),
  state         CHAR(2),
  zip_code      CHAR(5)
)
/
/*** Create person UDT containing an embedded address UDT ***/
CREATE TYPE person AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    address
)
/
/*** Create a typed table for person objects ***/
CREATE TABLE persons OF person
/
/*** Create a relational table with two columns that are REFs
     to person objects, as well as a column which is an Address ADT. ***/
```

```
CREATE TABLE  employees
(
  empnumber            INTEGER PRIMARY KEY,
  person_data     REF  person,
  manager         REF  person,
  office_addr          address,
  salary               NUMBER
)
/*** Insert some data--2 objects into the persons typed table ***/
INSERT INTO persons VALUES (
            person('Wolfgang Amadeus Mozart', 123456,
               address('Am Berg 100', 'Salzburg', 'AT','10424')))
/
INSERT INTO persons VALUES (
            person('Ludwig van Beethoven', 234567,
               address('Rheinallee', 'Bonn', 'DE', '69234')))
/
/** Put a row in the employees table **/
INSERT INTO employees (empnumber, office_addr, salary) VALUES (
            1001,
            address('500 Oracle Parkway', 'Redwood Shores', 'CA', '94065'),
            50000)
/
/** Set the manager and person REFs for the employee **/
UPDATE employees
   SET manager =
       (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/
UPDATE employees
   SET person_data =
       (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')
/
COMMIT
/
QUIT
```

Next, JPublisher is used to generate the `Address` class for mapping to Oracle `ADDRESS` objects. We omit the details.

The following SQLJ sample declares and sets an input host variable of Java type `Address` to update an `ADDRESS` object in a column of the `employees` table. Both before and after the update, the office address is selected into an output host variable of type `Address` and printed for verification.

```
...
```

```
// Updating an object

static void updateObject()
{

   Address addr;
   Address new_addr;
   int empno = 1001;

   try {
      #sql {
         SELECT office_addr
         INTO :addr
         FROM employees
         WHERE empnumber = :empno };
      System.out.println("Current office address of employee 1001:");

      printAddressDetails(addr);

      /* Now update the street of address */

      String street ="100 Oracle Parkway";
      addr.setStreet(street);

      /* Put updated object back into the database */

      try {
         #sql {
            UPDATE employees
            SET office_addr = :addr
            WHERE empnumber = :empno };
         System.out.println
            ("Updated employee 1001 to new address at Oracle Parkway.");

         /* Select new address to verify update */

         try {
            #sql {
               SELECT office_addr
               INTO :new_addr
               FROM employees
               WHERE empnumber = :empno };

            System.out.println("New office address of employee 1001:");
            printAddressDetails(new_addr);
         } catch (SQLException exn) {
         System.out.println("Verification SELECT failed with "+exn); }

      } catch (SQLException exn) {
      System.out.println("UPDATE failed with "+exn); }

   } catch (SQLException exn) {
   System.out.println("SELECT failed with "+exn); }
}
...
```

Note the use of the `setStreet()` accessor method of the `Address` instance. Remember that JPublisher provides such accessor methods for all attributes in any custom Java class that it produces.

SQLJ Stored Procedures in the Server

SQLJ applications can be stored and run in the server. You have the option of translating, compiling, and customizing SQLJ source on a client and loading the generated classes and resources into the server with the `loadjava` utility, typically using a Java archive (`.jar`) file.

Or, you have a second option of loading SQLJ source code into the server, also using `loadjava`, and having it translated and compiled by the server's embedded translator.

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) version. 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 above 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 inline 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 multi-threaded programs if your platform supports a threads package. Concurrent connections are supported in either single-threaded or multi-threaded applications.
- You can improve performance by embedding PL/SQL blocks. These blocks can call functions or procedures written by you or provided in Oracle packages, in either Java or PL/SQL.
- 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++ 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 VARRAYs, are supported with a set of SQL statements that allow a high degree of control over data.
- Large Objects (LOBs, CLOBs, NCLOBs, and external files known as BFILEs) 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.

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 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) version. 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 above 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 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.

Enter the options either in a configuration file, on the command line, or inline 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. For complete details, see the *Pro*COBOL Precompiler Programmer's Guide.*

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 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.

Overview of OCI and OCCI

The Oracle Call Interface (OCI) and Oracle C++ Interface (OCCI) are application programming interfaces (APIs) that allow you to create applications that use a third-generation language's native procedures or function calls to access an Oracle database server 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 multi-tier 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 an Oracle 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.

For more information about the OCI and OCCI calls, see *Oracle Call Interface Programmer's Guide, Oracle C++ Call Interface Programmer's Guide, Oracle9i Application Developer's Guide - Advanced Queuing, Oracle9i Globalization and National Language Support Guide,* and *Oracle9i Data Cartridge Developer's Guide.*

Advantages of OCI

OCI provides significant advantages over other methods of accessing an 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 roundtrips.
- Optimization for queries using transparent prefetch buffers to reduce roundtrips.
- Thread safety so you do not have to use mutual exclusive locks (mutex) on OCI handles.
- The server connection in non-blocking 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 server
- 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, such as 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, or 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 an 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 an Oracle database server. For example, an OCI program can run a query against an 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
```

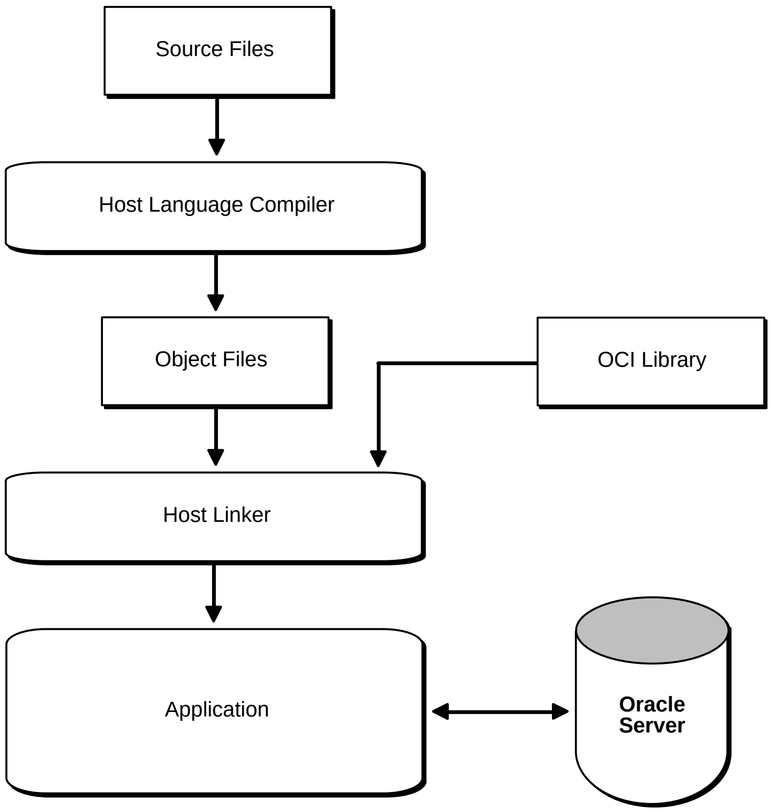
In the above SQL statement,`:empnumber` is a placeholder for a value that will be supplied by the application.

You can also 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 an Oracle database server.

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

Overview of Oracle Objects for OLE (OO4O)

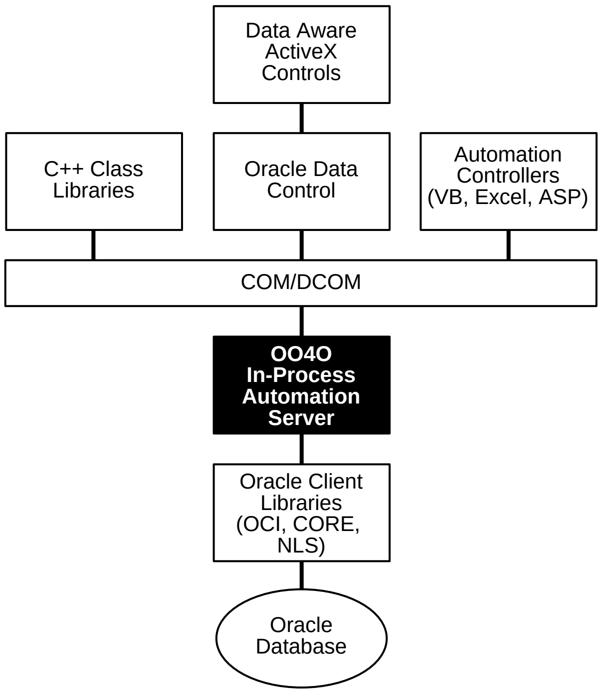
Oracle Objects for OLE (OO4O) is a product designed to allow easy access to data stored in Oracle databases with any programming or scripting language that supports the Microsoft COM Automation and ActiveX technology. This includes Visual Basic, Visual C++, Visual Basic For Applications (VBA), IIS Active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

OO4O 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


Data Aware ActiveX Controls
C++ Class Libraries
Oracle Data Control
Automation Controllers (VB, Excel, ASP)
COM/DCOM
OO4O In-Process Automation Server
Oracle Client Libraries (OCI, CORE, NLS)
Oracle Database

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

OO4O Automation Server

The OO4O Automation Server is a set of COM Automation objects for connecting to Oracle database servers, 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 servers.

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

OO4O provides key features for accessing Oracle databases 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 multi-tiered application server environments such as web server applications in Microsoft Internet Information Server (IIS) or Microsoft Transaction Server (MTS).

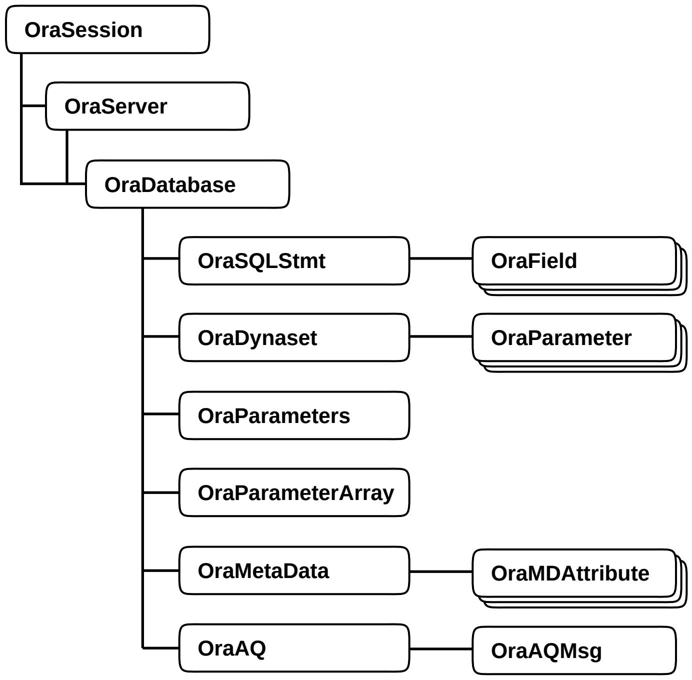
Features include:

- 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 1-36.
- 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 Relation".

Figure 1–3 Objects and Their Relation



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 level 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 an Oracle database server.

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 an n-tier distributed environments.

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

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 older releases, 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.

As described in the OraServer interface description, 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. These 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 OraDynaset's semantics. An OraDynaset 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 implementers 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

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 (0 to Count-1) results in the return of a NULL OraMDAttribute object.

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 objects' descriptions), by using the parameter’s 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 parameter has an identifying name and an associated value.

OraSQLStmt

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

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.

SQLStmt

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

```
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’s Advanced Queuing (AQ) Feature. It makes AQ accessible from popular COM-based development environments such as Visual Basic. For a detailed description of Oracle AQ, please refer to *Oracle9i Application Developer’s Guide - Advanced Queuing*.

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 AQ, please refer to *Oracle9i Application Developer's Guide - Advanced Queuing.*

OraAQAgent

The OraAQAgent object represents a message recipient and is only valid for queues which allow multiple consumers.

The OraAQAgent object represents a message recipient and is only valid for queues which allow multiple consumers.

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:

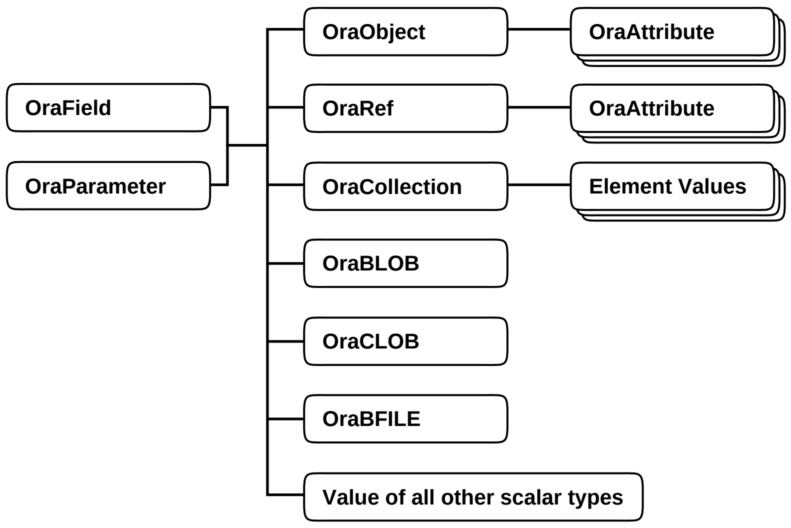
```
Set agent = qMsg.AddRecipient(name, address, protocol).
```

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 an Oracle database server. 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. These interfaces may be obtained from:

Figure 1–4 Supported Oracle Datatypes



OraBLOB and OraCLOB

The OraBlob and OraClob interfaces in OO4O provide methods for performing operations on large objects in the database of data types BLOB, CLOB, and NCLOB. In this help file BLOB, CLOB, and NCLOB datatypes are also referred to 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, then the lock is obtained by invoking the Edit method.

OraBFILE

The OraBFile interface in OO4O provides methods for performing operations on large objects BFILE data type in the database.

The BFILEs are large binary data objects stored in operating system files (external) outside of the database tablespaces.

The Oracle Data Control

The Oracle Data Control (ODC) is an ActiveX Control that is designed to simplify the exchange of data between an 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 an agent to handle the flow of information from an 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.

The 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 the 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.

The Oracle Objects for OLE C++ Class Library

The 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 that is provided with the OO4O product:

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

To view examples of the use of Oracle Object for OLE, see the samples located in the ORACLE_HOME\OO4O directory of the Oracle installation. Additional OO4O examples can be found in the following Oracle publications, including:

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*
- *Oracle9i Application Developer's Guide - Advanced Queuing*
- *Oracle9i Supplied 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 and then return a batch of errors.

- 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, as mentioned in the overviews for those languages.

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 versus PL/SQL

Both Java and PL/SQL can be used to build applications in the database and will have future performance improvements. Here are 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 the extension to SQL and uses the same datatypes. PL/SQL has 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 when using 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. However, PL/SQL packages can also 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

Managing Schema Objects

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

See Also:

- Indexes and clusters — Chapter 5, "Selecting an Index Strategy"
- Procedures, functions, and packages — Chapter 9, "Using Procedures and Packages"
- Object types — *Oracle9i Application Developer's Guide - Object-Relational Features*
- Dependency information — Chapter 9, "Using Procedures and Packages"
- If you use symmetric replication, then see *Oracle9i Replication* for information on managing schema objects, such as snapshots.

Managing Tables

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

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

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

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

The following sections explain how to create, alter, and drop tables. Some simple guidelines to follow when managing tables in your database are included.

> **See Also:** The *Oracle9i Database Administrator's Guide* has more suggestions. You should also refer to a text on relational database or table design.

Designing Tables

Consider the following guidelines when designing your tables:

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

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

See Also:

- *Oracle9i Database Administrator's Guide* for guidelines about tables and general guidelines for managing the space used by schema objects.
- Chapter 3, "Selecting a Datatype" for information about datatypes.
- Chapter 4, "Maintaining Data Integrity Through Constraints" for guidelines on integrity constraints.

Creating Tables

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

```
CREATE TABLE Emp_tab (
   Empno      NUMBER(5) PRIMARY KEY,
   Ename      VARCHAR2(15) NOT NULL,
   Job        VARCHAR2(10),
   Mgr        NUMBER(5),
   Hiredate   DATE DEFAULT (sysdate),
   Sal        NUMBER(7,2),
   Comm       NUMBER(7,2),
   Deptno     NUMBER(3) NOT NULL,
              CONSTRAINT dept_afkey REFERENCES Dept_tab(Deptno))
   PCTFREE 10
   PCTUSED 40
   TABLESPACE users
   STORAGE (  INITIAL 50K
              NEXT 50K
              MAXEXTENTS 10
              PCTINCREASE 25 );
```

Managing Temporary Tables

Oracle8*i* provides a special kind of table to hold temporary data. You specify whether the data is specific to a session or to a transaction. When the session or transaction finishes, the rows that it inserted are deleted. Multiple sessions or transactions can use the same temporary table, and each session or transaction only sees the rows that it created.

Temporary tables are useful any time you want to buffer a result set or construct a result set by running multiple DML operations. Here are a few specific examples:

- A Web-based airlines reservations application allows a customer to create several optional itineraries. Each itinerary is represented by a row in a temporary table. The application updates the rows to reflect changes in the itineraries. When the customer decides which itinerary you want to use, the application moves the row for that itinerary to a persistent table.

  During the session, the itinerary data is private. At the end of the session, the optional itineraries are dropped.

- Several sales agents for a large bookseller use a single temporary table concurrently while taking customer orders over the phone. To enter and modify customer orders, each agent accesses the table in a session that is unavailable to the other agents. When the agent closes a session, the data from that session is automatically dropped, but the table structure persists for the other agents to use.

- An administrator uses temporary tables to improve performance when running an otherwise complex and expensive query. To do this, the administrator caches the values from a more complex query in temporary tables, then runs SQL statements, such as joins, against those temporary tables. For a thorough explanation of how this can be done, see "Example 2: Using Temporary Tables to Improve Performance" on page 2-7.

Creating Temporary Tables

You create a temporary table by using special ANSI keywords. You specify the data as **session-specific** by using the ON COMMIT PRESERVE ROWS keywords. You specify the data as **transaction-specific** by using the ON COMMIT DELETE ROWS keywords.

Example 2–1 Creating a Session-Specific Temporary Table

```
CREATE GLOBAL TEMPORARY TABLE ...
     [ON COMMIT PRESERVE ROWS ]
```

Example 2–2 Creating a Transaction-Specific Temporary Table

```
CREATE GLOBAL TEMPORARY TABLE ...
     [ON COMMIT DELETE ROWS ]
```

Using Temporary Tables

You can create indexes on temporary tables as you would on permanent tables.

For a *session*-specific temporary table, a session gets bound to the temporary table with the first insert in the table in the session. This binding goes away at the end of the session or by issuing a TRUNCATE of the table in the session.

For a *transaction*-specific temporary table, a session gets bound to the temporary table with the first insert in the table in the transaction. The binding goes away at the end of the transaction.

DDL operations (except TRUNCATE) are allowed on an existing temporary table only if no session is currently bound to that temporary table.

Unlike permanent tables, temporary tables and their indexes do not automatically allocate a segment when they are created. Instead, segments are allocated when the first INSERT (or CREATE TABLE AS SELECT) is performed. This means that if a SELECT, UPDATE, or DELETE is performed before the first INSERT, the table appears to be empty.

Temporary segments are deallocated at the end of the transaction for transaction-specific temporary tables and at the end of the session for session-specific temporary tables.

If you rollback a transaction, the data you entered is lost, although the table definition persists.

You cannot create a table that is simultaneously both transaction- and session-specific.

A transaction-specific temporary table allows only one transaction at a time. If there are several autonomous transactions in a single transaction scope, each autonomous transaction can use the table only as soon as the previous one commits.

Because the data in a temporary table is, by definition, temporary, backup and recovery of a temporary table's data is not available in the event of a system failure. To prepare for such a failure, you should develop alternative methods for preserving temporary table data.

Examples: Using Temporary Tables

Example 1: A Session-Specific Temporary Table

The following statement creates a session-specific temporary table, `FLIGHT_SCHEDULE`, for use in an automated airline reservation scheduling system. Each client has its own session and can store temporary schedules. The temporary schedules are deleted at the end of the session.

```
CREATE GLOBAL TEMPORARY TABLE flight_schedule (
   startdate DATE,
   enddate DATE,
   cost NUMBER)
ON COMMIT PRESERVE ROWS;
```

Example 2: Using Temporary Tables to Improve Performance

You can use temporary tables to improve performance when you run complex queries. Running multiple such queries is relatively slow because the tables are accessed multiple times for each returned row. It is faster to cache the values from a complex query in a temporary table, then run the queries against the temporary table.

For example, even with a view like this defined to simplify further queries, the queries against the view may be slow because the contents of the view are recalculated each time:

```
CREATE OR REPLACE VIEW Profile_values_view AS
SELECT d.Profile_option_name, d.Profile_option_id, Profile_option_value,
       u.User_name, Level_id, Level_code
  FROM Profile_definitions d, Profile_values v, Profile_users u
 WHERE d.Profile_option_id = v.Profile_option_id
   AND ((Level_code = 'USER' AND Level_id = U.User_id) OR
        (Level_code = 'DEPARTMENT' AND Level_id = U.Department_id) OR
        (Level_code = 'SITE'))
   AND NOT EXISTS (SELECT 1 FROM PROFILE_VALUES P
                    WHERE P.PROFILE_OPTION_ID = V.PROFILE_OPTION_ID
                      AND ((Level_code = 'USER' AND
                            level_id = u.User_id) OR
                           (Level_code = 'DEPARTMENT' AND
                            level_id = u.Department_id) OR
                           (Level_code = 'SITE'))
                      AND INSTR('USERDEPARTMENTSITE', v.Level_code) >
                          INSTR('USERDEPARTMENTSITE', p.Level_code));
```

A temporary table allows us to run the computation once, and cache the result in later SQL queries and joins:

```
CREATE GLOBAL TEMPORARY TABLE Profile_values_temp
         (
             Profile_option_name   VARCHAR(60)   NOT NULL,
             Profile_option_id     NUMBER(4)     NOT NULL,
             Profile_option_value  VARCHAR2(20)  NOT NULL,
             Level_code            VARCHAR2(10)          ,
             Level_id              NUMBER(4)             ,
             CONSTRAINT Profile_values_temp_pk
                PRIMARY KEY (Profile_option_id)
         ) ON COMMIT PRESERVE ROWS ORGANIZATION INDEX;

INSERT INTO Profile_values_temp
       (Profile_option_name, Profile_option_id, Profile_option_value,
        Level_code, Level_id)
SELECT Profile_option_name, Profile_option_id, Profile_option_value,
        Level_code, Level_id
  FROM Profile_values_view;
COMMIT;
```

Now the temporary table can be used to speed up queries, and the results cached in the temporary table are freed automatically by the database when the session ends.

Tip: Referencing the Same Subquery Multiple Times

In complex queries that process the same subquery multiple times, you might be tempted to store the subquery results in a temporary table and perform additional queries against the temporary table. The `WITH` clause lets you factor out the subquery, give it a name, then reference that name multiple times within the original complex query.

This technique lets the optimizer choose how to deal with the subquery results -- whether to create a temporary table or inline it as a view.

For example, the following query joins two tables and computes the aggregate SUM(SAL) more than once. The bold text represents the parts of the query that are repeated.

```
SELECT dname, SUM(sal) AS dept_total
  FROM emp, dept
  WHERE emp.deptno = dept.deptno
  GROUP BY dname HAVING
    SUM(sal) >
   (
      SELECT SUM(sal) * 1/3
        FROM emp, dept
        WHERE emp.deptno = dept.deptno
   )
   ORDER BY SUM(sal) DESC;
```

You can improve the query by doing the subquery once, and referencing it at the appropriate points in the main query. The bold text represents the common parts of the subquery, and the places where the subquery is referenced.

```
WITH
summary AS
(
  SELECT dname, SUM(sal) AS dept_total
    FROM emp, dept
    WHERE emp.deptno = dept.deptno
    GROUP BY dname
)
SELECT dname, dept_total
  FROM summary
  WHERE dept_total >
(
  SELECT SUM(dept_total) * 1/3
    FROM summary
)
ORDER BY dept_total DESC;
```

See Also:

- *Oracle9i SQL Reference* for the full syntax of the WITH clause.
- *Oracle9i Database Performance Guide and Reference* for techniques to use this feature to improve performance.

Managing Views

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

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

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

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

Creating Views

Use the SQL command CREATE VIEW to create a view. For example, the following statement creates a view on a subset of data in the EMP_TAB table:

```
CREATE VIEW Sales_staff AS
    SELECT Empno, Ename, Deptno
    FROM Emp_tab
    WHERE Deptno = 10
    WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

The object names are resolved when the view is created or when the program containing the SQL is compiled, relative to the schema of the view owner.

You can define views with any query that references tables, snapshots, or other views.

The query that defines the SALES_STAFF view references only rows in department 10. Furthermore, WITH CHECK OPTION creates the view with the constraint that INSERT and UPDATE statements issued against the view are not allowed to create or result in rows that the view cannot select.

Considering the example above, the following INSERT statement successfully inserts a row into the EMP_TAB table through the SALES_STAFF view:

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

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

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

The following statement creates a view that joins data from the Emp_tab and Dept_tab tables:

```
CREATE VIEW Division1_staff AS
    SELECT Ename, Empno, Job, Dname
    FROM Emp_tab, Dept_tab
    WHERE Emp_tab.Deptno IN (10, 30)
    AND Emp_tab.Deptno = Dept_tab.Deptno;
```

The Division1_staff view is defined by a query that joins information from the Emp_tab and Dept_tab tables. The WITH CHECK OPTION is not specified in the CREATE VIEW statement because rows cannot be inserted into or updated in a view defined with a query that contains a join that uses the WITH CHECK OPTION.

Expansion of Defining Queries at View Creation Time

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

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

```
CREATE VIEW Dept_view AS SELECT * FROM scott.Dept_tab;
```

Oracle stores the defining query of the Dept_view view as

```
SELECT "DEPTNO", "DNAME", "LOC" FROM scott.Dept_tab;
```

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

Creating Views with Errors

A view can be created even if the defining query of the view cannot be executed, as long as the CREATE VIEW command has no syntax errors. We call such a view a **view with errors**. For example, if a view refers to a non-existent table or an invalid column of an existing table, or if the owner of the view does not have the required privileges, then the view can still be created and entered into the data dictionary.

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

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

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

Privileges Required to Create a View

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

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

Replacing Views

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

- A view can be dropped and then re-created. When a view is dropped, all grants of corresponding view privileges are revoked from roles and users. After the view is re-created, necessary privileges must be regranted.
- A view can be replaced by redefining it with a `CREATE VIEW` statement that contains the `OR REPLACE` option. This option replaces the current definition of a view, but preserves the present security authorizations.

  For example, assume that you create the `SALES_STAFF` view, as given in a previous example. You also grant several object privileges to roles and other users. However, now you realize that you must redefine the `SALES_STAFF` view to correct the department number specified in the `WHERE` clause of the defining query, because it should have been 30. To preserve the grants of object privileges that you have made, you can replace the current version of the `SALES_STAFF` view with the following statement:

  ```
  CREATE OR REPLACE VIEW Sales_staff AS
      SELECT Empno, Ename, Deptno
      FROM Emp_tab
      WHERE Deptno = 30
      WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
  ```

Replacing a view has the following effects:

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

Privileges Required to Replace a View

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

Using Views

Views can be queried in the same manner as tables. For example, to query the `Division1_staff` view, enter a valid `SELECT` statement that references the view:

```
SELECT * FROM Division1_staff;
```

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

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

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

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

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

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

```
CREATE VIEW Sales_staff AS
    SELECT Empno, Ename
    FROM Emp_tab
    WHERE Deptno = 10
    WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

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

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

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

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

Privileges Required to Use a View

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

Dropping Views

Use the SQL command DROP VIEW to drop a view. For example:

```
DROP VIEW Sales_staff;
```

Privileges Required to Drop a View

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

Modifying a Join View

Oracle allows you, with some restrictions, to modify views that involve joins. Consider the following simple view:

```
CREATE VIEW Emp_view AS
    SELECT Ename, Empno, deptno FROM Emp_tab;
```

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

```
UPDATE Emp_view SET Ename = 'CAESAR' WHERE Empno = 7839;
```

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

However, if you create a view that involves a join operation, such as:

```
CREATE VIEW Emp_dept_view AS
  SELECT e.Empno, e.Ename, e.Deptno, e.Sal, d.Dname, d.Loc
    FROM Emp_tab e, Dept_tab d    /* JOIN operation */
     WHERE e.Deptno = d.Deptno
       AND d.Loc IN ('DALLAS', 'NEW YORK', 'BOSTON');
```

then there are restrictions on modifying either the `EMP_TAB` or the `DEPT_TAB` base table through this view, for example, using a statement such as:

```
UPDATE Emp_dept_view SET Ename = 'JOHNSON'
    WHERE Ename = 'SMITH';
```

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

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

A further restriction on which join views are modifiable is that if a view is a join on other nested views, then the other nested views must be mergeable into the top level view.

See Also:

Oracle9i Database Concepts for more information about mergeable views.

"Modifying Complex Views (INSTEAD OF Triggers)" on page 15-7 for a way to simulate updating a join view by writing a customized trigger.

Scenario for Modifying a Join View

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

```
CREATE TABLE Dept_tab (
        Deptno   NUMBER(4) PRIMARY KEY,
        Dname    VARCHAR2(14),
        Loc      VARCHAR2(13));

CREATE TABLE Emp_tab (
        Empno    NUMBER(4) PRIMARY KEY,
        Ename    VARCHAR2(10),
        Job      varchar2(9),
        Mgr      NUMBER(4),
        Hiredate DATE,
        Sal      NUMBER(7,2),
        Comm     NUMBER(7,2),
        Deptno   NUMBER(2),
FOREIGN KEY (Deptno) REFERENCES Dept_tab(Deptno));
```

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

About Key-Preserved Tables

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

Note:

- It is not necessary that the key or keys of a table be selected for it to be key preserved. It is sufficient that if the key or keys were selected, then they would also be key(s) of the result of the join.
- The key-preserving property of a table does not depend on the actual data in the table. It is, rather, a property of its schema and not of the data in the table. For example, if in the EMP_TAB table there was at most one employee in each department, then DEPT_TAB.DEPTNO would be unique in the result of a join of EMP_TAB and DEPT_TAB, but DEPT_TAB would still not be a key-preserved table.

If you SELECT all rows from EMP_DEPT_VIEW defined in the "Modifying a Join View" section, then the results are:

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

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

Rule for DML Statements on Join Views

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

Updating a Join View

The following example shows an UPDATE statement that successfully modifies the EMP_DEPT_VIEW view:

```
UPDATE Emp_dept_view
  SET Sal = Sal * 1.10
    WHERE Deptno = 10;
```

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

```
UPDATE Emp_dept_view
  SET Loc = 'BOSTON'
    WHERE Ename = 'SMITH';
```

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

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

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

```
UPDATE Emp_dept_view
    SET Deptno = 10
        WHERE Ename = 'SMITH';
```

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

Deleting from a Join View

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

The following DELETE statement works on the EMP_DEPT view:

```
DELETE FROM Emp_dept_view
    WHERE Ename = 'SMITH';
```

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

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

```
CREATE VIEW emp_emp AS
```

```
    SELECT e1.Ename, e2.Empno, e1.Deptno
        FROM Emp_tab e1, Emp_tab e2
        WHERE e1.Empno = e2.Empno;
        WHERE e1.Empno = e2.Empno;
```

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

```
CREATE VIEW Emp_mgr AS
    SELECT e1.Ename, e2.Ename Mname
       FROM Emp_tab e1, Emp_tab e2
            WHERE e1.mgr = e2.Empno
            WITH CHECK OPTION;
```

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

Inserting into a Join View

The following INSERT statement on the EMP_DEPT view succeeds, because only one key-preserved base table is being modified (EMP_TAB), and 40 is a valid DEPTNO in the DEPT_TAB table (thus satisfying the FOREIGN KEY integrity constraint on the EMP_TAB table).

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
    VALUES ('KURODA', 9010, 40);
```

The following INSERT statement fails for the same reason: This UPDATE on the base EMP_TAB table would fail: the FOREIGN KEY integrity constraint on the EMP_TAB table is violated.

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
    VALUES ('KURODA', 9010, 77);
```

The following INSERT statement fails with an ORA-01776 error ("cannot modify more than one base table through a view").

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
    VALUES (9010, 'KURODA', 'BOSTON');
```

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

Using the UPDATABLE_COLUMNS Views

Three views you can use for modifying join views are shown in Table 2–1.

Table 2–1 UPDATABLE_COLUMNS Views

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

Outer Joins

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

```
CREATE VIEW Emp_dept_oj1 AS
    SELECT Empno, Ename, e.Deptno, Dname, Loc
    FROM Emp_tab e, Dept_tab d
    WHERE e.Deptno = d.Deptno (+);
```

The statement:

```
SELECT * FROM Emp_dept_oj1;
```

Results in:

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

```
14 rows selected.
```

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

The following view also contains an outer join:

```
CREATE VIEW Emp_dept_oj2 AS
SELECT e.Empno, e.Ename, e.Deptno, d.Dname, d.Loc
FROM Emp_tab e, Dept_tab d
WHERE e.Deptno (+) = d.Deptno;
```

The statement:

```
SELECT * FROM Emp_dept_oj2;
```

Results in:

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

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

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

```
SELECT Col1, Col2, ... FROM T;
```

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

Consider the following set of views:

```
CREATE VIEW Emp_v AS
    SELECT Empno, Ename, Deptno
        FROM Emp_tab;
CREATE VIEW Emp_dept_oj1 AS
    SELECT e.*, Loc, d.Dname
        FROM Emp_v e, Dept_tab d
            WHERE e.Deptno = d.Deptno (+);
```

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

```
CREATE VIEW Emp_v_2 AS
    SELECT Empno, Ename, Deptno
        FROM Emp_tab
            WHERE Sal > 1000;
```

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

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

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

This might return:

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

Managing Sequences

The sequence generator generates sequential numbers, which can help to generate unique primary keys automatically, and to coordinate keys across multiple rows or tables.

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

The following sections explain how to create, alter, use, and drop sequences using SQL commands.

Creating Sequences

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

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

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

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

See Also: For additional implications for caching sequence numbers when using Oracle Real Application Clusters, see *Oracle9i Real Application Clusters Deployment and Performance.*

General information about caching sequence numbers is included in "Caching Sequence Numbers" on page 2-28.

Privileges Required to Create a Sequence

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

Altering Sequences

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

Use the SQL command ALTER SEQUENCE to alter a sequence. For example:

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

Privileges Required to Alter a Sequence

To alter a sequence, your schema must contain the sequence, or you must have the ALTER ANY SEQUENCE system privilege.

Using Sequences

Once a sequence is defined, it can be accessed and incremented by multiple users with no waiting. Oracle does not wait for a transaction that has incremented a sequence to complete before that sequence can be incremented again.

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

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

Referencing a Sequence

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

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

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

```
INSERT INTO Orders_tab (Orderno, Custno)
    VALUES (Order_seq.NEXTVAL, 1032);
```

Or, the sequence number can be referenced in the SET clause of an UPDATE statement. For example:

```
UPDATE Orders_tab
    SET Orderno = Order_seq.NEXTVAL
    WHERE Orderno = 10112;
```

The sequence number can also be referenced outermost SELECT of a query or subquery. For example:

```
SELECT Order_seq.NEXTVAL FROM dual;
```

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

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

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

```
INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
    VALUES (Order_seq.CURRVAL, 20321, 3);

INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
    VALUES (Order_seq.CURRVAL, 29374, 1);
```

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

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

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

CURRVAL and NEXTVAL cannot be used in these places:

- A subquery
- A view's query or snapshot's query
- A SELECT statement with the DISTINCT operator
- A SELECT statement with a GROUP BY or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator

- The `WHERE` clause of a `SELECT` statement
- `DEFAULT` value of a column in a `CREATE TABLE` or `ALTER TABLE` statement
- The condition of a `CHECK` constraint

Caching Sequence Numbers

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

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

Follow these guidelines for fast access to all sequence numbers:

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

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

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

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

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

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

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

```
CREATE SEQUENCE Seq2
    CACHE 50;
```

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

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

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

```
CREATE SEQUENCE Seq3
    NOCACHE;
```

Privileges Required to Use a Sequence

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

Dropping Sequences

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

```
DROP SEQUENCE Order_seq;
```

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

Privileges Required to Drop a Sequence

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

Managing Synonyms

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

Creating Synonyms

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

```
CREATE PUBLIC SYNONYM Public_emp FOR jward.Emp_tab;
```

Privileges Required to Create a Synonym

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

Using Synonyms

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

```
INSERT INTO Emp_tab (Empno, Ename, Job)
    VALUES (Emp_sequence.NEXTVAL, 'SMITH', 'CLERK');
```

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

```
EXECUTE Fire_emp(7344);
```

You can also use synonyms for GRANT and REVOKE statements, but not with other DML statements.

Privileges Required to Use a Synonym

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

Dropping Synonyms

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

```
DROP SYNONYM Emp_tab;
```

The following statement drops the public synonym named PUBLIC_EMP:

```
DROP PUBLIC SYNONYM Public_emp;
```

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

Privileges Required to Drop a Synonym

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

Creating Multiple Tables and Views in One Operation

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

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

```
CREATE SCHEMA AUTHORIZATION scott
    CREATE VIEW Sales_staff AS
        SELECT Empno, Ename, Sal, Comm
        FROM Emp_tab
        WHERE Deptno = 30  WITH CHECK OPTION CONSTRAINT
               Sales_staff_cnst

CREATE TABLE Dept_tab (
    Deptno      NUMBER(3) PRIMARY KEY,
    Dname       VARCHAR2(15),
    Loc         VARCHAR2(25))
CREATE TABLE Emp_tab (
    Empno       NUMBER(5) PRIMARY KEY,
    Ename       VARCHAR2(15) NOT NULL,
    Job         VARCHAR2(10),
    Mgr         NUMBER(5),
    Hiredate    DATE DEFAULT (sysdate),
    Sal         NUMBER(7,2),
    Comm        NUMBER(7,2),
    Deptno      NUMBER(3) NOT NULL
        CONSTRAINT Dept_fkey REFERENCES Dept_tab(Deptno))

GRANT SELECT ON Sales_staff TO human_resources;
```

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

Privileges Required to Create Multiple Schema Objects

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

Naming Schema Objects

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

In a distributed database system, each database should have a unique global name. The global name is composed of the database name and the network domain that contains the database. Each schema object in the database then has a global object name consisting of the schema object name and the global database name.

Because Oracle ensures that the schema object name is unique within a database, you can ensure that it is unique across all databases by assigning unique global database names. You should coordinate with your database administrator on this task, because it is usually the DBA who is responsible for assigning database names.

Rules for Name Resolution in SQL Statements

An object name takes the following form:

[*schema*.]*name*[@*database*]

Some examples include:

```
Emp_tab
Scott.Emp_tab
Scott.Emp_tab@Personnel
```

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

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

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

```
DROP CLUSTER Test
```

Oracle looks up `TEST` in the cluster namespace.

Rather than supplying an object name directly, you can also refer to an object using a synonym. A private synonym name has the same syntax as an ordinary object name. A public synonym is implicitly in the `PUBLIC` schema, but users cannot explicitly qualify a synonym with the schema `PUBLIC`.

Synonyms can only be used to reference objects in the same namespace as tables. Due to the possibility of synonyms, the following rules are used to resolve a name in a context that requires an object in the table namespace:

1. Look up the name in the table namespace.
2. If the name resolves to an object that is not a synonym, then no further work is necessary.
3. If the name resolves to a private synonym, then replace the name with the definition of the synonym and return to step 1.
4. If the name was originally qualified with a schema, then return an error; otherwise, check if the name is a public synonym.
5. If the name is not a public synonym, return an error; otherwise, then replace the name with the definition of the public synonym and return to step 1.

When global object names are used in a distributed database (either explicitly or indirectly within a synonym), the local Oracle session resolves the reference as is locally required (for example, resolving a synonym to a remote table's global object name). After the partially resolved statement is shipped to the remote database, the remote Oracle session completes the resolution of the object as above.

> **See Also:** See *Oracle9i Database Concepts* for more information about name resolution in a distributed database.

Renaming Schema Objects

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

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

If you use the RENAME command to rename a table, view, sequence, or a private synonym of a table, view, or sequence, then grants made for the object are carried forward for the new name, and the next statement renames the SALES_STAFF view:

```
RENAME Sales_staff TO Dept_30;
```

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

Renaming a schema object has the following effects:

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

Privileges Required to Rename an Object

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

Switching to a Different Schema

The following statement sets the current schema of the session to the schema name given in the statement.

```
ALTER SESSION SET CURRENT_SCHEMA = <schema name>
```

Subsequent SQL statements use this schema name for the schema qualifier when the qualifier is missing. Note that the session still has only the privileges of the current user and does not acquire any extra privileges by the above ALTER SESSION statement.

For example:

```
CONNECT scott/tiger
ALTER SESSION SET CURRENT_SCHEMA = joe;
SELECT * FROM emp_tab;
```

Since emp_tab is not schema-qualified, the table name is resolved under schema joe. But if scott does not have select privilege on table joe.emp_tab, then scott cannot execute the SELECT statement.

Listing Information about Schema Objects

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

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

Example 1: Listing Different Schema Objects by Type The following query lists all of the objects owned by the user issuing the query:

```
SELECT Object_name, Object_type FROM User_objects;
```

The query above might return results similar to the following:

```
OBJECT_NAME              OBJECT_TYPE
------------------------ -------------------
EMP_DEPT                 CLUSTER
EMP_TAB                  TABLE
```

```
DEPT_TAB                  TABLE
EMP_DEPT_INDEX            INDEX
PUBLIC_EMP                SYNONYM
EMP_MGR                   VIEW
```

Example 2: Listing Column Information Column information, such as name, datatype, length, precision, scale, and default data values, can be listed using one of the views ending with the _COLUMNS suffix. For example, the following query lists all of the default column values for the EMP_TAB and DEPT_TAB tables:

```
SELECT Table_name, Column_name, Data_default
    FROM User_tab_columns
    WHERE Table_name = 'DEPT_TAB' OR Table_name = 'EMP_TAB';
```

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

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

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

Example 3: Listing Dependencies of Views and Synonyms When you create a view or a synonym, the view or synonym is based on its underlying base object. The `_DEPENDENCIES` data dictionary views can be used to reveal the dependencies for a view and the `_SYNONYMS` data dictionary views can be used to list the base object of a synonym. For example, the following query lists the base objects for the synonyms created by the user `JWARD`:

```
SELECT Table_owner, Table_name
    FROM All_synonyms
    WHERE Owner = 'JWARD';
```

This query could return information similar to the following:

```
TABLE_OWNER                    TABLE_NAME
------------------------------ ------------
SCOTT                          DEPT_TAB
SCOTT                          EMP_TAB
```

3 Selecting a Datatype

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

See Also:

- *Oracle9i Application Developer's Guide - Object-Relational Features* , for information about more complex types, such as object types, varrays, and nested tables.
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* , for information about LOB datatypes.
- *PL/SQL User's Guide and Reference* , for information the PL/SQL data types. 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 to treat values of one datatype differently from values of another datatype. For example, Oracle can add values of `NUMBER` datatype, but not values of `RAW` datatype.

Oracle supplies the following built-in datatypes:

- Character datatypes
  - `CHAR`
  - `NCHAR`
  - `VARCHAR2 and VARCHAR`
  - `NVARCHAR2`
  - `CLOB`
  - `NCLOB`
  - `LONG`
- `NUMBER` datatype
- 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`

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

> **See Also:** See *Oracle9i SQL Reference* for general descriptions of these datatypes, and see *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about the `LOB` datatypes.

Table 3–1 summarizes the information about each Oracle built-in datatype.

Table 3–1 Summary of Oracle Built-In Datatypes

Datatype	**Description**	**Column Length and Default**
`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. 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.	Variable for each row, up to 4000 bytes per row. Consider the character set (single-byte or multibyte) before setting *size*. A maximum *size* must be specified.

Table 3–1 Summary of Oracle Built-In Datatypes (Cont.)

`NCHAR (`*`size`*`)`	Fixed-length Unicode character data of length *size* characters.	Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters. (The number of bytes is 2 times this number for the `AL16UTF16` encoding and 3 times this number for the `UTF8` encoding.) The upper limit is 2000 bytes per row. Default is 1 character.
`NVARCHAR2 (`*`size`*`)`	Variable-length Unicode character data of length *size* characters. A maximum *size* must be specified.	Variable for each row. Column *size* is the number of characters. (The number of bytes may be up to 2 times this number for a the `AL16UTF16` encoding and 3 times this number for the `UTF8` encoding.) The upper limit is 4000 bytes per row. Default is 1 character.
`CLOB`	Single-byte character data	Up to 2^{32} - 1 bytes, or 4 gigabytes.
`NCLOB`	Unicode national character set (`NCHAR`) data.	Up to 2^{32} - 1 bytes, or 4 gigabytes.
`LONG`	Variable-length character data.	Variable for each row in the table, up to 2^{32} - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility.
`NUMBER (`*`p, s`*`)`	Variable-length numeric data. Maximum precision *p* and/or scale *s* is 38.	Variable for each row. The maximum space required for a given column is 21 bytes per row.
`DATE`	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E.	Fixed at 7 bytes for each row in the table. Default format is a string (such as `DD-MON-RR`) specified by the `NLS_DATE_FORMAT` parameter.
`INTERVAL YEAR (`*`precision`*`) TO MONTH`	A period of time, represented as years and months. The *precision* value specifies the number of digits in the `YEAR` field of the date. The precision can be from 0 to 9, and defaults to 2 for years.	Fixed at 5 bytes.

Table 3–1 Summary of Oracle Built-In Datatypes (Cont.)

`INTERVAL DAY (`*`precision`*`) TO SECOND (`*`precision`*`)`	A period of time, represented as days, hours, minutes, and seconds. The *precision* values specify the number of digits in the `DAY` and the fractional `SECOND` fields of the date. The precision can be from 0 to 9, and defaults to 2 for days and 6 for seconds.	Fixed at 11 bytes.
`TIMESTAMP (`*`precision`*`)`	A value representing a date and time, including fractional seconds. (The exact resolution depends on the operating system clock.) The precision value specifies the number of digits in the fractional second part of the `SECOND` date field. The precision can be from 0 to 9, and defaults to 6	Varies from 7 to 11 bytes, depending on the precision. The default is determined by the `NLS_TIMESTAMP_FORMAT` initialization parameter.
`TIMESTAMP (`*`precision`*`) 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'`.	Fixed at 13 bytes. The default is determined by the `NLS_TIMESTAMP_TZ_FORMAT` initialization parameter.
`TIMESTAMP (`*`precision`*`) 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.	Varies from 7 to 11 bytes, depending on the precision. The default is determined by the `NLS_TIMESTAMP_FORMAT` initialization parameter.
`BLOB`	Unstructured binary data	Up to 2^{32} - 1 bytes, or 4 gigabytes.

Table 3–1 Summary of Oracle Built-In Datatypes (Cont.)

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

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: *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

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

  See Also: "Restrictions on LONG and LONG RAW Datatypes"

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

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

Space Usage

- To store data more efficiently, use the VARCHAR2 datatype. The CHAR datatype blank-pads and stores trailing blanks up to a fixed column length for all column values, while the VARCHAR2 datatype does not 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 from the database character set to the character set defined for the user session by the NLS_LANGUAGE parameter, where these are different.

Column Lengths for Single-Byte and Multibyte Character Sets

The lengths of CHAR and VARCHAR2 columns 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 physically 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:

- *Oracle9i Globalization and National Language Support Guide*
- *Oracle9i SQL Reference*
- *Oracle Time Series User's Guide*

for information about globalization support within Oracle and support for different character encoding schemes.

Implicit Conversion Between CHAR/VARCHAR2 and NCHAR/NVARCHAR2

In previous releases (Oracle8*i* and earlier), 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'`. Now, you can specify `NCHAR` and `NVARCHAR2` without the `N` qualifier, and can mix them with `CHAR` and `VARCHAR2` values in SQL statements and functions.

Comparison Semantics

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

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

Because Oracle blank-pads values stored in `CHAR` columns but not in `VARCHAR2` columns, a value stored in a `VARCHAR2` column may take up less space than if it were stored in a `CHAR` column. For this reason, a full table scan on a large table containing `VARCHAR2` columns may read fewer data blocks than a full table scan on a table containing the same data stored in `CHAR` columns. If your application often performs full table scans on large tables containing character data, 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 uses different semantics to compare values of each datatype. You might choose one datatype over the other if your application is sensitive to the differences between these semantics. For example, if you want Oracle to ignore trailing blanks when comparing character values, then you must store these values in `CHAR` columns.

> **See Also:** For more information on comparison semantics for these datatypes, see the *Oracle9i SQL Reference*.

Representing Numeric Data

Use the `NUMBER` datatype to store real numbers in a fixed-point or floating-point format. Numbers using this datatype are guaranteed to be portable among different Oracle platforms, and offer up to 38 decimal digits of precision. You can store positive and negative numbers of magnitude 1 x 10^{-130} through $9.9\overline{9}$ x10^{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 right of 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 given. Table 3–2 shows examples of how data different scale factors affect storage.

Table 3–2 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, exceeds precision)
7,456,123.89	NUMBER (7, -2)	7456100

See Also: For information about the internal format for the `NUMBER` datatype, see *Oracle9i Database Concepts*.

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 precise values, down 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` values when the time zone is not significant. For example, you might use it in an application that schedules teleconferences, where each participant sees 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. You should not use it in three-tier applications, such as those involving a web server, because in that case the client is the web server, so data displayed in a web browser is formatted according to the time zone of the web server rather than the time zone of the browser.

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

Date Format

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

```
'13-NOV-1992'
```

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 Julian dates might not be compatible with Julian dates generated by other date algorithms. For information about Julian dates, see *Oracle9i Database Concepts*.

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`.

Time Format

Time is stored in 24-hour format, `HH24:MI:SS`. By default, the time in a date field is 12:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the `TO_DATE` function with a format mask indicating the time portion, as in:

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)
```

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

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

Performing Date Arithmetic

Oracle provides a number of functions 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`
- `EXTRACT`
- `NUMTODSINTERVAL`
- `NUMTOYMINTERVAL`
- `TO_DATE` (and its opposite, `TO_CHAR`)
- `TO_DSINTERVAL`
- `TO_TIMESTAMP`
- `TO_TIMESTAMP_TZ`
- `TO_YMINTERVAL`

See Also: *Oracle9i SQL Reference*. for full details about each function.

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.

Handling Time Zones

Oracle provides a number of functions to help with calculations involving time zones. Some useful functions include:

- CURRENT_DATE
- CURRENT_TIMESTAMP
- DBTIMEZONE
- EXTRACT
- FROM_TZ
- LOCALTIMESTAMP
- SESSIONTIMEZONE
- SYS_EXTRACT_UTC
- SYSTIMESTAMP
- TO_TIMESTAMP_TZ

See Also: *Oracle9i SQL Reference*. for full details about each function.

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

Centuries and the Year 2000

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

Applications that use the Oracle RDBMS (Oracle7 or later) and exploit the `DATE` data type (for date and/or date with time values) need have no concerns about their stored data and the year 2000. Beginning with Oracle7, the `DATE` data type 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*). The application might hand over a two-digit year to the database, and the procedures that Oracle uses for determining the century could be different from what the programmer expects (see

"Troubleshooting Y2K Problems in Applications" on page 3-19). For this reason, you should review and test your code with regard to the Year 2000.

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 has always stored the YEAR component of a date in its four-digit form.

Here is an example of the RR usage:

```
INSERT INTO emp (empno, deptno,hiredate) VALUES
   (9999, 20, TO_DATE('01-jan-03', 'DD-MON-RR'));

 INSERT INTO emp (empno, deptno,hiredate) VALUES
    (8888, 20, TO_DATE('01-jan-67',  'DD-MON-RR'));

SELECT empno, deptno,
   TO_CHAR(hiredate, 'DD-MON-YYYY') hiredate
FROM emp;
```

This produces the following data:

```
EMPNO          DEPTNO         HIREDATE
----------     ----------     -----------------
8888           20             01-JAN-1967
9999           20             01-JAN-2003
```

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 '1900' is not the twentieth century (which began in 1901) but rather the nineteenth century.

The following workaround computes the correct century for any Common Era (CE, formerly known as AD) date. If *userdate* is a CE date for which you want the true century, use the 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 it is '00', then it is a year in which the Oracle century is one year too large, and compute a date in the preceding year (whose Oracle century is the desired true century). Otherwise, use the Oracle century.

See Also: For more information about date format codes, see *Oracle9i SQL Reference*.

Storing Dates in Character Data Types

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 which 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, we advise that you convert dates to use the Oracle `DATE` data type. If this is not feasible, store the dates in a form which is language and format independent, and which handles full years. For example, utilize `‘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 displayed or received from users or other programs.

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 unambiguous.
- The time is represented fully. Since the most significant elements occur first, character-based sort operations will process the dates correctly.

The “S” format element prefixes BC dates with “-“.

Viewing Date Settings

The following views let you verify what your settings are:

- `V$NLS_DATABASE_PARAMETERS` — shows instance-wide NLS parameters, whether the default or a value explicitly declared in the initialization parameter file.
- `NLS_SESSION_PARAMETERS` — shows current session values, which may have been changed by means of `ALTER SESSION`

A format model 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 to use in returning a value from the database.
- To specify the format for a value you have specified for Oracle to store in the database.

Please note that the format does not change the internal representation of the value in the database.

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

Altering Date Settings

You may set the date format in your environment or as the default for the entire database. If you set this in your environment, it will override the setting in the initialization parameter.

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 theSession 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 init.ora `NLS_DATE_FORMAT` parameter. To change the default date format for the entire database, change `INIT.ORA` to include the following

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

The `NLS_DATE_FORMAT` setting relies on the above order. Therefore, for a client/server application, `NLS_DATE_FORMAT` must be set on the server and on the client.

Caution: Changing this parameter at the *database* level will change all existing date fields as described above. Oracle Corporation suggests that you 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 are seen to arise from incorrect use of Oracle 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

You 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 which contain dates earlier than 1900, and then adjust for this.

Y2K Example: Wide Range of Years Stored as 2 Digits

Examine your applications to determine if it processes dates prior to 1950 or later than 2049, and store the year as 2-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 number in the database.

Y2K Example: Handling Feb. 29, 2000

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

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

The above query returns 02/28/2000. This is consistent with the defined behavior of the ‘RR’ format element, 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 as the Oracle Server engine 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’ format mask 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

When the DECODE function is used and if the third argument has data type CHAR, VARCHAR2, or if it is NULL, then Oracle converts the return value to datatype VARCHAR2. Therefore, the following statement:

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

inserts date 31.12.1900.

Another sample statement:

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

inserts date 04.10.1901.

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

Note: When inserting the record into a table, Oracle implicitly converts the string into a date, using the first 2-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

If creating a partitioned table using a DATE data type column in the partition key, use a 4-digit year when specifying 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 views depend on the session state. In particular, a predicate with a 2-digit year, such as:

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

is allowed in a view. Interpretation of the full 4-digit year depends on the setting of `NLS_DATE_FORMAT`.

Representing Geographic Coordinate Data

To represent Geographic Information System (GIS) or spatial data in the database, you can use the 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`s or `BFILE`s, or store it externally on a web 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 User's Guide and 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 *inter*Media Text. It stores the search data in a special kind of index, and lets you query it 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 Data Types

In times gone by, the way to represent large data objects in the database was to use the `LONG`, `RAW`, and `LONG RAW` types. Oracle recommends that current applications use the various LOB types, such as `CLOB`, `BLOB` and `BFILE`, for this data.

See the *Oracle9i Application Developer's Guide - Large Objects (LOBs)* , for information about LOB datatypes.

The following sections deal with ways to migrate data from the older datatypes to the LOB types. The LOB types can be used in many situations that formerly required other types such as `LONG` or `VARCHAR2`.

Migrating LONG Datatypes to LOB Datatypes

The `LONG` datatype can store variable-length character data containing up to two gigabytes of information, depending upon available memory. `LONG` columns have many of the characteristics of `VARCHAR2` columns. You can use them in `SELECT` lists, `SET` clauses of `UPDATE` statements, and `VALUES` clauses of `INSERT` statements.

Oracle Corporation recommends using the `LONG` datatype only for backward compatibility with old applications. For new applications, you should use the `CLOB` and `NCLOB` datatypes for large amounts of character data. Typically, you can change LONG data to LOBs in your tables without changing existing applications. SQL, PL/SQL, and OCI interfaces for LONG data can all work on LOB data as well.

See Also:

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about the CLOB and NCLOB datatypes.
- *Oracle Call Interface Programmer's Guide* for details about each of the OCI functions.
- *Oracle9i SQL Reference* for syntax of the ALTER TABLE command.

Changing a LONG or LONG RAW Column to a LOB Datatype

You can use the ALTER TABLE command to change the underlying datatype of a column from LONG to CLOB, or LONG RAW to BLOB. For example:

```
ALTER TABLE employees MODIFY (resume BLOB) DISABLE STORAGE IN ROW;
ALTER TABLE newspaper MODIFY (article CLOB DEFAULT 'Has not been written yet');
```

This technique preserves all the constraints and triggers on the table. All indexes must be rebuilt. Any domain indexes on a long column, such as indexes for data cartridge or interMedia applications, must be dropped before changing the type of the column.

Restrictions on Changing LONG or LONG RAW Columns to LOB Datatypes

1. LOBs are not allowed in clustered tables. So if a table is a part of a cluster, its LONG or LONG RAW column cannot be changed to LOB.
2. If a table is replicated or has materialized views, and its LONG column is changed to LOB, you might have to manually fix the replicas.
3. Not all triggers are preserved when the column is changed to a LOB datatype.
   a. LOB columns are not allowed in the column list of an UPDATE trigger. For example, the following trigger becomes invalid after changing the type of the column to a LOB, and cannot be recompiled:

      ```
      CREATE TABLE t(changed_col LONG);
      CREATE TRIGGER trig BEFORE UPDATE OF lobcol ON t ...;
      ```

   b. If a view with a LOB column has an INSTEAD OF trigger, string inserts or updates are not allowed on the LOB column. These operations were allowed when the column was a LONG. So some SQL statements that worked before the migration do not work afterwards. For example:

      ```
      create table t(a LONG);
      ```

```
create view v as select * from t;
create trigger trig instead of insert on v....;
alter table t modify (a CLOB);
insert into v values ('abc'); -- Throws an error because
-- Implicit conversion from LOB is not allowed in instead-of triggers.
```

Transparent Access to LOBs from Applications that Use LONG and LONG RAW Datatypes

If your application uses DML (INSERT, UPDATE, DELETE) statements from SQL or PL/SQL for LONG or LONG RAW data, these statements work the same after the column is converted to a LOB datatype. You can use input parameters and output buffers of various character types, and they are converted to and from the corresponding LOB datatypes, and truncated if the output type is not large enough to hold the entire result. For example, you can SELECT a CLOB into a character variable, or a BLOB into a RAW variable.

The following SQL functions that accept or output character types now accept or output CLOB data as well:

||, CONCAT, INSTR, INSTRB, LENGTH, LENGTHB, LIKE, LOWER, LPAD, LTRIM, NLS_LOWER, NLS_UPPER, NVL, REPLACE, RPAD, RTRIM, SUBSTR, SUBSTRB, TRIM, UPPER

In PL/SQL, all the SQLfunctions listed above and the comparison operators (>, =, < and !=), and all user-defined procedures and functions, accept CLOB datatypes as parameters or output types. You can also assign a CLOB to a character variable and vice versa in PL/SQL.

If your application uses OCI calls to perform piecewise inserts, updates, or fetches of LONG data, these calls work the same after the column is converted to a LOB datatype. You can define a CLOB column as SQLT_CHR or a BLOB column as SQLT_BIN and select the LOB data directly into a CHARACTER or RAW buffer without selecting out the locator first. The OCI functions that provide this transparent access, by accepting datatypes of SQLT_LNG, SQLT_CHR, SQLT_BIN, and SQLT_LBI) are:

- OCIBindByName()
- OCIBindByPos()
- OCIDefineByPos()

Restrictions on LONG and LONG RAW Datatypes

You can reference LONG columns in SQL statements in these places:

- SELECT lists
- SET clauses of UPDATE statements
- VALUES clauses of INSERT statements

The use of LONG values is subject to some restrictions:

- A table can contain only one LONG column.
- You cannot create an object type with a LONG attribute.
- LONG columns cannot appear in WHERE clauses or in integrity constraints (except that they can appear in NULL and NOT NULL constraints).
- LONG columns cannot be indexed.
- A stored function cannot return a LONG value.
- You can declare a variable or argument of a PL/SQL program unit using the LONG datatype. However, you cannot then call the program unit from SQL.
- Within a single SQL statement, all LONG columns, updated tables, and locked tables must be located on the same database.
- LONG and LONG RAW columns cannot be used in distributed SQL statements and cannot be replicated.
- If a table has both LONG and LOB columns, you cannot bind more than 4000 bytes of data to both the LONG and LOB columns in the same SQL statement. However, you can bind more than 4000 bytes of data to either the LONG or the LOB column.
- A table with LONG columns cannot be stored in a tablespace with automatic segment-space management.

LONG columns cannot appear in certain parts of SQL statements:

- GROUP BY clauses, ORDER BY clauses, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements
- The UNIQUE operator of a SELECT statement
- The column list of a CREATE CLUSTER statement
- The CLUSTER clause of a CREATE MATERIALIZED VIEW statement
- SQL built-in functions, expressions, or conditions
- SELECT lists of queries containing GROUP BY clauses

- SELECT lists of subqueries or queries combined by the UNION, INTERSECT, or MINUS set operators
- SELECT lists of CREATE TABLE ... AS SELECT statements
- ALTER TABLE ... MOVE statements
- SELECT lists in subqueries in INSERT statements

Triggers can use the LONG datatype in the following manner:

- A SQL statement within a trigger can insert data into a LONG column.
- If data from a LONG column can be converted to a constrained datatype (such as CHAR and VARCHAR2), a LONG column can be referenced in a SQL statement within a trigger.
- Variables in triggers cannot be declared using the LONG datatype.
- :NEW and :OLD cannot be used with LONG columns.

You can use the Oracle Call Interface functions to retrieve a portion of a LONG value from the database.

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

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

Example of LONG Datatype

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

```
CREATE TABLE Article_header
    (Id             NUMBER  PRIMARY KEY,
    Title           VARCHAR2(200),
    First_author    VARCHAR2(30),
    Journal         VARCHAR2(50),
    Pub_date        DATE);

CREATE TABLE article_text
```

```
    (Id                NUMBER
                       REFERENCES
                       Article_header,
    Text               LONG);
```

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

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

```
SELECT First_author
    FROM Article_header
    WHERE Journal = 'NATURE'
        AND TO_CHAR(Pub_date, 'MM YYYY') = '07 1991';
```

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

Using RAW and LONG RAW Datatypes

Note: The RAW and LONG RAW datatypes are provided for backward compatibility with existing applications. For new applications, you should use the BLOB and BFILE datatypes for large amounts of binary data.

See Also: See *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about the BLOB and BFILE datatypes.

The RAW and LONG RAW datatypes store data that is not interpreted by Oracle (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.

Net8 and the Export and Import utilities do not perform character conversion when transmitting RAW or LONG RAW data. When Oracle automatically converts RAW or

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

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

> **See Also:** For more information about restrictions on LONG RAW data, see "Restrictions on LONG and LONG RAW Datatypes" on page 3-24.

Addressing Rows Directly with the ROWID Datatype

Every row in an Oracle 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 an Oracle database has a pseudocolumn named ROWID.

> **See Also:** *Oracle9i Database Concepts* for general information about the ROWID pseudocolumn and the ROWID datatype.

Extended ROWID Format

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

The extended ROWID includes the following information:

- Data object (segment) identifier
- Datafile identifier
- Block identifier
- Row identifier

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

```
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: *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about other ways to get the data object identifier, using the `DBMS_ROWID` package functions.

Different Forms of the ROWID

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

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:** *Oracle9i Supplied 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 Oracle7 data, and the extended form of the ROWID is required only in global indexes on partitioned tables. New tables always get extended ROWIDs.

> **See Also:** *Oracle9i Database Administrator's Guide.*

It is possible for an Oracle7 client to access a more recent database, and vice versa. A client in this sense can include a remote database accessing a server using database links, as well as a client 3GL or 4GL application accessing a server.

> **See Also:** There is more information on the `ROWID_TO_EXTENDED` function in *Oracle9i Supplied PL/SQL Packages and Types Reference* and *Oracle9i Database Migration*.

Accessing an Oracle7 Database from an Oracle9i Client The `ROWID` values that are returned are always restricted `ROWIDs`. Also, Oracle9*i* uses restricted `ROWIDs` when returning a `ROWID` value to an Oracle7 or earlier server.

The following `ROWID` functionality works when accessing an Oracle7 Server:

- Selecting a `ROWID` and using the obtained value in a `WHERE` clause
- `WHERE CURRENT OF` cursor operations
- Storing `ROWID`s in user columns of `ROWID` or `CHAR` type
- Interpreting `ROWID`s using the hexadecimal encoding (not recommended, use the `DBMS_ROWID` functions)

Accessing an Oracle9i Database from an Oracle7 Client Oracle9*i* returns `ROWID`s 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 `ROWID`s in user columns of `CHAR(18)` datatype

Import and Export It is not possible for an Oracle7 client 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 an Oracle database using ANSI / ISO, DB2, and SQL / DS datatypes. Oracle internally converts such datatypes to Oracle datatypes.

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

Table 3–3 ANSI Datatype Conversions to Oracle Datatypes

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

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

Table 3–4 shows the DB2 and SQL / DS conversions.

Table 3–4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes

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

Table 3–4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes

DB2 or SQL/DS Datatype	Oracle Datatype
DATE	DATE
TIMESTAMP	TIMESTAMP

How Oracle Converts Datatypes

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

- TO_NUMBER()
- TO_CHAR()
- TO_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 below.

Datatype Conversion During Assignments

For assignments, Oracle can automatically convert the following:

- VARCHAR2, NVARCHAR2, CHAR, or NCHAR to NUMBER

- `NUMBER` to `VARCHAR2` or `NVARCHAR2`
- `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 can convert the datatype of the value used in the assignment to that of the assignment's target.

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

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

```
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 automatically converts the data provided in the following assignment within the body of a stored procedure:

  ```
  VAR1 := 0;
  ```

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

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

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

- `UPDATE` *table* `SET` *column* = *expression*

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

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

- `SELECT` *column* `INTO` *variable* `FROM` *table*

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

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

Datatype Conversion During Expression Evaluation

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

Oracle can automatically convert the following:

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

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

Some common types of expressions follow:

- Simple expressions, such as:

  ```
  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 uses the rule for expression evaluation when a datatype conversion is needed in places not covered by the rule for assignment conversions.

In assignments of the form:

```
variable := expression
```

Oracle first evaluates *expression* using the 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 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:

```
CREATE TABLE tab1 (a SYS.AnyData);
/
-- Insert a built-in type value after explicit conversion to an AnyData.
insert into tab1 values (SYS.AnyData.Convert(5));
/
-- Insert a user-defined type value after explicit conversion to an AnyData.
insert into tab1 values(SYS.AnyData.Convert(employee(5555, "John"));
/
declare
a SYS.AnyData;
t SYS.AnyType;
CURSOR C1 is SELECT a FROM tab1;
n NUMBER;
e EMPLOYEE;
tc PLS_INTEGER;
prec, scale, len, csid, csfrm, count, rval PLS_INTEGER;
sch, tname, version VARCHAR2(50);
begin
OPEN C1;
LOOP
FETCH C1 INTO a;
EXIT WHEN C1%NOTFOUND;
t := a.AnyDataGetType();
tc := t.AnyTypeGetInfo(prec, scale, len, csid. csfrm, sch, tname, version, count
);
/* Dynamically describe the type and do appropriate explicit casts. */
if (tc == DBMS_TYPES.TYPECODE_NUMBER) THEN
rval := a.AnyDataGetValue(n);
DBMS_OUTPUT.PUT_LINE(n);
ELSE if ((tc == DBMS_TYPES.TYPECODE_OBJECT) AND (sch == 'SCOTT') AND
(tname == 'EMPLOYEE')) THEN
rval := a.AnyDataGetValue(e);
DBMS_OUTPUT.PUT_LINE('EMPLOYEE ( ' || e.empno || ',' || e.ename || ')');
END IF;
END LOOP;
EXCEPTION
WHEN DBMS_TYPES.TYPE_MISMATCH
  DBMS_OUTPUT.PUT_LINE('This operation is not allowed on this type.');
END;
/
```

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

See Also:

Oracle9i Supplied PL/SQL Packages and Types Reference for details about the `DBMS_TYPES` package.

Oracle9i 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 `CLOB`s. 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:

- *Oracle9i Application Developer's Guide - XML* for information about all aspects of working with XML.
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for details about the `XMLType` type and the `DBMS_XMLQuery`, `DBMS_XMLGEN`, and `DBMS_XMLSave` packages.
- *Oracle9i SQL Reference* for information about the `SYS_XMLGEN` and `SYS_XMLAGG` functions.

4 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 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 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 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 constrain, because SELECT in Oracle uses "consistent read" and 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, and the database does not do this for you.

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's data will ever attempt to conflict with an existing row's data.

Because Oracle 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, put a `NOT NULL` constraint on at least one indexed column.

> **See Also:** "Defining Relationships Between Parent and Child Tables" on page 4-11

A `NOT NULL` constraint is specified like this:

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

Figure 4–1 Table with NOT NULL Integrity Constraints

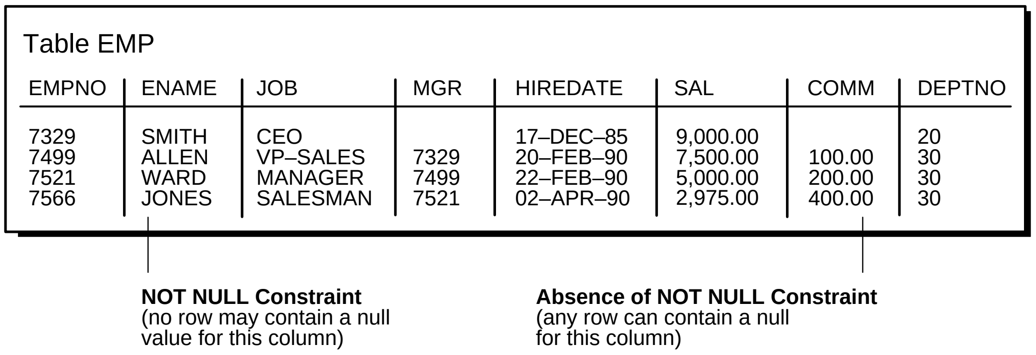


Table EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17–DEC–85	9,000.00		20
7499	ALLEN	VP–SALES	7329	20–FEB–90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22–FEB–90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02–APR–90	2,975.00	400.00	30

NOT NULL Constraint
(no row may contain a null value for this column)
Absence of NOT NULL Constraint
(any row can contain a null for this column)

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
);
```

See Also: For another example of assigning a default column value, refer to the section "Creating Tables".

Setting Default Column Values

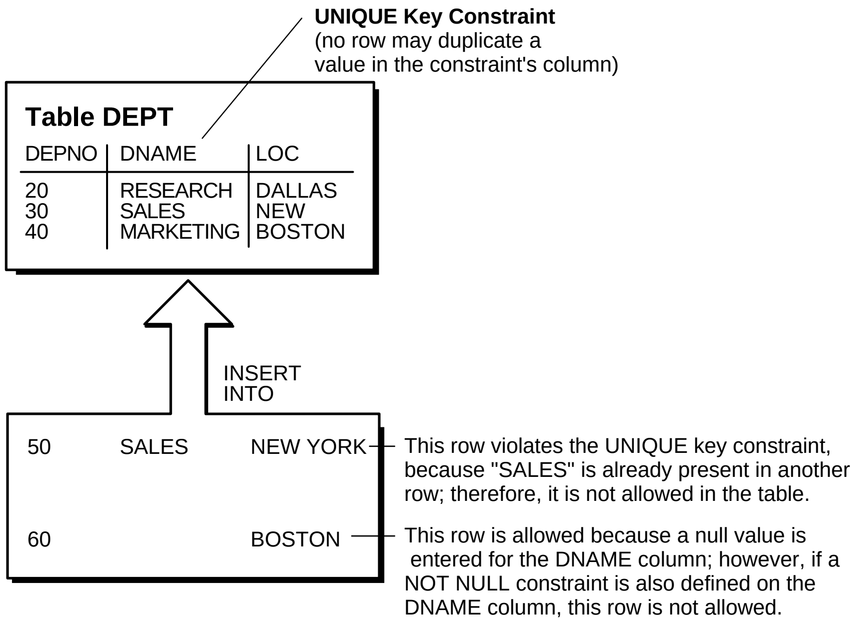
Default values can include any literal, or almost any expression, including calls to `SYSDATE`, `SYS_CONTEXT`, `USER`, `USERENV`, and `UID`. Default values cannot include expressions that refer to a sequence, PL/SQL function, column, `LEVEL`, `ROWNUM`, or `PRIOR`. The datatype of a default literal or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

Figure 4–2 Table with a UNIQUE Key Constraint


UNIQUE Key Constraint
(no row may duplicate a
value in the constraint's column)
Table DEPT
DEPNO
DNAME
LOC
20
30
40
RESEARCH
SALES
MARKETING
DALLAS
NEW
BOSTON
INSERT
INTO
50
SALES
NEW YORK
This row violates the UNIQUE key constraint, because "SALES" is already present in another row; therefore, it is not allowed in the table.
60
BOSTON
This row is allowed because a null value is entered for the DNAME column; however, if a NOT NULL constraint is also defined on the DNAME column, this row is not allowed.

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.
- 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.
- 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.

When to Use UNIQUE Key Integrity Constraints

Choose columns for unique keys carefully. The purpose of these contraints 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.

Note: Although `UNIQUE` key constraints allow null values, you cannot have identical values in the non-null columns of a composite `UNIQUE` key constraint.

Some examples of good unique keys include:

- An employee's social security number (the primary key is the employee number)
- A truck's license plate number (the primary key is the truck number)
- A customer's phone number, consisting of the two columns `AREA` and `PHONE` (the primary key is the customer number)
- A department's name and location (the primary key is the department number)

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: *Oracle9i 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 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: Depending on this relationship, you may want to define additional integrity constraints including the foreign key, as listed in the section "Defining Relationships Between Parent and Child Tables" on page 4-11.

Figure 4–3 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.

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 nulls in composite foreign keys, which requires that all components of the key be null or all be non-null, define a `CHECK` constraint that allows only all nulls or all non-nulls in the composite foreign key. 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 nulls in composite foreign keys, which requires the non-null portions of the key to appear in the corresponding portions in the primary or unique key of a single row in the referenced table. You can often use triggers to handle this case, as described in Chapter 15, "Using Triggers".

Figure 4–3   Tables with Referential Integrity Constraints

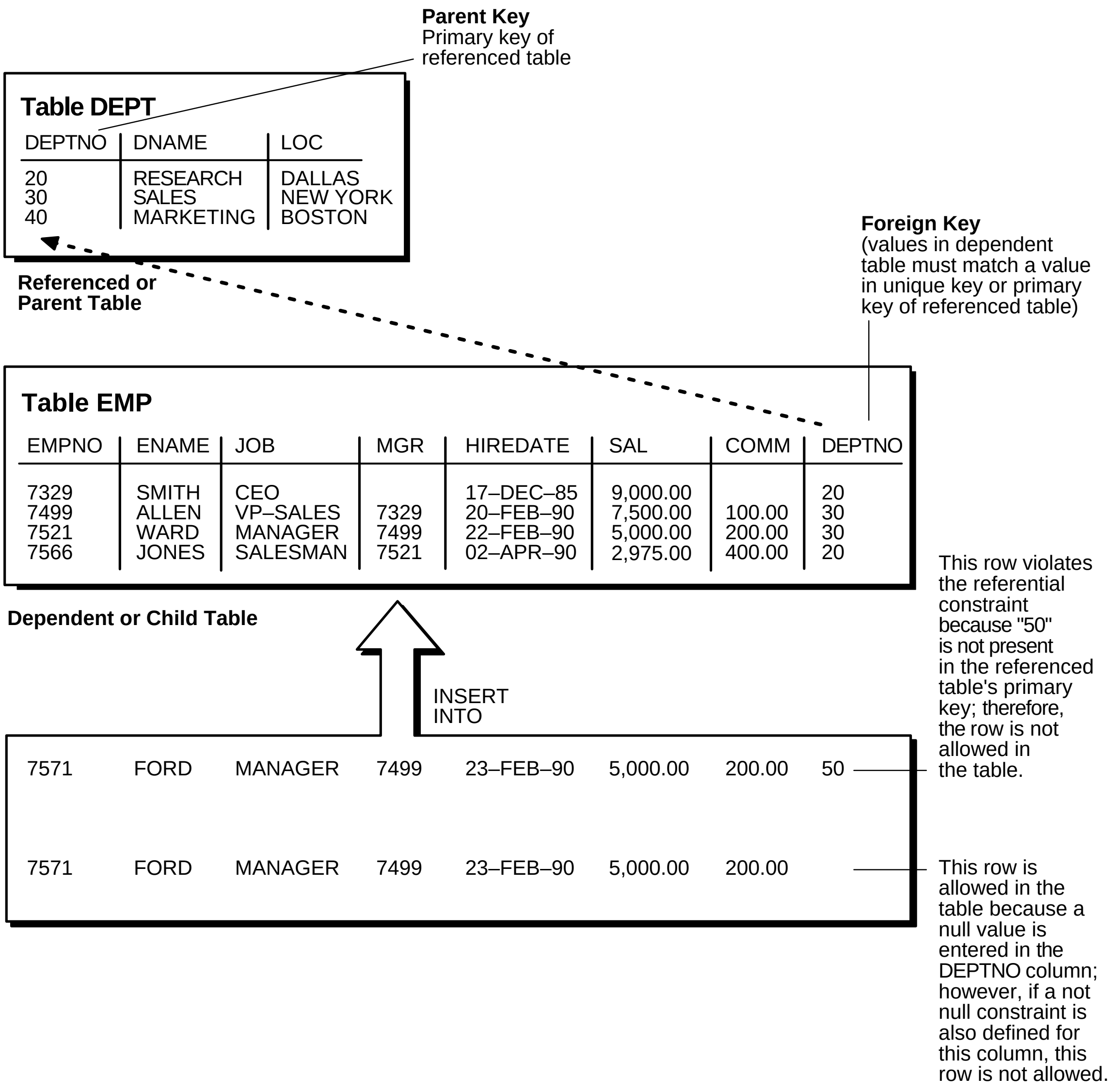

Parent Key
Primary key of referenced table
Table DEPT
Referenced or Parent Table
Foreign Key
(values in dependent table must match a value in unique key or primary key of referenced table)
Table EMP
Dependent or Child Table
INSERT INTO
This row violates the referential constraint because "50" is not present in the referenced table's primary key; therefore, the row is not allowed in the table.
This row is allowed in the table because a null value is entered in the DEPTNO column; however, if a not null constraint is also defined for this column, this row is not allowed.

Table DEPT

DEPTNO	DNAME	LOC
20	RESEARCH	DALLAS
30	SALES	NEW YORK
40	MARKETING	BOSTON

Table EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17–DEC–85	9,000.00		20
7499	ALLEN	VP–SALES	7329	20–FEB–90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22–FEB–90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02–APR–90	2,975.00	400.00	20

7571	FORD	MANAGER	7499	23–FEB–90	5,000.00	200.00	50
7571	FORD	MANAGER	7499	23–FEB–90	5,000.00	200.00	

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. However, any number of rows in the child table can reference the same parent key value.

This model establishes a "one-to-many" relationship between the parent and foreign keys. However, each row in the child table must have a reference to a parent key value; the absence of a value (a null) in the foreign key is not allowed. The same example in the previous section can be used to illustrate such a relationship. However, in this case, employees must have a reference to a specific department.

UNIQUE Constraint on the Foreign Key When a `UNIQUE` constraint is defined on the foreign key, one row in the child table can reference a parent key value. This model allows nulls in the foreign key.

This model establishes a "one-to-one" relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. For example, assume that the employee table had a column named `MEMBERNO`, referring to an employee’s membership number in the company’s insurance plan. Also, a table named `INSURANCE` has a primary key named `MEMBERNO`, and other columns of the table keep respective information relating to an employee’s insurance policy. The `MEMBERNO` in the 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 parent key value. Because nulls are not allowed in the foreign key, each row in the child table must explicitly reference a value in the parent key.

This model establishes a "one-to-one" relationship between the parent and foreign keys that does not allow undetermined values (nulls) in the foreign key. If you expand the previous example by adding a NOT NULL constraint on the MEMBERNO column of the 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 allows a column to be referenced by multiple FOREIGN KEY constraints; effectively, there is no limit on the number of dependent keys. This situation might be present if a single column is part of two different composite foreign keys.

Deferring Constraint Checks

When Oracle 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: For more details about the SET CONSTRAINTS statement, see the *Oracle9i SQL Reference.*

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

- Tables that 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

When dealing with bulk data being manipulated by outside applications, you can defer checking constraints for validity until the end of a transaction.

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 a statement 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 deferrable 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 checks to see if an existing index can be used to enforce uniqueness for the constraint. If there is no such index, Oracle creates one.

Minimizing Space and Time Overhead for Indexes Associated with Constraints

When Oracle 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, or if 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: Deferrable UNIQUE and PRIMARY keys all must use non-unique indexes.

To reuse existing indexes when creating unique and primary key constraints, you can include USING INDEX in the constraint clause. Fpr 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: *Oracle9i 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: For more information about triggers that enforce referential integrity, refer to Chapter 15, "Using Triggers".

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 4-18

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: *Oracle9i 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. Therefore, 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's salary is greater than zero or the employee's commission is greater than or equal to zero." But note that if a row is inserted with a null salary, then the row does not violate the `CHECK` constraint regardless of whether 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 *Oracle9i 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.

Notice how all constraints are given a name. 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:** *Oracle9i Database Administrator's Guide* for information on creating and maintaining constraints for a large production database.

Defining Integrity Constraints with the CREATE TABLE Command: Example

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);
```

Defining Constraints with the ALTER TABLE Command: Example

You can also define integrity constraints using the constraint clause of the ALTER TABLE command. For example, the following examples of ALTER TABLE statements show the definition of several integrity constraints:

```
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 4-27

Naming Integrity Constraints

Assign names to `NOT NULL`, `UNIQUE KEY`, `PRIMARY KEY`, `FOREIGN KEY`, and `CHECK` constraints using the `CONSTRAINT` option of the constraint clause. This name must be unique with respect to other constraints that you own. If you do not specify a constraint name, one is assigned by Oracle.

Picking your own name makes error messages for constraint violations more understandable, and prevents the creation of multiple constraints 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 4-28 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 everyone's employee number by adding 1000 to the existing number)
- When importing or exporting one table at a time

Turning off integrity constraints temporarily speeds 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 either 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:** This procedure is discussed in the section "Fixing Constraint Exceptions" on page 4-24.

Enabling Constraints

When you define an integrity constraint in a CREATE TABLE or ALTER TABLE statement, Oracle 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 are 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 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 4-24 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 in 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 in 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 when the rows of the table violate the integrity constraint. The statement is rolled back and the constraint is not enabled.

> **See Also:** "Fixing Constraint Exceptions" on page 4-24 for more information about rows that violate integrity constraints.

Disabling Existing Constraints

If you need to perform a large load or update when the 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 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 4-28 and *Oracle9i 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: "Managing FOREIGN KEY Integrity Constraints" on page 4-26 and the *Oracle9i Database Administrator's Guide*

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: *Oracle9i Database Administrator's Guide* for more information about fixing 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: For information on the parameters you can modify, see the `ALTER TABLE` section in *Oracle9i SQL Reference*.

Examples of MODIFY CONSTRAINT

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;
```

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:** "Managing FOREIGN KEY Integrity Constraints" on page 4-26 and the *Oracle9i Database Administrator's Guide*.

Managing FOREIGN KEY Integrity Constraints

General information about defining, enabling, disabling, and dropping all types of integrity constraints is given in the previous sections. The following section supplements this information, focusing specifically on issues regarding FOREIGN KEY integrity constraints, which enforce relationships between columns in different tables.

Rules for FOREIGN KEY Integrity Constraints

The following topics are of interest when defining FOREIGN KEY integrity constraints.

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 assumes that you intend to reference the primary key of the specified table. Alternatively, you can explicitly specify the column(s) to reference in the parent table within parentheses. Oracle automatically checks to verify that this column list references a primary or unique key of the parent table. If it does not, 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 both the parent and the child table.

- **The Parent Table** The creator of the referential integrity constraint must own the parent table or have REFERENCES object privileges on the columns that constitute the parent key of the parent table.
- **The Child Table** The creator of the referential integrity constraint must have the ability to create tables (that is, the CREATE TABLE or CREATE ANY TABLE system privilege) or the ability to alter the child table (that is, the ALTER object privilege for the child table or the ALTER ANY TABLE system privilege).

In both cases, necessary privileges *cannot* be obtained via a role; they must be explicitly granted to the creator of the constraint.

These restrictions allow:

- The owner of the child table to explicitly decide what constraints are enforced on her or his tables and the other users that can create constraints on her or his tables
- The owner of the parent table to explicitly decide if foreign keys can depend on the primary and unique keys in her tables

Choosing How Foreign Keys Enforce Referential Integrity

Oracle allows different types of referential integrity actions to be enforced, as specified with the definition of a FOREIGN KEY constraint:

- **Prevent Update or Delete of Parent Key** The default setting prevents the update or deletion 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);
```

Restriction on Enabling FOREIGN KEY Integrity Constraints

`FOREIGN KEY` integrity constraints cannot be enabled if the referenced primary or unique key's constraint is not present or not enabled.

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:** Refer to *Oracle9i Database Reference* for detailed information about each view.

Examples of Defining Integrity Constraints

Consider the following `CREATE TABLE` statements that 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 the one below 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 table below summarizes the characters used for each constraint type.

Constraint Type	**Character**
PRIMARY KEY	P
UNIQUE KEY	U
FOREIGN KEY	R
CHECK, NOT NULL	C

Note: An additional constraint type is indicated by the character "V" in the CONSTRAINT_TYPE column. This constraint type corresponds to constraints created by the WITH CHECK OPTION for views. See Chapter 2, "Managing Schema Objects" for more information about views and the WITH CHECK OPTION.

Example 2: Distinguishing NOT NULL Constraints from CHECK Constraints In the previous example, several constraints are listed with a constraint type of "C". To distinguish which constraints are NOT NULL constraints and which are CHECK constraints in the EMP_TAB and DEPT_TAB tables, issue 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 the one below is returned:

```
CONSTRAINT_NAME  SEARCH_CONDITION
---------------  ----------------------------------------
LOC_CHECK1       loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')
SYS_C00278       ENAME IS NOT NULL
SYS_C00279       DEPTNO IS NOT NULL
```

Notice the following:

- NOT NULL constraints are clearly identified in the SEARCH_CONDITION column.

- The conditions for user-defined CHECK constraints are explicitly listed in the SEARCH_CONDITION column.

Example 3: Listing Column Names that Constitute an Integrity Constraint The following query lists all columns that constitute the constraints defined on all tables accessible to you, the user:

```
SELECT Constraint_name, Table_name, Column_name
    FROM User_cons_columns;
```

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

```
CONSTRAINT_NAME  TABLE_NAME  COLUMN_NAME
---------------  ----------- ---------------
DEPT_FKEY        EMP_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
```

5

Selecting an Index Strategy

This chapter discusses the 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 Function-Based Indexes

See Also:

- *Oracle9i Database Performance Guide and Reference* for detailed information about using indexes.
- *Oracle9i Database Administrator's Guide* for information about creating and managing indexes.
- *Oracle9i SQL Reference* for the syntax of commands to work with indexes.

Guidelines for Application-Specific Indexes

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

Although Oracle allows an unlimited number of indexes on a table, the indexes only help if they are used to speed up queries. Otherwise, they just take up space and add overhead when the indexed columns are updated. You should use the EXPLAIN PLAN feature to determine how the indexes are being used in your queries. Sometimes, if an index is not being used by default, you can use a query hint so that the index is used.

The following sections explain how to create, alter, and drop indexes using SQL commands. Some simple guidelines to follow when managing indexes are included.

> **See Also:** *Oracle9i Database Performance Guide and Reference* for information on query hints and measuring the performance benefits of 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 must use sort space to create the index. Oracle 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 must also swap sort information to and from temporary segments allocated on behalf of the index creation. If the index is extremely large, it might be beneficial to complete the following steps:

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

Under certain conditions, you can load data into a table with the SQL*Loader "direct path load", and an index can be created as data is loaded.

> **See Also:** *Oracle9i 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 15% of the rows in a large table. The percentage varies greatly according to the relative speed of a table scan and how clustered the row data is about the index key. The faster the table scan, the lower the percentage; the more clustered the row data, the higher the percentage.
- Index columns used for joins to improve performance on joins of multiple tables.
- Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key; see Chapter 4, "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 candidates for indexing:

- Values are relatively unique in the column.
- 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 EachTable

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 5–1.

Figure 5–1 The VENDOR_PARTS Table

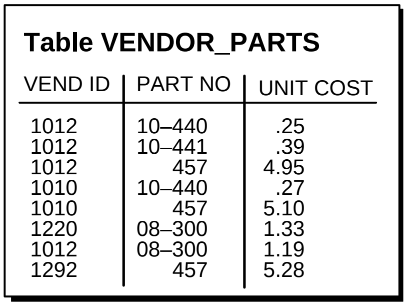


Table VENDOR_PARTS

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 the above example, queries with WHERE clauses using only the PART_NO column also note a performance gain. Because there are only five distinct values, placing a separate index on VENDOR_ID would serve no purpose.

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.

Function-based indexes also require the `QUERY_REWRITE` privilege, and that the `QUERY_REWRITE_ENABLED` initialization parameter to be set to `TRUE`.

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 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, time-series, audio, or video data. If you need to develop such an application, see *Oracle9i Data Cartridge Developer's Guide* .

Oracle 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.

Notes:

- You must set the `QUERY_REWRITE_ENABLED` initialization parameter to `TRUE`.
- 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 5-9.

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 5-11.

> **See Also:** For more information on function-based indexes, see *Oracle9i Database Concepts*. For information on creating function-based indexes, see *Oracle9i Database Administrator's Guide*.

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 the WHERE clause below:

  ```
  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 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 pre-Oracle 8.1 release DESC functionality, 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 NLS 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 set the `QUERY_REWRITE_ENABLED` initialization parameter to `TRUE`, and 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 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 have the initialization parameters COMPATIBLE set to 8.1.0.0.0 or higher, QUERY_REWRITE_ENABLED=TRUE, and QUERY_REWRITE_INTEGRITY=TRUSTED.
- 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:

```
-- The INITIALS() function might return 1 letter, 2 letters, 3 letters, etc.
-- 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;
```

6

Speeding Up Index Access with Index-Organized Tables

This chapter covers the following topics:

See Also: For the syntax of the `ORGANIZATION INDEX` clause of the `CREATE TABLE` statement, see *Oracle9i SQL Reference*.

What Are Index-Organized Tables?

An index-organized table—in contrast to an ordinary table—has its own way of structuring, storing, and indexing data. A comparison with an ordinary table may help to explain its uniqueness.

Index-Organized Tables Versus Ordinary Tables

A row in an *ordinary table* has a stable physical location. Once this location is established, the row never completely moves. Even if it is partially moved with the addition of new data, there is always a row piece at the original physical address—identified by the original physical rowid—from which the system can find the rest of the row. As long as the row exists, its physical rowid does not change. An index in an ordinary table stores both the column data and the rowid.

A row in an *index-organized table* does *not* have a stable physical location. It keeps data in sorted order, in the leaves of a B*-tree index built on the table's primary key. These rows can move around to preserve the sorted order. For example, an insertion can cause an existing row to move to a different slot, or even to a different block.

The leaves of the B*-tree index hold the primary key and the actual row data. Changes to the table data—for example, adding new rows, or updating or deleting existing rows—result only in updating the index.

See Also: For more information on B*-tree indexes, see *Oracle9i Database Concepts*

Advantages of Index-Organized Tables

Because they store rows in a format optimized for access by the primary key, index-organized tables offer the following advantages over ordinary tables:

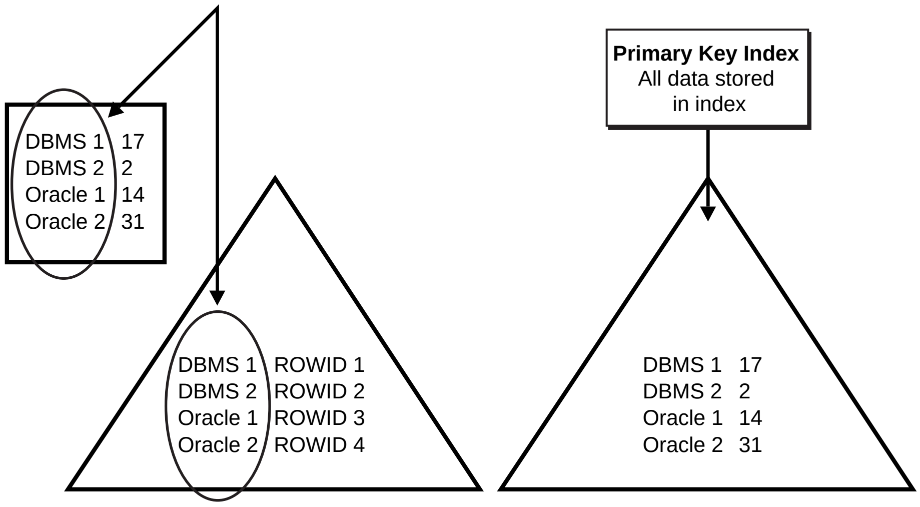
Fast access to table data for queries involving exact match and/or range search on a primary key Once a search has located the key values, the remaining data is present at that location. The index-organized table eliminates the I/O operation of following a rowid back to table data.

Best table organization for 24x7 operations When your database must be available 100% of the time, index-organized tables provide the following advantages:

- You can reorganize an index-organized table or an index-organized table partition (to recover space or improve performance) without rebuilding its secondary indexes. This results in a short reorganization maintenance window.
- You can reorganize an index-organized table online.Along with online reorganization of secondary indexes, this capability eliminates the reorganization maintenance window.

Reduced storage requirements The key columns are not duplicated in both the table and the index, and no additional storage is needed for rowids. When the key columns take up a large part of the row, the storage savings can be as much as 50%.

Figure 6–1 Ordinary Table and an Index versus Index-Organized Table



Features of Index-Organized Tables

You can move your existing data into an index-organized table and do all the operations you would perform in an ordinary table. Some of the features of index-organized tables are:

Full Support for ALTER TABLE Options All of the alter options available on ordinary tables are available for index-organized tables. This includes `ADD`, `MODIFY`, and `DROP COLUMNS` and `CONSTRAINTS`. However, the primary key constraint for an index-organized table cannot be dropped, deferred, or disabled.

Logical ROWID Support Because of the inherent movability of rows in a B*-tree index, a secondary index on an index-organized table cannot be based on a *physical* rowid, which is inherently fixed. Instead, a secondary index for an index-organized table is based on the *logical* rowid. An index-organized table row has no permanent physical address and can move across data blocks when new rows are inserted. However, even if the physical location of a row changes, its logical rowid remains valid.

A logical rowid includes the table's primary key and a physical guess which identifies the database block address at which the row is likely to be found. The physical guess makes rowid-based access to non-volatile index-organized tables comparable to similar access of ordinary tables.

Logical rowids are similar to physical rowids in the following ways:

- You can select `ROWID` from an index-organized table, and access the rows using `ROWID` as a column name in a `WHERE` clause.
- Access through the logical rowid is the fastest possible way to get to a specific row, even if it takes more than one block access to get it.
- The same logical rowid can be used to access a row as long as the primary key value for the row does not change.

The database server uses a single datatype, called universal rowid, to support both logical and physical rowids.

To switch to index-organized tables, applications that use rowids might have to change to universal rowids, but these changes are made easier by the `UROWID` datatype, which lets applications access logical and physical rowids in a unified manner.

For more information: See "Declaring a Universal Rowid Datatype: Example" on page 6-11.

Secondary Index Support Secondary indexes on index-organized tables differ from indexes on ordinary tables in two ways:

- They store logical rowids instead of physical rowids. Thus, a table maintenance operation such as `ALTER TABLE MOVE` does not make the secondary index unusable.
- The logical rowid also includes a physical guess that provides a direct access to the index leaf block containing the index-organized table row. If the physical guess is correct, a secondary index scan would incur a single additional I/O once the secondary key is found. The performance would be similar to that of a secondary index-scan on an ordinary table.

Both unique and non-unique secondary indexes, as well as function-based secondary indexes, are supported. Bitmap indexes on non-partitioned index-organized tables are supported, provided the index-organized table is created with a mapping table. For more information about mapping tables, see the information about index-organized tables in *Oracle9i Database Concepts*

LOB Columns You can create internal and external `LOB` columns in index-organized tables to store large unstructured data such as audio, video, and images. The SQL DDL, DML, and piece-wise operations on `LOBs` in index-organized tables exhibit the same behavior as in ordinary tables. The main differences are:

- Tablespace mapping—By default (or unless specified otherwise), the `LOB's` data and index segments are created in the tablespace in which the primary key index segment of the index-organized table is created.
- Inline versus Out-of-line storage—`LOBs` in index-organized tables with overflow segments are the same as those in ordinary tables. All `LOBs` in index-organized tables created without an overflow segment are stored out-of-line (that is, the default storage attribute is `DISABLE STORAGE IN ROW`). Specifying an `ENABLE STORAGE IN ROW` for such `LOBs` causes an error.

`LOB` columns are supported in range-partitioned index-organized tables.

Other `LOB` features—such as `BFILEs`, temporary `LOBs`, and varying character width `LOBs`—are also supported in index-organized tables. You use them as you would in ordinary tables.

Parallel Query Queries on index-organized tables involving primary key index scan can be executed in parallel.

Object Support Most of the object features are supported on index-organized tables, including Object Type, VARRAYs, Nested Table, and REF Columns.

SQL*Loader This utility supports both ordinary and direct path load of index-organized tables and their associated indexes (including partitioning support). However, direct path parallel load to an index-organized table is not supported. An alternate method of achieving the same result is to perform parallel load to an ordinary table using SQL*Loader, then use the parallel CREATE TABLE AS SELECT option to build the index-organized table.

Export/Import This utility supports export (both ordinary and direct path) and import of non-partitioned and partitioned index-organized tables.

Distributed Database and Replication Support You can replicate both non-partitioned and partitioned index-organized tables.

Tools The Oracle Enterprise Manager supports generating SQL statements for CREATE and ALTER operations on an index-organized table.

Key Compression Key compression allows elimination of repeated occurrences of key column prefixes in index-organized tables and indexes. The salient characteristics of the scheme are:

- Key compression breaks an index key into a prefix entry and suffix entry. Compression is achieved by sharing the prefix entries among all the suffix entries in an index block.
- Only keys in the leaf blocks of a B*-tree are compressed. Keys in the branch blocks of a B*-tree are still suffix truncated but not subjected to key compression.

Why Use Index-Organized Tables?

There are several occasions when you may prefer to use index-organized tables over ordinary tables.

Index-Organized Tables Are Part of Oracle Advanced Queuing Oracle Advanced Queuing provides message queuing as an integrated part of the database server, and uses index-organized tables to hold metadata information for multiple consumer queues.

Index-Organized Tables Avoid Redundant Data Storage For tables, where the majority of columns form the primary key, there is a significant amount of redundant data stored. You can avoid this redundant storage by using an index-organized table. Also, by using an index-organized table, you increase the efficiency of the primary key-based access to non-key columns.

Index-Organized Tables Are Suited to VLDB and OLTP Applications The ability to partition index-organized tables on a range of column values makes them suitable for VLDB applications.

One major advantage of an index-organized table comes from the logical nature of its secondary indexes. After an `ALTER TABLE MOVE` and `SPLIT` operation, *global* indexes on index-organized tables remain usable because the index rows contain logical rowids. You can avoid a complete index rebuild, which can be very expensive. Also, the `ALTER TABLE MOVE` operation can be done on-line, making index-organized tables ideal for applications requiring 24x7 availability.

Similarly, after an `ALTER TABLE MOVE` operation, *local* indexes on index-organized tables are still usable.

These partition maintenance operations do make the local and global indexes on index-organized table slower as the guess component of the logical rowid becomes invalid. However, the indexes are still usable through the primary key-component of the logical rowid. Note that the invalid physical guesses in these indexes can be fixed online with the help of `ALTER INDEX ... UPDATE BLOCK REFERENCES` operation.

Index-Organized Tables Are Suited to Time-Series Applications Time-series applications use a set of time-stamped rows belonging to a single item, such as a stock price. The ability to cluster rows based on the primary key makes index-organized tables attractive for such applications. By defining an index-organized table with primary key (stock symbol, time stamp), the Oracle8 Time Series option can store and manipulate time-series data efficiently. You can achieve more storage savings by compressing repeated occurrences of the item identifier (for example, the stock symbol) in a time series by using an index-organized table with key compression.

Index-Organized Tables Store Nested Tables Efficiently For a nested table column, Oracle internally creates a storage table to hold all the nested table rows.

You can store the nested table as an index-organized table:

```
CREATE TYPE Project_t AS OBJECT(Pno NUMBER, Pname VARCHAR2(80));
CREATE TYPE Project_set AS TABLE OF Project_t;
CREATE TABLE Employees (Eno NUMBER, Projects PROJECT_SET)
   NESTED TABLE Projects_ntab STORE AS Emp_project_tab
                ((PRIMARY KEY(Nested_table_id, Pno)) ORGANIZATION INDEX)
                RETURN AS LOCATOR;
```

The rows belonging to a single nested table instance are identified by a `NESTED_TABLE_ID` column. If an ordinary table is used to store nested table columns, the nested table rows typically get de-clustered. But when you use an index-organized table, the nested table rows can be clustered based on the `NESTED_TABLE_ID` column.

Index-Organized Tables can Store Extensible Index Data The Extensible Indexing Framework lets you add a new access method to the database. Typically, domain-specific indexing schemes need some storage mechanism to hold their index data. Index-organized tables are ideal candidates for such domain index storage. The interMedia Spatial and Text features use index-organized tables for storing their index data.

Example of an Index-Organized Table

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

```
CONNECT system/manager
GRANT CREATE TABLESPACE TO scott;
CONNECT scott/tiger
CREATE TABLESPACE Ind_tbs DATAFILE ’disk1:moredata2’
SIZE 100K;
CREATE TABLESPACE Doc_tab DATAFILE ’disk1:moredata2’
SIZE 100K;
CREATE TABLESPACE Ovf_tbs DATAFILE ’disk1:moredata3’
SIZE 100K;
CREATE TABLESPACE Ind_ts0 DATAFILE ’disk1:moredata5’
SIZE 100K REUSE;
CREATE TABLESPACE Ov_ts0 DATAFILE ’disk1:moredata6’
SIZE 100K REUSE;
CREATE TABLESPACE Ind_ts1 DATAFILE ’disk1:moredata7’
SIZE 100K REUSE;
CREATE TABLESPACE Ov_ts1 DATAFILE ’disk1:moredata8’
SIZE 100K REUSE;
CREATE TABLESPACE Ind_ts2 DATAFILE ’disk1:moredata9’
SIZE 100K REUSE;
CREATE TABLESPACE Ov_ts2 DATAFILE ’disk1:moredata10’
SIZE 100K REUSE;
CREATE TABLE Doc_tab (tok VARCHAR2(4),id
VARCHAR2(14),freq NUMBER);
```

This example illustrates some of the basic tasks in creating and using index-organized tables. In this example, a text search engine keeps a record of all the web pages that use specific words or phrases, so that it can return a list of hypertext links in response to a search query.

This example illustrates the following tasks:

- Moving Existing Data from an Ordinary Table into an Index-Organized Table: Example
- Creating Index-Organized Tables: Example
- Declaring a Universal Rowid Datatype: Example
- Creating Secondary Indexes on Index-Organized Tables: Example

- Manipulating Index-Organized Tables: Example
- Specifying an Overflow Data Segment: Example
- Determining the Last Non-Key Column Included in the Index Row Head Piece: Example
- Storing Columns in the Overflow Segment: Example
- Modifying Physical and Storage Attributes: Example
- Partitioning an Index-Organized Table: Example
- Rebuilding an Index-Organized Table: Example

Moving Existing Data from an Ordinary Table into an Index-Organized Table: Example

The `CREATE TABLE AS SELECT` command lets you move existing data from an ordinary table into an index-organized table. In the following example, an index-organized table, called `docindex`, is created from an ordinary table called `doctable`.

```
CREATE TABLE Docindex
   (   Token,
       Doc_id,
       Token_frequency,
       CONSTRAINT Pk_docindex PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs
PARALLEL (DEGREE 2)
AS SELECT * from Doc_tab;
```

Note that the `PARALLEL` clause allows the table creation to be performed in parallel.

Creating Index-Organized Tables: Example

To create an index-organized table, you use the `ORGANIZATION INDEX` clause. In the following example, an inverted index—typically used by Web text-search engines—uses an index-organized table.

```
CREATE TABLE Docindex
   (  Token           CHAR(20),
      Doc_id          NUMBER,
      Token_frequency NUMBER,
      CONSTRAINT Pk_docindex PRIMARY KEY (Token, Doc_id)
   )
```

```
ORGANIZATION INDEX TABLESPACE Ind_tbs;
```

Declaring a Universal Rowid Datatype: Example

The following example shows how you declare the UROWID datatype.

```
DECLARE
     Rid UROWID;
BEGIN
     INSERT INTO Docindex VALUES (’Or80’, 2, 30)
                  RETURNING Rowid INTO RID;
     UPDATE Docindex SET Token=’Or81’ WHERE ROWID = Rid;
END;
```

Creating Secondary Indexes on Index-Organized Tables: Example

You can create secondary indexes on index-organized tables to provide multiple access paths. The following example shows the creation of an index on (doc_id, token).

```
CREATE INDEX Doc_id_index on Docindex(Doc_id, Token);
```

This secondary index allows Oracle to efficiently process queries involving predicates on doc_id, as the following example illustrates.

```
SELECT Token FROM Docindex WHERE Doc_id = 1;
```

Manipulating Index-Organized Tables: Example

Applications manipulate the index-organized tables just like an ordinary table, using standard SQL statements for SELECT, INSERT, UPDATE, or DELETE operations. For example, you can manipulate the docindex table as follows:

```
INSERT INTO Docindex VALUES (’Oracle8.1’, 3, 17);
SELECT * FROM Docindex;
UPDATE Docindex SET Token = ’Oracle8’ WHERE Token = ’Oracle8.1’;
DELETE FROM Docindex WHERE Doc_id = 1;
```

Also, you can use SELECT FOR UPDATE statements to lock rows of an index-organized table. All of these operations result in manipulating the primary key B*-tree index. Both query and DML operations involving index-organized tables are optimized by using this cost-based approach.

Specifying an Overflow Data Segment: Example

Storing all non-key columns in the primary key B*-tree index structure may not always be desirable because, for example:

- Each additional non-key column stored in the primary key index reduces the dense clustering of index rows in the B*-tree index leaf blocks

  or because

- A leaf block of aB*-tree must hold at least two index rows, and putting all non-key columns as part of an index row may not be possible.

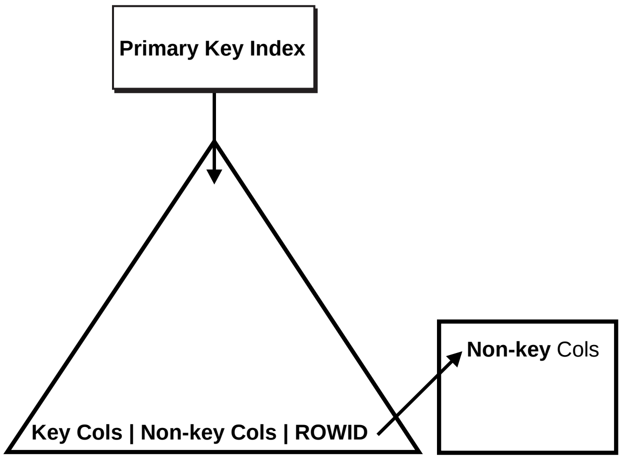
To overcome these problems, you can associate an overflow data segment with an index-organized table. In the following example, an additional column, `token_offsets`, is required for the `docindex` table. This example shows how you can create an index-organized table and use the `OVERFLOW` option to create an overflow data segment.

```
CREATE TABLE Docindex2
   (   Token           CHAR(20),
       Doc_id          NUMBER,
       Token_frequency NUMBER,
       Token_offsets   VARCHAR(512),
       CONSTRAINT Pk_docindex2 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs PCTTHRESHOLD 20
OVERFLOW TABLESPACE Ovf_tbs INITRANS 4;
```

For the overflow data segment, you can specify physical storage attributes such as `TABLESPACE`, `INITRANS`, and so on.

For an index-organized table with an overflow segment, the index row contains a <key, row head> pair, where the row head contains the first few non-key columns and a rowid that points to an overflow row-piece containing the remaining column values. Although this approach incurs the storage cost of one rowid for each row, it nevertheless avoids key column duplication.

Figure 6–2 Overflow Segment



Determining the Last Non-Key Column Included in the Index Row Head Piece: Example

To determine the last non-key column to include in the index row head piece, you use the `PCTTHRESHOLD` option specified as a percentage of the leaf block size. The remaining non-key columns are stored in the overflow data segment as one or more row-pieces. Specifically, the last non-key column to be included is chosen so that the index row size (key +row head) does not exceed the specified threshold (which, in the following example, is 20% of the index leaf block). By default, `PCTTHRESHOLD` is set at 50 when omitted.

The `PCTTHRESHOLD` option determines the last non-key column to be included in the index for each row. It does not, however, allow you to specify that the same set of columns be included in the index for all rows in the table. For this purpose, the `INCLUDING` option is provided.

The `CREATE TABLE` statement in the following example includes all the columns up to the `token_frequency` column in the index leaf block and forces the `token_offsets` column to the overflow segment.

```
CREATE TABLE Docindex3
   (   Token           CHAR(20),
       Doc_id          NUMBER,
       Token_frequency NUMBER,
       Token_offsets   VARCHAR(512),
       CONSTRAINT Pk_docindex3 PRIMARY KEY (Token, Doc_id)
```

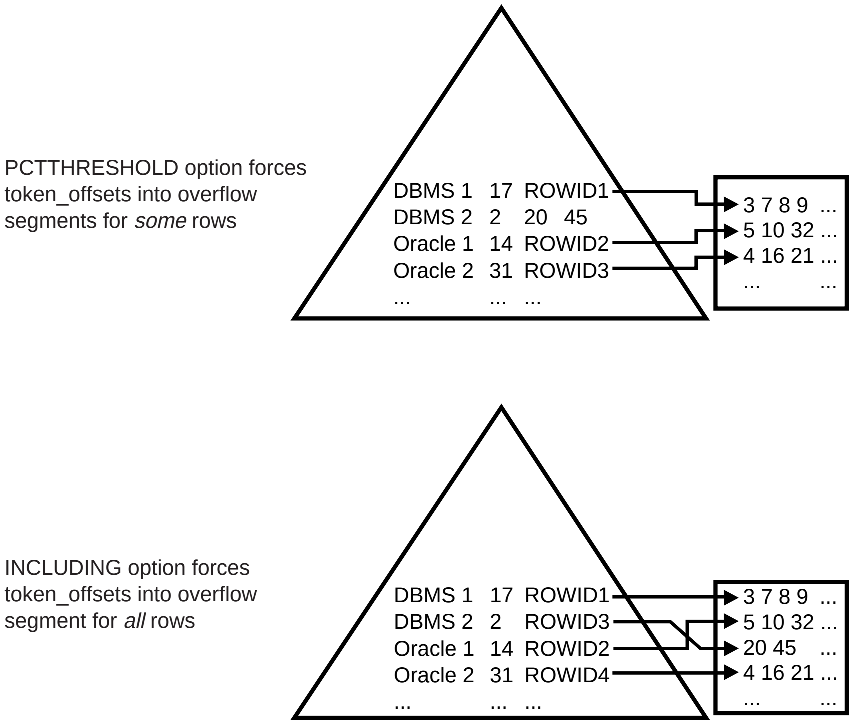
```
    )
ORGANIZATION INDEX TABLESPACE Ind_tbs INCLUDING Token_frequency
OVERFLOW TABLESPACE Ovf_tbs;
```

Such vertical partitioning of a row between the index and data segments allows for higher clustering of rows in the index. This results in better query performance for the columns stored in the index. For example, if the `token_offsets` column is infrequently accessed, then pushing this column out of the index results in better clustering of index rows in the primary key B*-tree structure (Figure 6–3). This in turn results in overall improved query performance. However, there is one additional block access for columns stored in the overflow data segment, and this can slow performance.

Storing Columns in the Overflow Segment: Example

The `INCLUDING` option ensures that all columns after the specified including column are stored in the overflow segment. If the including column specified is such that corresponding index row size exceeds the specified threshold, then the last non-key column to be included is determined according to the `PCTTHRESHOLD` option.

Figure 6–3 PCTTHRESHOLD versus INCLUDING Column Usage



Modifying Physical and Storage Attributes: Example

You can use the `ALTER TABLE` command to modify physical and storage attributes for both the index and overflow data segments as well as alter `PCTTHRESHOLD` and `INCLUDING` column values. The following example sets the `INITRANS` of index segment to 4, `PCTTHRESHOLD` to 20, and the `INITRANS` of the overflow data segment to 6. The altered values are used for subsequent operations on the table.

```
ALTER TABLE Docindex INITRANS 4 PCTTHRESHOLD 20 OVERFLOW INITRANS 6;
```

For index-organized tables created without an overflow data segment, you can add an overflow data segment using `ALTER TABLE ADD OVERFLOW` option. The following example shows how to add an overflow segment to the `docindex` table.

```
ALTER TABLE Docindex ADD OVERFLOW;
```

Analyzing an Index-Organized Table: Example

Index-organized tables are analyzed by the ANALYZE command, just like ordinary tables. The following example analyzes the docindex table:

```
ANALYZE TABLE Docindex COMPUTE STATISTICS;
```

The ANALYZE command analyzes both the primary key index segment and the overflow data segment, and computes logical as well as physical statistics for the table. You can determine how many rows have one or more chained overflow row-pieces using the ANALYZE LIST CHAINED ROWS option. With the logical rowid feature, a separate CHAINED_ROWS table is not needed.

Loading, Exporting, Importing, or Replicating an Index-Organized Table

Data can be loaded into both non-partitioned and partitioned index-organized tables using the ordinary or direct path with the SQL*Loader. The data can also be exported or imported using the Export/Import utility. Index-organized tables can also be replicated in a distributed database just like ordinary tables.

Partitioning an Index-Organized Table: Example

You can partition index-organized tables by range of column values or by a hash value derived from a set of columns. The set of partitioning columns must be a subset of the primary key columns. Only a single partition needs to be searched for to verify the uniqueness of the primary key during DML operations. This preserves the partition independence property.

The following are key aspects of partitioned index-organized tables:

- You must specify the ORGANIZATION INDEX clause to create an index-organized table as part of table-level attributes. This property is implicitly inherited by all partitions.
- You must specify the OVERFLOW option as part of table-level attribute to create an index-organized table with overflow data segment.
- The OVERFLOW option results in the creation of overflow data segments, which are themselves equi-partitioned with the primary key index segments. That is, each partition has an index segment and an overflow data segment.
- For hash-partitioned tables, include the ROW MOVEMENT ENABLE clause of the CREATE TABLE statement. Rows might move from one partition to another due to changes in the columns used to derive the hash value.

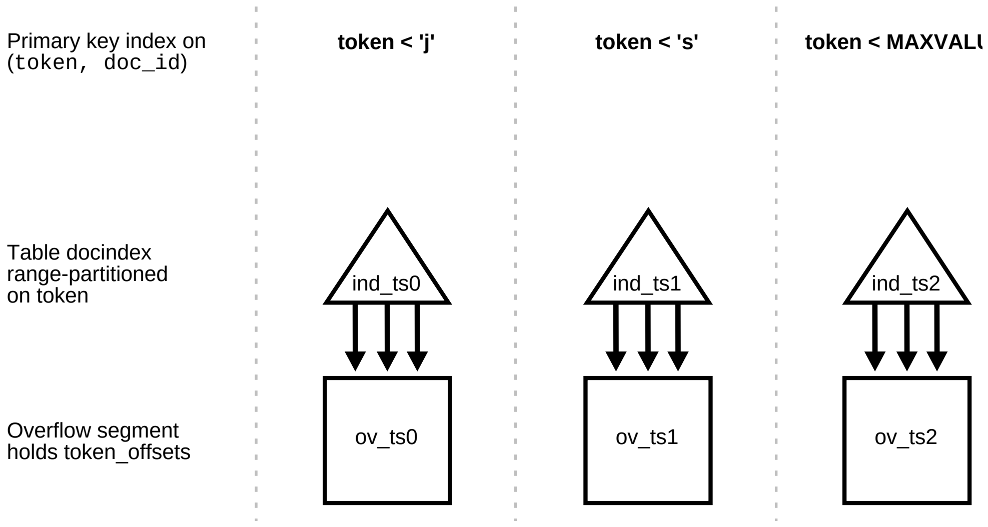
- As in ordinary partitioned tables, you can specify default values for physical attributes at the table-level. These can be overridden for each partition (both for index and overflow data segment).
- The tablespace for index segment, if not specified for a partition, is set to the table level default. If the table level default is not specified, then the default tablespace for the user is used.
- The default values for `PCTTHRESHOLD` and `INCLUDING` column can only be specified at the table level.
- All the attributes that are specified before the `OVERFLOW` keyword apply to primary index segments. All the attributes specified after the `OVERFLOW` keyword apply to overflow data segments.
- The tablespace for an overflow data segment, if not specified for a partition, is set to the table-level default. If the table-level default is not specified, the tablespace of the corresponding partition's index segment is used.

The following example continues the example of the `docindex` table. It illustrates a range partition on token values.

```
CREATE TABLE Docindex4
   (Token           CHAR(20),
    Doc_id          NUMBER,
    Token_frequency NUMBER,
    Token_offsets   VARCHAR(512),
    CONSTRAINT Pk_docindex4 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX INITRANS 4  INCLUDING Token_frequency
OVERFLOW INITRANS 6
      PARTITION BY RANGE(token)
          ( PARTITION P1 VALUES LESS THAN ('j')
TABLESPACE Ind_ts0 OVERFLOW TABLESPACE Ov_ts0,
   PARTITION P2 VALUES LESS THAN ('s')
TABLESPACE Ind_ts1 OVERFLOW TABLESPACE Ov_ts1,
  PARTITION P3 VALUES LESS THAN (MAXVALUE)
TABLESPACE Ind_ts2 OVERFLOW TABLESPACE Ov_ts2);
```

This creates the table shown in Figure 6–4. The `INCLUDING` clause stores the `token_offsets` column in the overflow data segment for each partition.

Figure 6–4 Range-partitioned Index-organized Table with Overflow Segment


Primary key index on (token, doc_id)
token < 'j'
token < 's'
token < MAXVALU
Table docindex range-partitioned on token
ind_ts0
ind_ts1
ind_ts2
Overflow segment holds token_offsets
ov_ts0
ov_ts1
ov_ts2

Partitioned indexes on index-organized tables are supported. Local prefixed, local non-prefixed, and global prefixed partitioned indexes are supported on index-organized tables. The only difference is that these indexes store logical rowids instead of physical rowids.

All of the `ALTER TABLE` operations are available for partitioned index-organized tables. These operations are slightly different with index-organized tables than with ordinary tables:

- For `ALTER TABLE MOVE` partition operations, *all indexes*—local, global, and non-partitioned—remain `USABLE` because the indexes contain logical rowids. However, the guess stored in the logical rowid becomes invalid.
- For `SPLIT` partition operations, all non-partitioned indexes or global index partitions remain usable.
- For `ALTER TABLE EXCHANGE` partition, the target table must be a compatible index-organized table.
- Users can use the `ALTER TABLE ADD OVERFLOW` command to add an overflow segment and specify table-level default and partition-level physical and storage attributes. This operation results in adding an overflow data segment to each partition.

ALTER INDEX operations are very similar to those on ordinary tables. The only difference is that operations that reconstruct the entire index—namely, ALTER INDEX REBUILD and SPLIT_PARTITION—result in reconstructing the guess stored as part of the logical rowid. New ALTER INDEX UPDATE BLOCK REFERENCES syntax fixes the invalid physical guesses without reconstructing the indexes.

Query and DML operations on partitioned index-organized tables work the same as on ordinary partitioned tables.

Compressing the Keys of an Index-Organized Table: Example

You enable key compression by using the COMPRESS clause when specifying physical attributes for the index segment. You can specify the prefix length (as number of columns) to identify how the key can be broken into a prefix and a suffix. Prefix length can be between 1 and the number of primary key columns minus 1.

```
CREATE TABLE Docindex5
   ( Token           CHAR(20),
     Doc_id          NUMBER,
     Token_frequency NUMBER,
     Token_offsets   VARCHAR(512),
     CONSTRAINT pk_docindex5 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs COMPRESS 1 INCLUDING Token_frequency
OVERFLOW TABLESPACE Ovf_tbs;
```

Common prefixes of length 1 (that is, token column) are compressed in the primary key (token, doc_id) occurrences. For the list of primary key values (‘DBMS’, 1), (‘DBMS’, 2), (‘Oracle’, 1), (‘Oracle’, 2), the repeated occurrences of ‘DBMS’ and ‘Oracle’ are compressed away.

If a prefix length is not specified, by default it is set to the number of primary key columns minus 1. You can specify the compress option when creating an index-organized table or when moving an index-organized table using ALTER TABLE MOVE. For example, you can disable compression as follows:

```
ALTER TABLE Docindex5 MOVE NOCOMPRESS;
```

Similarly, the indexes for ordinary tables and index-organized tables can be compressed using the COMPRESS option.

Compressing the Keys for Partitioned Index-Organized Tables: Example You can also compress the keys for partitioned index-organized tables, by specifying the compression clause as part of the table-level defaults. Compression can be enabled or disabled for each partition. The prefix length cannot be changed at the partition level.

```
CREATE TABLE Docindex6
   ( Token          CHAR(20),
     Doc_id         NUMBER,
     Token_frequency NUMBER,
     Token_offsets   VARCHAR(512),
     CONSTRAINT Pk_docindex6 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX INITRANS 4 COMPRESS 1 INCLUDING Token_frequency
OVERFLOW INITRANS 6
            PARTITION BY RANGE(Token)
           ( PARTITION P1 VALUES LESS THAN ('j')
TABLESPACE Ind_ts0 OVERFLOW TABLESPACE Ov_ts0,
   PARTITION P2 VALUES LESS THAN ('s')
TABLESPACE Ind_ts1 NOCOMPRESS OVERFLOW TABLESPACE Ov_ts1,
  PARTITION P3 VALUES LESS THAN (MAXVALUE)
TABLESPACE Ind_ts2 OVERFLOW TABLESPACE Ov_ts2
);
```

All partitions inherit the table-level default for prefix length. Partitions P1 and P3 are created with key-compression enabled. For partition P2, the compression is disabled by the partition level NOCOMPRESS option.

For ALTER TABLE MOVE and SPLIT operations, the COMRPESS option can be altered. The following example rebuilds the partition with key compression enabled.

```
ALTER TABLE Docindex6 MOVE PARTITION P2 COMPRESS;
```

Rebuilding an Index-Organized Table: Example

An SQL command, ALTER TABLE MOVE, lets you rebuild the table. This should be used when the B*-tree structure containing an index-organized table gets fragmented due to a large number of inserts, updates, or deletes. The MOVE option rebuilds the primary key B*-tree index.

By default, the overflow data segment is not rebuilt, except when:

- The OVERFLOW clause is explicitly specified,
- The PCTHRESHOLD and/or INCLUDING column value are altered as part of the MOVE statement.
- Any LOBs are moved explicitly

By default, LOB columns related to index and data segments are not rebuilt, except when the LOB columns are explicitly specified as part of the MOVE statement. The following example rebuilds the B*-tree index containing the table data after setting INITRANS to 6 for index blocks.

```
ALTER TABLE docindex MOVE INITRANS 6;
```

The following example rebuilds both the primary key index and overflow data segment.

```
ALTER TABLE docindex MOVE TABLESPACE Ovf_tbs OVERFLOW TABLESPACE ov_ts0;
```

By default, during the move, the table is not available for other operations. However, you can move an index-organized table using the ONLINE option. The following example allows the table to be available for DML and query operations during the actual move operation. This feature makes the index-organized table suitable for applications requiring 24x7 availability.

Caution: You may need to set your COMPATIBLE initialization parameter to '8.1.3.0' or higher to get the following to work:

```
ALTER TABLE Docindex MOVE ONLINE;
```

ONLINE move is supported only for index-organized tables that do not have an overflow segment.

7

How Oracle Processes SQL Statements

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

Although some Oracle tools and applications simplify or mask the use of SQL, all database operations are performed using SQL, to take advantage of the security and data integrity features built into Oracle.

Overview of SQL Statement Execution

Table 7–1 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 applications.

See Also: Refer to *Oracle9i Database Concepts* for a description of each stage of SQL statement processing 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 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 the Oracle Server flags them as errors and displays the violating syntax.

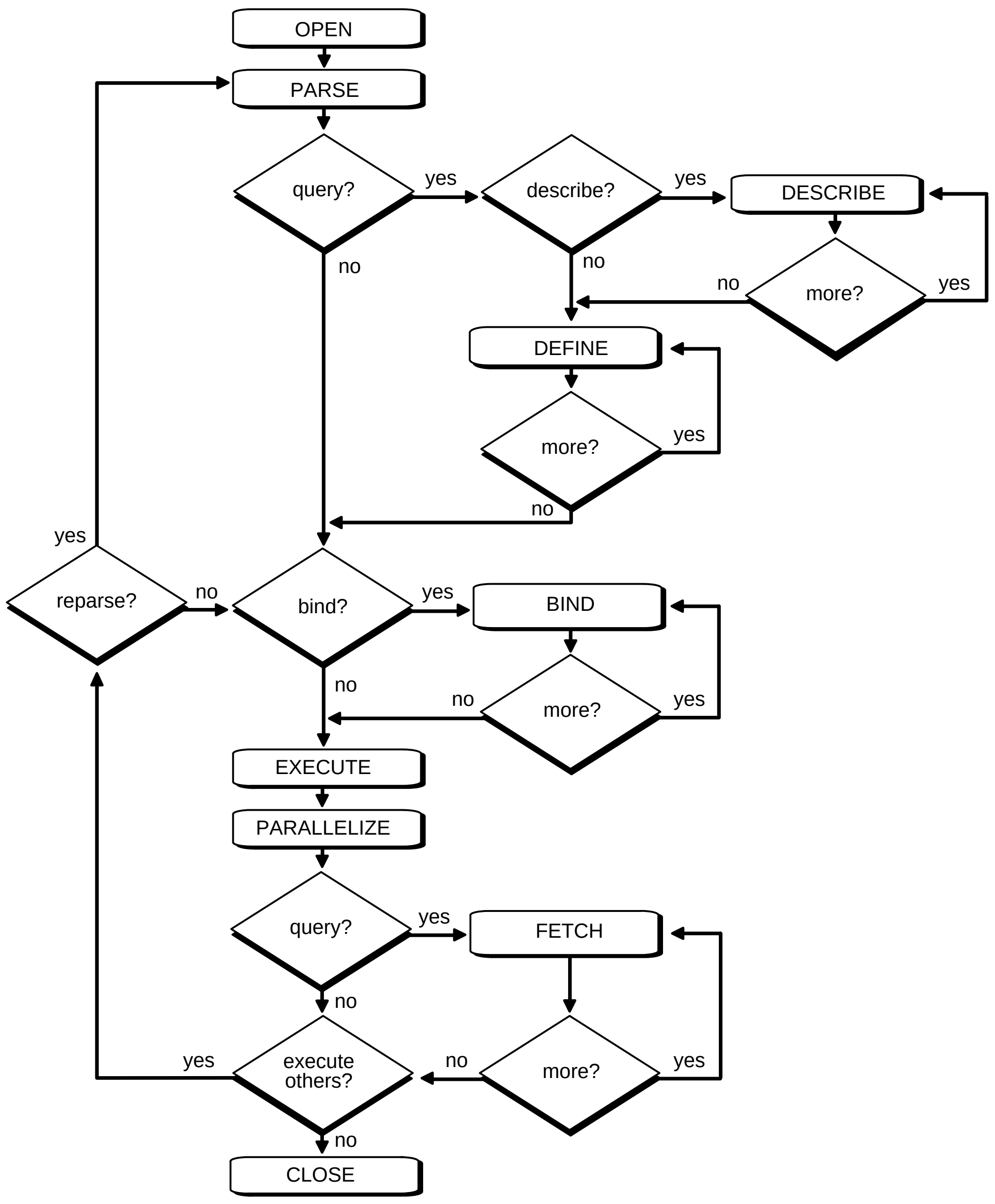
The FIPS flagging feature supports flagging through interactive SQL statements submitted using Enterprise Manager or SQL*Plus. The Oracle Precompilers and SQL*Module also support FIPS flagging of embedded and module language SQL.

When flagging is on and non-standard SQL is encountered, the 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 7–1 The Stages in Processing a SQL Statement



Grouping Operations into Transactions

In general, only application designers using the programming interfaces to Oracle are concerned with which types of actions should be grouped together as one transaction. Transactions must be defined properly so 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 *Oracle9i Database Concepts.*

- Use the `BEGIN_DISCRETE_TRANSACTION` procedure to improve the performance of short, non-distributed transactions.
- Use the `SET TRANSACTION` command with the `USE ROLLBACK SEGMENT` parameter to explicitly assign a transaction to an appropriate rollback segment. This can eliminate the need to dynamically allocate additional extents, which can reduce overall system performance.
- Use the `SET TRANSACTION` command with the `ISOLATION LEVEL` set to `SERIALIZABLE` to get ANSI/ISO serializable transactions.

  See Also:

  - "How Serializable Transactions Interact" on page 7-25
  - *Oracle9i Database Concepts.*

- Establish standards for writing SQL statements so that you can take advantage of shared SQL areas. Oracle recognizes identical SQL statements and allows them to share memory areas. This reduces memory usage on the database server and increases system throughput.

- Use the ANALYZE command to collect statistics that can be used by Oracle to implement a cost-based approach to SQL statement optimization. You can supply additional "hints" to the optimizer as needed.
- Call the DBMS_APPLICATION_INFO.SET_ACTION procedure before beginning a transaction to register and name a transaction for later use when measuring performance across an application. You should specify what type of activity a transaction performs so that the system tuners can later see which transactions are taking up the most system resources.
- Increase user productivity and query efficiency by including user-written PL/SQL functions in SQL expressions as described 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 7-9

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';
```

See Also: For additional information about committing in-doubt distributed transactions, see *Oracle8 Distributed Database Systems.*

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;
```

See Also: For additional information about rolling back in-doubt distributed transactions, see *Oracle8 Distributed Database Systems.*

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

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

SQL Statement	Results
`SAVEPOINT a;`	First savepoint of this transaction
`DELETE...;`	First DML statement of this transaction
`SAVEPOINT b;`	Second savepoint of this transaction
`INSERT INTO...;`	Second DML statement of this transaction
`SAVEPOINT c;`	Third savepoint of this transaction
`UPDATE...;`	Third DML statement of this transaction.
`ROLLBACK TO c;`	`UPDATE` statement is rolled back, savepoint C remains defined
`ROLLBACK TO b;`	`INSERT` statement is rolled back, savepoint C is lost, savepoint B remains defined
`ROLLBACK TO c;`	`ORA-01086` error; savepoint C no longer defined
`INSERT INTO...;`	New DML statement in this transaction
`COMMIT;`	Commits all actions performed by the first DML statement (the `DELETE` statement) and the last DML statement (the second `INSERT` statement) All other statements (the second and the third statements) of the transaction 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 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 control number (SCN) determined when the read-only transaction begins. Because no data locks are acquired, other transactions can query and update data being queried concurrently by a read-only transaction.

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

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

```
SET TRANSACTION READ ONLY;
```

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

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

Declaring and Opening Cursors

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

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

See Also: Parameters are described in *Oracle9i Database Reference*.

Explicitly creating cursors for precompiler programs can offer some advantages in tuning those applications. For example, increasing the number of cursors can often reduce the frequency of parsing and improve performance. If you know how many cursors may be required at a given time, 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 re-execute the statement without starting over, as long as no other SQL statement has been associated with that cursor. This is illustrated in Figure 7–1. Notice that the statement can be re-executed without including the parse stage.

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

To understand the performance characteristics of a cursor, a DBA can retrieve the text of the query represented by the cursor using the `V$SQL` catalog view. Because the results of `EXPLAIN PLAN` on the original query might differ from the way the query is actually processed, the DBA can get more precise information by examining the `V$SQL_PLAN` and `V$SQL_PLAN_STATS` catalog views. The `V$SQL_PLAN_ENV` catalog view shows what parameters have changed from their default values, which might cause the `EXPLAIN PLAN` output to differ from the actual execution plan for the cursor.

See Also: *Oracle9i Database Reference.* for details about each of these catalog 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: For more information about cancelling cursors, see *Oracle Call Interface Programmer's Guide.*

Locking Data Explicitly

Oracle 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 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 two different levels:

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

The following sections describe each option available for overriding the default locking of Oracle. The initialization parameter `DML_LOCKS` determines the maximum number of DML locks allowed.

> **See Also:** See the *Oracle9i 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 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.

Note: A distributed transaction waiting for a table lock can time-out waiting for the requested lock if the elapsed amount of time reaches the interval set by the initialization parameter DISTRIBUTED_LOCK_TIMEOUT. Because no data has been modified, no actions are necessary as a result of the time-out. Your application should proceed as if a deadlock has been encountered. For more information on distributed transactions, refer to *Oracle8 Distributed Database Systems.*

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's 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 (via 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 the above 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 Control Table Locking

Letting Oracle 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.

Table locks are also controlled by the ROW_LOCKING and SERIALIZABLE initialization parameters. By default, SERIALIZABLE is set to FALSE and ROW_LOCKING is set to ALWAYS. In almost every case, these parameters should not be altered. They are provided for sites that must run in ANSI/ISO compatible mode, or that want to use applications written to run with earlier versions of Oracle. Only these sites should consider altering these parameters, as there is a significant performance degradation caused by using other than the defaults.

> **See Also:** *Oracle9i SQL Reference* for details about the `SET TRANSACTION` and `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.

Summary of Nondefault Locking Options

Three combinations of settings for `SERIALIZABLE` and `ROW_LOCKING`, other than the default settings, are available to change the way locking occurs for transactions. Table 7–1 summarizes the nondefault settings and why you might choose to execute your transactions in a particular way.

Table 7–1 Summary of Nondefault Locking Options

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

Table 7–2 illustrates the difference in locking behavior resulting from the three possible settings of the `SERIALIZABLE` option and `ROW_LOCKING` initialization parameter, as shown in Table 7–1.

Table 7–2 Nondefault Locking Behavior

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

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 15, "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 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.

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.

As described on "Choosing a Locking Strategy" on page 7-12, a distributed transaction waiting for a row lock can time-out waiting for the requested lock if the elapsed amount of time reaches the interval set by the initialization parameter DISTRIBUTED_LOCK_TIMEOUT.

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 lock, it has all the features of an Oracle lock, such as deadlock detection. Be certain that any user locks used in distributed transactions are released upon COMMIT, or an undetected deadlock can occur.

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* has 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, above that by check.     *
* This code prints the check. The one printer is opened by all  *
* the cashiers to avoid the overhead of opening and closing it  *
* for every check. This means that lines of output from multiple*
* cashiers could become interleaved if we don’t ensure exclusive*
* access to the printer. The DBMS_LOCK package is used to       *
* ensure exclusive access.                                      *
*****************************************************************
CHECK-PRINT
*
*    Get the lock "handle" for the printer lock.
   MOVE "CHECKPRINT" TO LOCKNAME-ARR.
```

```
    MOVE 10 TO LOCKNAME-LEN.
    EXEC SQL EXECUTE
       BEGIN DBMS_LOCK.ALLOCATE_UNIQUE ( :LOCKNAME, :LOCKHANDLE );
       END; END-EXEC.
*
*    Lock the printer in exclusive mode (default mode).
    EXEC SQL EXECUTE
       BEGIN DBMS_LOCK.REQUEST ( :LOCKHANDLE );
       END; END-EXEC.
*    We now have exclusive use of the printer, print the check.

   ...

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

       END; END-EXEC.
```

Viewing and Monitoring Locks

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

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

Using Serializable Transactions for Concurrency Control

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

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

For most applications, this concurrency model is the appropriate one, 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.

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 7–3.

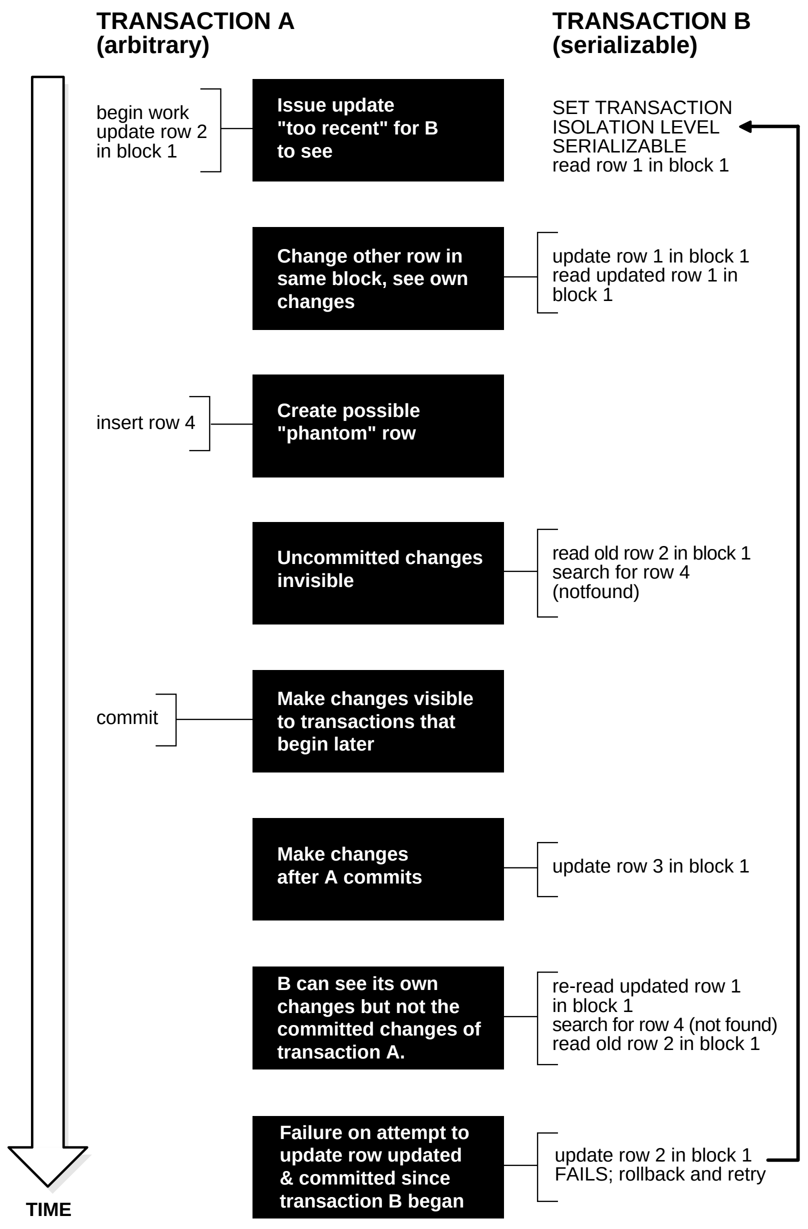
Table 7–3 Summary of ANSI Isolation Levels

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
Notes:	(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 with respect to these isolation levels is summarized below:

`READ UNCOMMITTED`	Oracle 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.
`READ COMMITTED`	Oracle meets the `READ COMMITTED` isolation standard. This is the default mode for all Oracle applications. Because an Oracle query only sees data that was committed at the beginning of the query (the snapshot time), Oracle actually offers more consistency than is required by the ANSI/ISO SQL92 standards for `READ COMMITTED` isolation.
`REPEATABLE READ`	Oracle does not normally support this isolation level, except as provided by `SERIALIZABLE`.
`SERIALIZABLE`	You can set this isolation level using the `SET TRANSACTION` command or the `ALTER SESSION` command.

Figure 7–2   Time Line for Two Transactions


TRANSACTION A
(arbitrary)
TRANSACTION B
(serializable)
begin work
update row 2
in block 1
Issue update "too recent" for B to see
SET TRANSACTION
ISOLATION LEVEL
SERIALIZABLE
read row 1 in block 1
Change other row in same block, see own changes
update row 1 in block 1
read updated row 1 in block 1
insert row 4
Create possible "phantom" row
Uncommitted changes invisible
read old row 2 in block 1
search for row 4
(notfound)
commit
Make changes visible to transactions that begin later
Make changes after A commits
update row 3 in block 1
B can see its own changes but not the committed changes of transaction A.
re-read updated row 1
in block 1
search for row 4 (not found)
read old row 2 in block 1
Failure on attempt to update row updated & committed since transaction B began
update row 2 in block 1
FAILS; rollback and retry
TIME

How Serializable Transactions Interact

Figure 7–3 on page 7-27 shows how a serializable transaction (Transaction B) interacts with another transaction (A, which can be either `SERIALIZABLE` or `READ COMMITTED`).

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

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

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

Setting the Isolation Level 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: *Oracle9i Database Reference* for the complete syntax of the `SET TRANSACTION` and `ALTER SESSION` commands.

The INITRANS Parameter

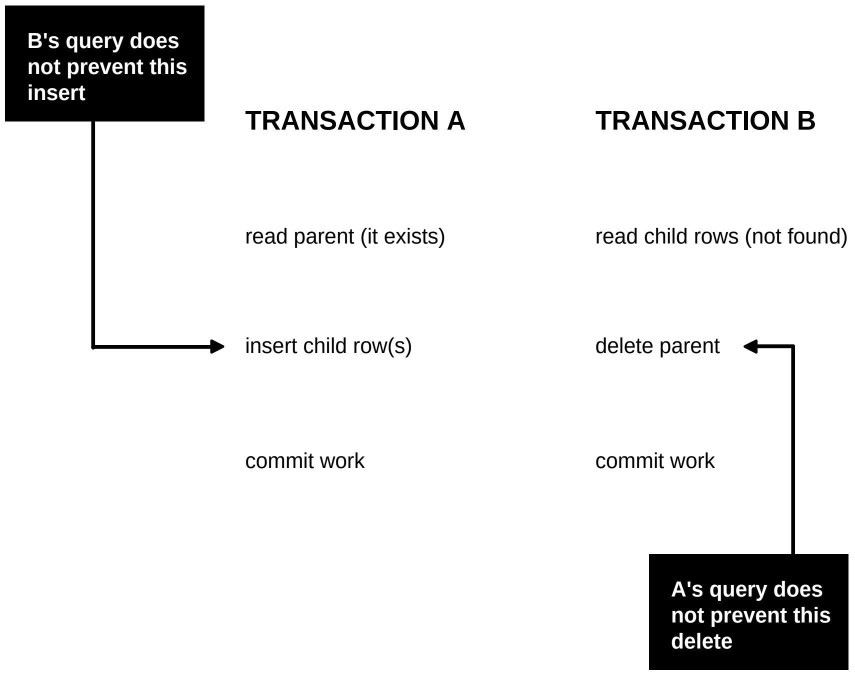
Oracle stores control information in each data block to manage access by concurrent transactions. Therefore, if you set the transaction isolation level to serializable, then you must use the `ALTER TABLE` command to set `INITRANS` to at least 3. This parameter causes Oracle to allocate sufficient storage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

Referential Integrity and Serializable Transactions

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

Figure 7–3 on page 7-27 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 7–3 Referential Integrity Check


B's query does not prevent this insert
TRANSACTION A
TRANSACTION B
read parent (it exists)
read child rows (not found)
insert child row(s)
delete parent
commit work
commit work
A's query does not prevent this delete

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 to prevent the anomaly described above:

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

Referential integrity can also be enforced in Oracle using database triggers, instead of a separate query as in Transaction A above. For example, an INSERT into the child table can fire a 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's execution, and in a transaction executing in SERIALIZABLE mode, the SQL statements see the database as of the beginning of the transaction. In either case, the use of SELECT FOR UPDATE by the trigger correctly enforces referential integrity.

READ COMMITTED and SERIALIZABLE Isolation

Oracle 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 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 transactions executing in READ COMMITTED mode are transaction set consistent on a per-statement basis, because all rows read by a query must be committed before the query begins.

Oracle transactions executing in SERIALIZABLE mode are transaction set consistent on a per-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'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 7–4 summarizes key similarities and differences between READ COMMITTED and SERIALIZABLE transactions.

Table 7–4 Read Committed Versus Serializable Transaction

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

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 isolation modes provide high levels of consistency and concurrency (and performance) through the combination of row-level locking and Oracle's multi-version concurrency control system. Because readers and writers do not block one another in Oracle, 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: For detailed information on autonomous transactions, see *PL/SQL User's Guide and Reference* and Chapter 15, "Using Triggers".

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.

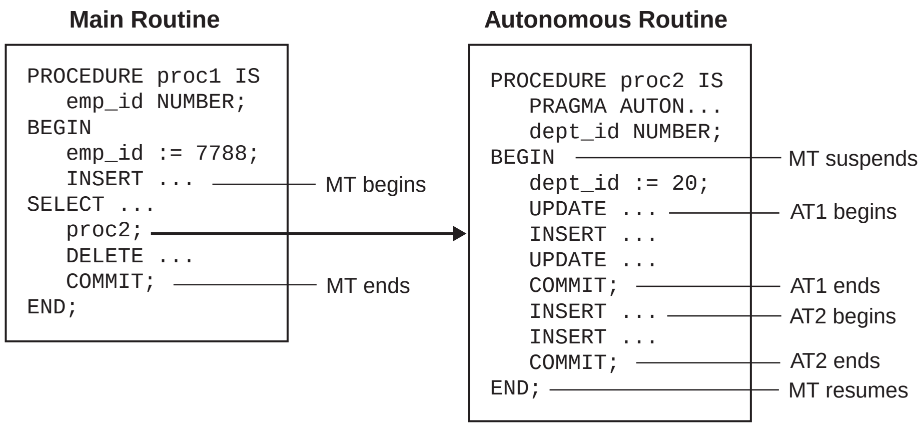
An **autonomous transaction** (AT) is an independent transaction started by another transaction, the **main transaction** (MT). It lets you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction.

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 7–4 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 7–4 Transaction Control Flow



```
PROCEDURE proc1 IS
   emp_id NUMBER;
BEGIN
   emp_id := 7788;
   INSERT ...
SELECT ...
   proc2;
   DELETE ...
   COMMIT;
END;
```

```
PROCEDURE proc2 IS
   PRAGMA AUTON...
   dept_id NUMBER;
BEGIN
   dept_id := 20;
   UPDATE ...
   INSERT ...
   UPDATE ...
   COMMIT;
   INSERT ...
   INSERT ...
   COMMIT;
END;
```

When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes. `COMMIT` and `ROLLBACK` end the active autonomous transaction but do not exit the autonomous routine. As Figure 7–4 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 7–5 illustrates some of the possible sequences autonomous transactions can follow.

Figure 7–5 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.

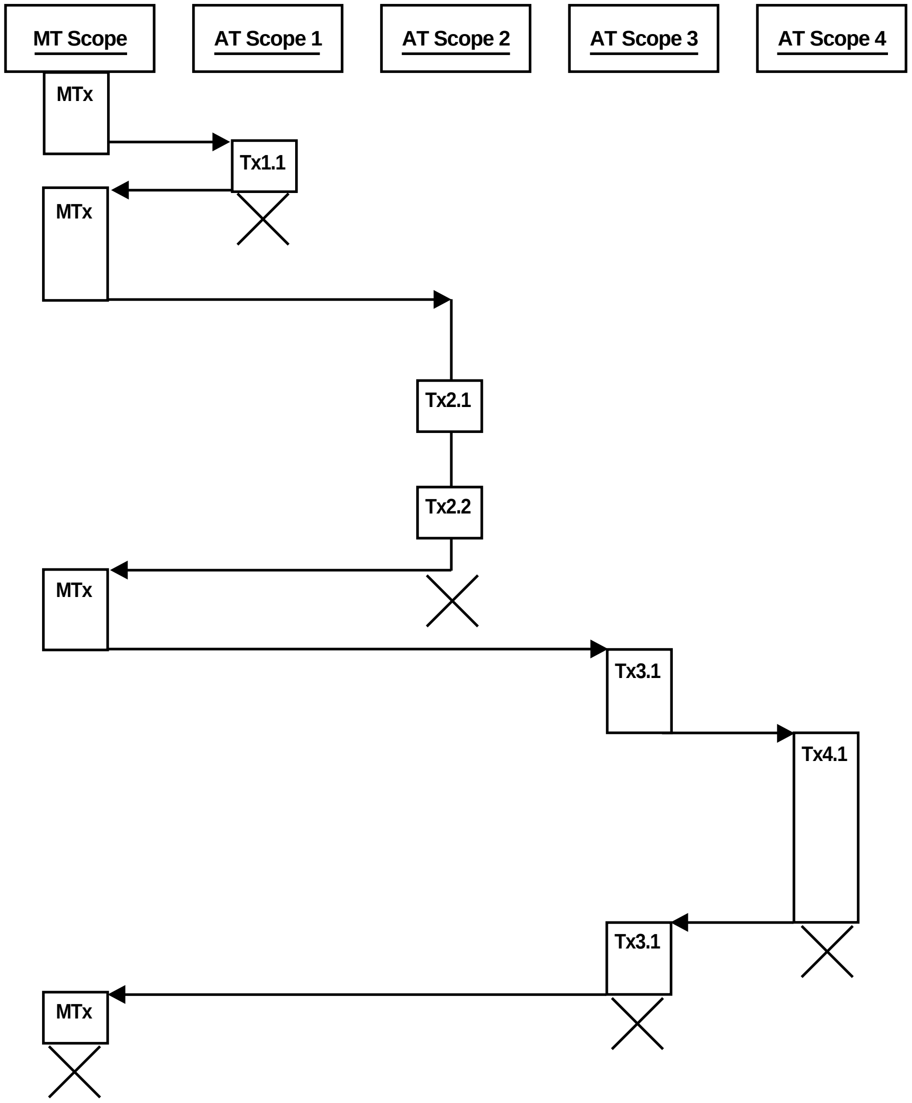
MTx invokes AT Scope 3. MTx suspends, AT Scope 3 begins.

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.

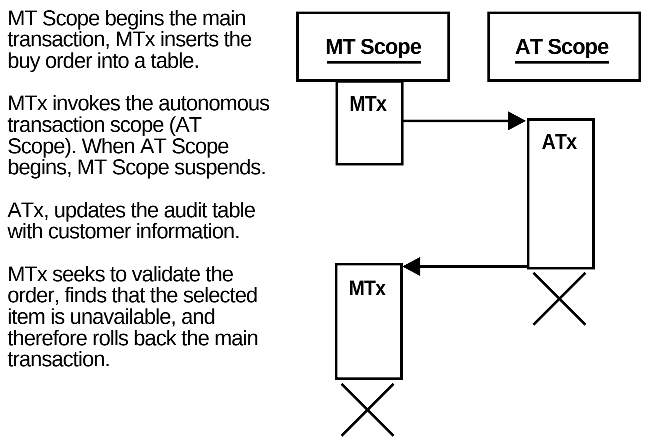
As these examples illustrate, there are four possible outcomes that can occur when you use autonomous and main transactions. The following table presents these possible outcomes. As you can see, there is no dependency between the outcome of an autonomous transaction and that of a main transaction.

Autonomous Transaction	Main Transaction
Commits	Commits
Commits	Rolls back
Rolls back	Commits
Rolls back	Rolls back

Entering a Buy Order

In this example, 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 7–6 Example: A Buy Order


MT Scope begins the main transaction, MTx inserts the buy order into a table.
MTx invokes the autonomous transaction scope (AT Scope). When AT Scope begins, MT Scope suspends.
ATx, updates the audit table with customer information.
MTx seeks to validate the order, finds that the selected item is unavailable, and therefore rolls back the main transaction.
MT Scope
AT Scope
MTx
ATx
MTx

Example: Making a Bank Withdrawal

In the following banking application, a customer tries to make a withdrawal from his or her account. In the process, a main transaction calls one of two autonomous transaction scopes (AT Scope 1, and AT Scope 2).

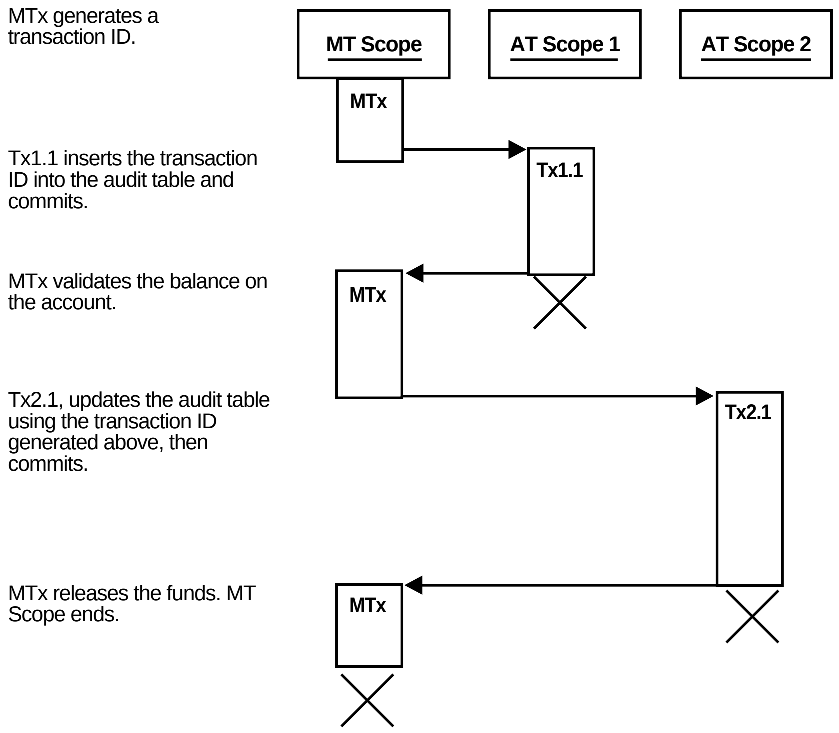
The following diagrams illustrate three possible scenarios for this transaction.

- 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

Figure 7–7 Example: Bank Withdrawal—Sufficient Funds



MTx generates a transaction ID.

Tx1.1 inserts the transaction ID into the audit table and commits.

MTx validates the balance on the account.

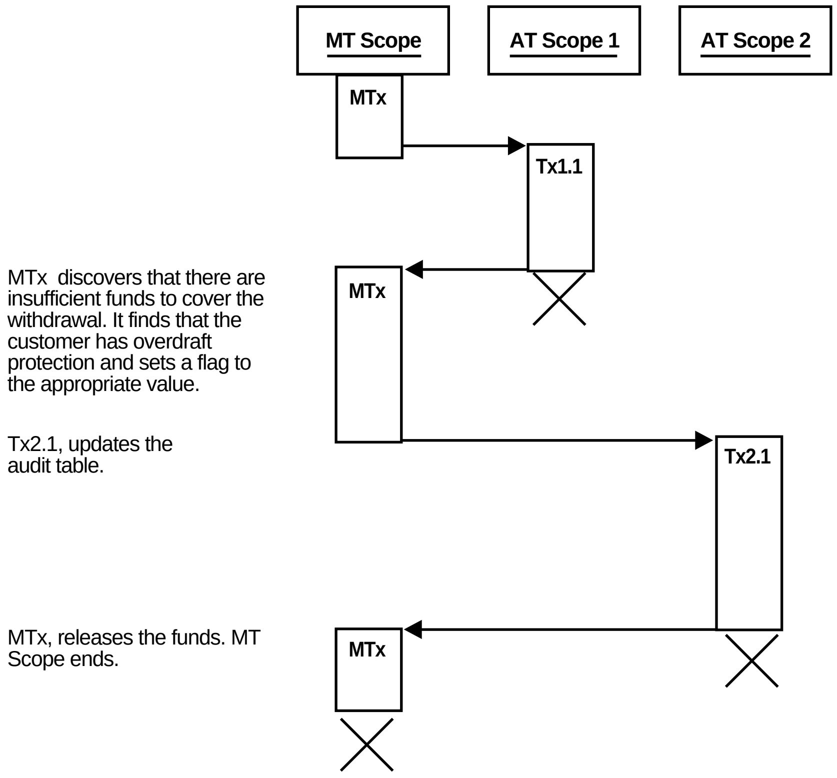
Tx2.1, updates the audit table using the transaction ID generated above, then commits.

MTx releases the funds. MT Scope ends.

Scenario 2:

There are insufficient funds to cover the withdrawal, but the customer has overdraft protection. The bank therefore releases the funds.

Figure 7–8 Example: Bank Withdrawal—Insufficient Funds WITH Overdraft Protection



MTx discovers that there are insufficient funds to cover the withdrawal. It finds that the customer has overdraft protection and sets a flag to the appropriate value.

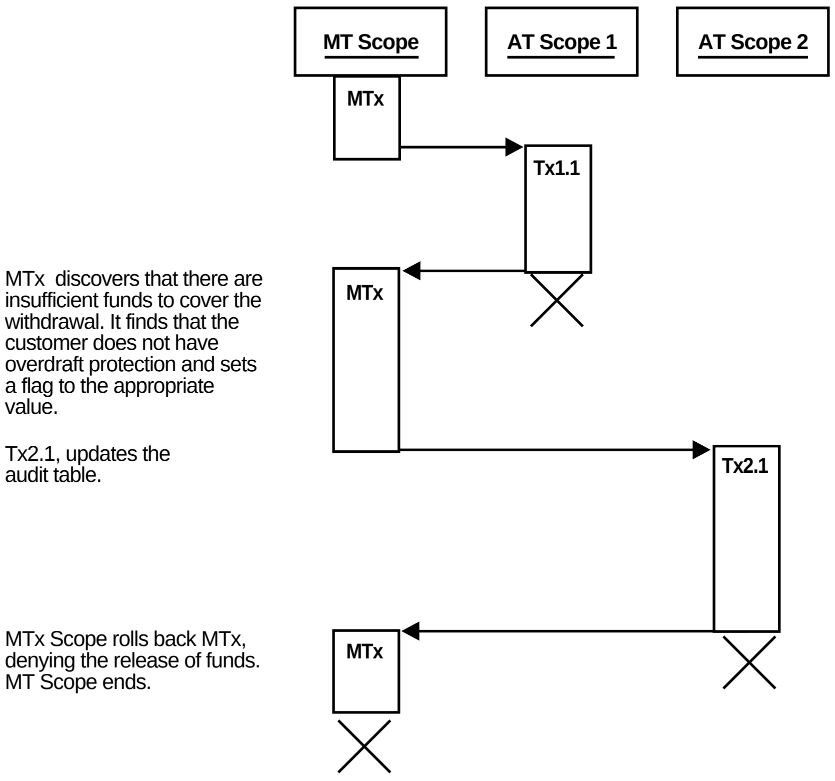
Tx2.1, updates the audit table.

MTx, releases the funds. MT Scope ends.

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.

Figure 7–9 Example: Bank Withdrawal—Insufficient Funds WITHOUT Overdraft Protection


MT Scope
AT Scope 1
AT Scope 2
MTx
Tx1.1
MTx discovers that there are insufficient funds to cover the withdrawal. It finds that the customer does not have overdraft protection and sets a flag to the appropriate value.
MTx
Tx2.1, updates the audit table.
Tx2.1
MTx Scope rolls back MTx, denying the release of funds. MT Scope ends.
MTx

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/or 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/or packages...
END Banking;
```

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;
```

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:

- *Oracle9i Database Concepts*
- *Oracle9i 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. Oracle encourages users to 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 *Oracle9i Supplied 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: *Oracle9i 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;
```

Querying Data at a Point in Time (Flashback Query)

By default, operations on the database use the most recent committed data available. If you want to query the database as it was at some time in the past, you can do so with the flashback query feature. It lets you specify either a time or a system change number (SCN) and query using the committed data from the corresponding time.

Some potential applications of flashback query are:

- Recovering lost data or undoing incorrect changes, even after the changes are committed. For example, a user who deletes or updates rows and then commits can immediately repair a mistake.
- Comparing current data against the data at some time in the past. For example, you might run a weekly report that shows the change from last week, rather than just the current aggregate data.
- Checking the state of transactional data at a particular time. For example, you might want to verify an account balance on a certain day.
- Simplifying application design by removing the need to store some kinds of temporal data.
- Enabling packaged applications, such as report generation tools, to work on past versions of data.

The flashback query mechanism relies on automatic undo management to maintain the necessary undo data. The DBA requests that undo data be kept for a specified period of time. Depending on the available storage capacity, the database might not always be able to keep all the requested undo data. If you use flashback queries, you might need to familiarize yourself with automatic undo management to understand its capabilities and limitations.

Other features are available to recover lost data. The unique feature of flashback query is that you can see the data as it was in the past, then choose exactly how to process the information; you might just want to do an analysis rather than undoing the changes.

See Also:

- *Oracle9i Database Concepts*
- *Oracle9i Database Administrator's Guide*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

Setting Up the Database for Flashback Query

Before you can perform flashback queries, enlist the help of your DBA. Ask them to:

- Use automatic undo management to maintain read consistency, rather than the older technique using rollback segments. In particular, the DBA should:
  - Set the `UNDO_RETENTION` initialization parameter to a value that represents how far in the past you might want to query. The value depends

    on your needs. If you only need to recover data immediately after a mistaken change is committed, the parameter can be set to a small value. If you need to recover deleted data from days before, you might need several worth of data.

  - Set the initialization parameter `UNDO_MANAGEMENT=AUTO.`
  - Create an `UNDO` tablespace, with enough space to keep the required data. The more often the data is updated, the more space is required.
- Grant `EXECUTE` privilege on the `DBMS_FLASHBACK` package to users, roles, or applications that need to perform flashback queries.

Writing an Application that Uses Flashback Query

To use the flashback query feature in an application, use these coding techniques:

- Put calls to the `DBMS_FLASHBACK` package around any queries that apply to data at a past time:
  - Before doing the query, call `DBMS_FLASHBACK.ENABLE_AT_TIME` or `DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER`.
  - After doing the query, call `DBMS_FLASHBACK.DISABLE`.
  - Between these calls, you can only perform queries and not any DDL or DML statements.
- To use the results of a flashback query in DDL or DML statements against the current state of the database, open a cursor before calling `DBMS_FLASHBACK.DISABLE`. You can fetch results from past data from the cursor, then issue `INSERT` or `UPDATE` statements against the current state of the database.
- To compare current data against past data, you can open a cursor with the flashback feature enabled, then disable it and open another cursor. Fetching from the first cursor retrieves data based on the flashback time; fetching from the second cursor retrieves current data. You can store the older data in a temporary table and then use set operators such as `MINUS` or `UNION` to show differences in the data or combine past and current data.
- At certain points in the application, you might call `DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER`. You can store the returned number, and later use it to perform flashback queries against the data at that point in time. Perform a `COMMIT` before calling this procedure, so that the database is in a consistent state to which you can return.

Limitations of Flashback Query

- Some DDLs that alter the structure of a table, such as drop / modify column, move table, drop partition, truncate table / partition, and so on, invalidate the old undo data for the table. It is not possible to retrieve a snapshot of data from a point earlier than the time such DDLs were executed. An attempt to perform such a query will result in a ORA-1466 error. This restriction does not apply to DDL operations that alter the storage attributes of a table, such as `PCTFREE`, `INITTRANS`, `MAXTRANS`, and so on. Operations such as adding new extents, constraints or partitions are also exempted from this restriction.
- The time specified in `DBMS_RESUMABLE.ENABLE_AT_TIME` is mapped to an SCN value. Currently, the SCN-time mapping is recorded every 5 minutes after database startup. Thus it might appear as if the specified time is being rounded down by up to 5 minutes.

  For example, assume that the SCN values 1000 and 1005 are mapped to the times 8:41 and 8:46 AM respectively. A flashback query for a time anywhere between 8:41:00 and 8:45:59 AM is mapped to SCN 1000; a flashback query for 8:45 AM is mapped to SCN 1005.

  Due to this time-to-SCN mapping, a flashback query for a time immediately after creation of a table may result in an ORA-1466 error. An SCN-based flashback query therefore gives you a more precise way to retrieve a past snapshot of data.
- Because SCNs are only recorded every 5 minutes for use by flashback queries, you might specify a time or SCN that is slightly after a DDL operation, but the database might use a slightly earlier SCN that is before the DDL operation. So the previous restriction might also apply if you try to perform flashback queries to a point just after a DDL operation.
- Currently, the flashback query feature keeps track of times up to a maximum of 5 days. This period reflects server uptime, not wall-clock time. For example, if the server is down for a day during this period, then you can specify as far back as 6 days. To query data farther back than this, you must specify an SCN rather than a date and time. You must record the SCN yourself at the time of interest, such as before doing a `DELETE`.
- You must disable flashback before enabling it again for a different time. You cannot nest `ENABLE` /`DISABLE` pairs. You can call `DISABLE` multiple times in succession, although this introduces a little extra overhead, as shown in the following examples.

- Only the state of table data is affected by a flashback query. During a query, the current state of the data dictionary is used.
- A flashback query might fail if it uses an index that is created or rebuilt after the flashback time or SCN. In this case, you can supply a query hint to perform a full table scan:

  ```
  SELECT /*+ FULL(employees) */ * FROM employees;
  ```

- You cannot perform a flashback query on a remote table through a database link.

Examples of Flashback Query

The flashback query mechanism is flexible enough to be used in many situations. You can:

- Query data as it existed in the past.
- Compare current data to past data. You can compare individual rows, or do more complex comparisons such as finding the intersection or union.
- Recover deleted or changed data.
- Refer to a point in time using a DATE value, or record system change numbers.

Retrieving Data in the Past: Example

To find out someone’s salary at year-end 2000 through a flashback query:

```
EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME('01-JAN-2001');
SELECT salary FROM employee WHERE empid = 41863;
EXECUTE DBMS_FLASHBACK.DISABLE;
```

Remember that a date with no time represents the very beginning of that day. Because of the limited amount of mapping data that is stored for times, such a query might only be able to look back a few days in the past. To look back farther, you need to store the SCN at the time of interest. Even then, the data might be unavailable if the saved undo data does not extend back that far.

Recovering Incorrectly Updated or Deleted Data: Examples

This example makes an incorrect update, commits the changes, then immediately recovers the old information using a flashback query:

```
UPDATE employee SET salary = salary - 1000;
DELETE FROM employee WHERE salary =
```

```
COMMIT;
-- Those updates and deletes were in error. We need to recover the data.
-- Use the data as it existed approximately 15 minutes ago.
-- This requires confidence that the data has not changed during that interval.
EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE - (15/(24*60));
FOR item IN (SELECT * FROM employee WHERE ...)
LOOP
-- Within the loop we can see the old data. If we want to put it back
-- into the current table, we must disable flashback query inside
-- the loop.
  ...
END LOOP;
```

A more reliable method of specifying the flashback point uses the SCN directly:

```
DECLARE
  old_scn NUMBER := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
BEGIN
  UPDATE employee SET salary = salary - 1000;
  DELETE FROM employee WHERE salary = 60000;
  COMMIT;
-- Those updates and deletes were in error. We need to recover the data.
-- Use the data as it existed immediately before the update and delete.
  EXECUTE DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER(old_scn);
  FOR item IN (SELECT * FROM employee WHERE ...)
  LOOP
    ...
  END LOOP;
```

The following example illustrates how this can be done for a case where the deletion of a senior employee triggers the deletion of all the reports under him. Using flashback query, we can recover and re-insert the missing employees.

```
rem keep_scn is a temporary table to store scns that we are interested in.
create table keep_scn (scn number);
set echo on
create table employee (
  employee_no   number(5) primary key,
  employee_name varchar2(20),
  employee_mgr  number(5)
     constraint mgr_fkey references employee on delete cascade,
  salary        number,
  hiredate      date
);
```

```
rem Now populate the company with employees.
insert into employee values (1, 'Dennis Potter', null, 1000000, '5-jul-91');
insert into employee values (10, 'Margaret O'Neil', 1, 500000, '12-aug-94');
insert into employee values (20, 'Charles Evans', 10, 250000, '13-dec-97');
insert into employee values (100, 'Roger Smith', 20, 200000, '3-feb-96');
insert into employee values (200, 'Juan Hernandez', 100, 150000, '22-mar-98');
insert into employee values (210, 'Jonathan Takeda', 100, 100000, '11-apr-97');
insert into employee values (220, 'Nancy Schoenfeld', 100, 100000, '18-sep-95');
insert into employee values (300, 'Edward Ngai', 210, 75000, '4-nov-96');
insert into employee values (310, 'Amit Sharma', 210, 65000, '3-may-95');
commit;

rem Show the entire org
select lpad(' ', 2*(level-1)) || employee_name Name
  from employee
  connect by prior employee_no = employee_mgr
  start with employee_no = 1
  order by level;

execute dbms_flashback.disable;
rem Store this snapshot for later access through FlashBack.
declare
I number;
begin
I := dbms_flashback.get_system_change_number;
insert into keep_scn values (I);
commit;

rem Now Roger decides it’s time to retire but the HR department does
rem the transaction incorrectly

delete from employee where employee_name = 'Roger Smith';
commit;

rem Notice that all of Roger's employees are now gone.
select lpad(' ', 2*(level-1)) || employee_name Name
  from employee
  connect by prior employee_no = employee_mgr
  start with employee_no = 1
  order by level;

rem Well, lets put back Roger's organization now.
declare
  restore_scn number;
begin
```

```
  select  scn into restore_scn from keep_scn;
  dbms_flashback.enable_at_system_change_number (restore_scn);
end;
/

rem First show Roger's org.
select lpad(' ', 2*(level-1)) || employee_name Name
  from employee
  connect by prior employee_no = employee_mgr
  start with employee_no =
    (select employee_no from employee where employee_name = 'Roger Smith')
  order by level;

declare
  rogers_emp number;
  rogers_mgr number;
  cursor c1 is
   select employee_no, employee_name, employee_mgr, salary, hiredate
     from employee
     connect by prior employee_no = employee_mgr
     start with employee_no =
       (select employee_no from employee where employee_name = 'Roger Smith');
   c1_rec is c1 % ROWTYPE;
begin
  select employee_no, employee_mgr into rogers_emp, rogers_mgr from employee
    where employee_name = 'Roger Smith';
rem Open c1 with FlashBack enabled.
 open c1;
rem Disable FlashBack now.
dbms_flashback.disable;
loop
rem Note that all the DML operations inside the loop are performed
rem with FlashBack disabled.
 fetch c1 into c1_rec;
exit when c1%NOTFOUND;
  for c1_rec in c1 loop
    if (c1_rec.employee_mgr = rogers_emp) then
     insert into employee values (c1_rec.employee_no,
                                  c1_rec.employee_name,
                                  rogers_mgr,
                                  c1_rec.salary,
                                  c1_rec.hiredate);
    else
      if (c1_rec.employee_no != rogers_emp) then
         insert into employee values (c1_rec.employee_no,
                                      c1_rec.employee_name,
                                      c1_rec.employee_mgr,
                                      c1_rec.salary,
                                      c1_rec.hiredate);
    end if;
  end if;
end loop;
end;
/

execute dbms_flashback. disable;
```

Comparing Current Data to Past Data: Example

To compare today's data with yesterday's using a flashback query:

```
DECLARE
  old_value NUMBER;
  new_value NUMBER;
BEGIN
  EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE - 1);
  SELECT ... INTO old_value FROM ...;
  DBMS_FLASHBACK.DISABLE;
  SELECT ... INTO new_value FROM ...;
  DBMS_OUTPUT.PUT_LINE('Value from one day ago: ' || old_value);
  DBMS_OUTPUT.PUT_LINE('Value from today: ' || new_value);
  DBMS_OUTPUT.PUT_LINE('Percentage change over past day: ' ||
    (old_value / (new_value - old_value)) * 100);
END;
```

Storing a System Change Number: Example

To store a system change number so that it can be used later for flashback queries:

```
DECLARE
  scn NUMBER;
BEGIN
-- We make a set of updates so that the data is at a known state.
  INSERT ...;
  UPDATE ...;
  DELETE ...;
  COMMIT;
-- We can use this saved SCN to return to this known state later.
  scn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
  INSERT INTO ... VALUES (scn, SYSDATE, 'State of the system after ...');
END;
```

Using Explicit and Implicit Cursors with Flashback Queries: Example

Although you can use an implicit cursor loop and call `DBMS_FLASHBACK.DISABLE` multiple times within the loop body, doing so results in some extra overhead. It is more efficient to use an explicit cursor:

```
-- Most efficient technique. Open the cursor, disable flashback, then use
-- the older data in DML statements on the current table.
EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE - 1);
OPEN c FOR 'SELECT * FROM employees WHERE ...';
DBMS_FLASHBACK.DISABLE;
LOOP
   FETCH ...;
   EXIT WHEN c%NOTFOUND;
   INSERT ...;
END LOOP;
```

You can also use an implicit cursor, although this is slightly less efficient:

```
-- Less efficient technique. To allow DML statements against the current
-- table within the loop body, DISABLE must be called for each loop iteration.
EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE - 1);
-- The FOR loop is examining data values from the past.
FOR c in (SELECT * FROM employees WHERE ...')
LOOP
   DBMS_FLASHBACK.DISABLE;
-- Because flashback is disabled within the loop body, we can access the
-- present state of the data, and issue DML statements to undo the changes
-- or store the old data.
   INSERT ...;
END LOOP;
```

8 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 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:

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: *Oracle9i 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 8-17, and see the query examples in "A Dynamic SQL Scenario Using Native Dynamic SQL" on page 8-8.

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 8-2, 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 and/or concatenates 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: *Oracle9i Database Performance Guide and Reference* 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 ythroughthroughou 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-Rights

By using the invoker-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-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 invokers-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 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 *Oracle9i Database Migration* 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 and the *Oracle9i Supplied PL/SQL Packages and Types Reference* for detailed information about using the DBMS_SQL package. In the *PL/SQL User's Guide and Reference*, native dynamic SQL is referred to simply as dynamic SQL.

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 8–1 illustrates the difference in the amount of code required to perform the same operation using the DBMS_SQL package and native dynamic SQL.

Table 8–1 Code Comparison of DBMS_SQL Package and Native Dynamic SQL

DBMS_SQL Package	Native Dynamic SQL
`CREATE PROCEDURE insert_into_table (`	`CREATE PROCEDURE insert_into_table (`
`      table_name  VARCHAR2,`	`      table_name  VARCHAR2,`
`      deptnumber  NUMBER,`	`      deptnumber  NUMBER,`
`      deptname    VARCHAR2,`	`      deptname    VARCHAR2,`
`      location    VARCHAR2) IS`	`      location    VARCHAR2) IS`
`   cur_hdl         INTEGER;`	`   stmt_str    VARCHAR2(200);`
`   stmt_str        VARCHAR2(200);`	
`   rows_processed  BINARY_INTEGER;`	`BEGIN`
	`   stmt_str := ’INSERT INTO ’ ||`
`BEGIN`	`      table_name || ’ values`
`   stmt_str := 'INSERT INTO ' ||`	`      (:deptno, :dname, :loc)’;`
`      table_name || ' VALUES`	
`      (:deptno, :dname, :loc)';`	`   EXECUTE IMMEDIATE stmt_str`
	`      USING`
`   -- open cursor`	`      deptnumber, deptname, location;`
`   cur_hdl := dbms_sql.open_cursor;`	
	`END;`
`   -- parse cursor`	`/`
`   dbms_sql.parse(cur_hdl, stmt_str,`	`SHOW ERRORS;`
`      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 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 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 *Oracle9i Supplied 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.

DBMS_SQL Supports Multiple Row Updates and Deletes with a RETURNING Clause

The DBMS_SQL package supports statements with a RETURNING clause that update or delete multiple rows. Native dynamic SQL only supports a RETURNING clause if a single row is returned.

See Also: "Performing DML with RETURNING Clause Using Dynamic SQL: Example" on page 8-20 for examples of DBMS_SQL package code and native dynamic SQL code that uses a RETURNING clause.

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 8–2 shows sample code that accomplishes this query using the `DBMS_SQL` package and native dynamic SQL.

Table 8–2 Querying Using the DBMS_SQL Package and Native Dynamic SQL

DBMS_SQL Query Operation	Native Dynamic SQL Query Operation
`DECLARE`	`DECLARE`
`  stmt_str varchar2(200);`	`  TYPE EmpCurTyp IS REF CURSOR;`
`  cur_hdl int;`	`  cur EmpCurTyp;`
`  rows_processed int;`	`  stmt_str VARCHAR2(200);`
`  name varchar2(10);`	`  name VARCHAR2(20);`
`  salary int;`	`  salary NUMBER;`
`BEGIN`	`BEGIN`
`cur_hdl := dbms_sql.open_cursor; -- open cursor`	`  stmt_str := ’SELECT ename, sal FROM emp`
`stmt_str := ’SELECT ename, sal FROM emp WHERE`	`    WHERE job = :1’;`
`job = :jobname’;`	`  OPEN cur FOR stmt_str USING ’SALESMAN’;`
`dbms_sql.parse(cur_hdl, stmt_str,`	
`dbms_sql.native);`	`LOOP`
	`  FETCH cur INTO name, salary;`
`-- supply binds (bind by name)`	`  EXIT WHEN cur%NOTFOUND;`
`dbms_sql.bind_variable(`	`  -- <process data>`
`  cur_hdl, ’jobname’, ’SALESMAN’);`	`END LOOP;`
	`CLOSE cur;`
`-- describe defines`	`END;`
`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;`	
`/`	

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 8–3 shows sample code that accomplishes this DML operation using the DBMS_SQL package and native dynamic SQL.

Table 8–3 DML Operation Using the DBMS_SQL Package and Native Dynamic SQL

DBMS_SQL DML Operation	Native Dynamic SQL DML Operation
`DECLARE`	`DECLARE`
`  stmt_str VARCHAR2(200);`	`  stmt_str VARCHAR2(200);`
`  cur_hdl NUMBER;`	`  deptnumber NUMBER := 99;`
`  deptnumber NUMBER := 99;`	`  deptname VARCHAR2(20);`
`  deptname VARCHAR2(20);`	`  location VARCHAR2(10);`
`  location VARCHAR2(10);`	`BEGIN`
`  rows_processed NUMBER;`	`  stmt_str := 'INSERT INTO dept_new VALUES`
`BEGIN`	`  (:deptno, :dname, :loc)';`
`  stmt_str := 'INSERT INTO dept_new VALUES`	`  EXECUTE IMMEDIATE stmt_str`
`    (:deptno, :dname, :loc)';`	`    USING deptnumber, deptname, location;`
`  cur_hdl := DBMS_SQL.OPEN_CURSOR;`	`END;`
`  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;`	
`/`	

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 8–4 shows sample code that accomplishes this operation using both the DBMS_SQL package and native dynamic SQL.

Table 8–4 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 roundtrips 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 stored procedures and stored functions from various languages, refer to:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*
- *Pro*COBOL Precompiler Programmer's Guide*
- *Oracle9i Java Stored Procedures Developer's Guide*

9
Using Procedures and Packages

This chapter describes some of the procedural capabilities of Oracle for application development, including:

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 the 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.

Oracle9i Supplied PL/SQL Packages and Types Reference. for information about the PL/SQL packages that come with the Oracle database server.

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: For complete information about the DBMS_OUTPUT package, see *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Exceptions let you handle Oracle error conditions within PL/SQL program logic. This allows your application to prevent the server from issuing an error that could cause the client application to abend. The following anonymous block handles the predefined Oracle exception NO_DATA_FOUND (which would result in an ORA-01403 error if not handled):

```
DECLARE
   Emp_number   INTEGER := 9999;
   Emp_name     VARCHAR2(10);
BEGIN
   SELECT Ename INTO Emp_name FROM Emp_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: "Handling Run-Time PL/SQL Errors" on page 9-37 and see the *PL/SQL User's Guide and Reference.*

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 9-32.

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 9-2.

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 parts:

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 9–1 summarizes the information about parameter modes.

See Also: Parameter modes are explained in detail in the *PL/SQL User's Guide and Reference.*

Table 9–1 Parameter Modes

IN	OUT	IN OUT
The default.	Must be specified.	Must be specified.
Passes values to a subprogram.	Returns values to the caller.	Passes initial values to a subprogram; returns updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an initialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression; must be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, initialized variable, literal, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.

Parameter Datatypes The datatype of a formal parameter consists of one of the following:

- An **unconstrained** type name, such as `NUMBER` or `VARCHAR2`.
- A type that is **constrained** using the `%TYPE` or `%ROWTYPE` attributes.

Note: Numerically constrained types such as `NUMBER(2)` or `VARCHAR2(20)` are not allowed in a parameter list.

%TYPE and %ROWTYPE Attributes 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 9-5 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 9-7, 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: *Oracle9i 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 a procedure's or a package's owner 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 procedure's owner. 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 9-45.

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 recreates it as specified.

Caution: The procedure or function is dropped without any warning.

Controlling the Behavior of PL/SQL Procedures and Functions

You can control some runtime behavior for PL/SQL procedures and functions by setting parameters within the database. These parameters can apply to all PL/SQL procedures and functions, or to a particular procedure or function. For example:

```
-- Set default behavior for PL/SQL procedures and functions
ALTER SESSION SET PLSQL_V2_COMPATIBILITY = TRUE;
-- Use a different setting for this one procedure
ALTER PROCEDURE myproc SET PLSQL_V2_COMPATIBILITY = FALSE;
```

The parameters can apply to procedures, functions, packages, types, and triggers. Once specified, the settings apply whenever these schema objects are updated by CREATE OR REPLACE or the automatic recompilation that happens when the object is invalidated. When a schema object is dropped, any settings that apply to only that object are lost.

You can find out what settings are in effect by querying the catalog views ALL_PLSQL_SWITCH_SETTINGS and USER_PLSQL_SWITCH_SETTINGS. You can find out all the possible setting names and parameters by querying the view ALL_PLSQL_SWITCHES. Within an application, you can also find this information by calling functions in the DBMS_DESCRIBE package.

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 a package's specification and body.

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 Server can call an external procedure written in a 3GL. The 3GL procedure runs in a separate address space from that of the Oracle Server.

> **See Also:** For information about external procedures, see the Chapter 10, "Calling 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 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:** The *PL/SQL User's Guide and Reference* has 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 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 9-10 and "Privileges to Drop Procedures and Functions" on page 9-12.

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 subsequently invalidated and recompiled, then all other dependent package instantiations (including state) for the session are lost.

For example, assume that session `S` instantiates packages `P1` and `P2`, and that a procedure in package `P1` calls a procedure in package `P2`. If `P1` is invalidated and recompiled (for example, as the result of a DDL operation), then the session `S` instantiations of both `P1` and `P2` are lost. In such situations, a session receives the following error the first time it attempts to use any object of an invalidated package instantiation:

```
ORA-04068: existing state of packages has been discarded
```

The second time a session makes such a package call, the package is reinstantiated for the session without error.

Note: Oracle has been optimized to not return this message to the session calling the package that it invalidated. Thus, in the example above, session `S` receives this message the first time it called package `P2`, but it does not receive it when calling `P1`.

In most production environments, DDL operations that can cause invalidations are usually performed during inactive working hours; therefore, this situation might not be a problem for end-user applications. However, if package specification or body invalidations are common in your system during working hours, then you might want to code your applications to detect for this error when package calls are made.

Oracle Supplied Packages

There are many built-in packages provided with the Oracle Server, either to extend the functionality of the database or to give PL/SQL access to SQL features. You may take advantage of the functionality provided by these packages when creating your application, or you may simply want to use these packages for ideas in creating your own stored procedures.

These packages run as the calling user, rather than the package owner. Unless otherwise noted, the packages are callable through public synonyms of the same name.

For details on all of these Oracle-supplied packages, see:

- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *Oracle Spatial User's Guide and Reference*
- *Oracle Time Series User's Guide*

Overview of Bulk Binds

Oracle 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 15, "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: For complete instructions on using the PL/SQL Wrapper, see the *PL/SQL User's Guide and Reference*.

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 process.

You can use this technique with both the supplied Oracle 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.

See Also: For full details, see the Tuning chapter of the *PL/SQL User's Guide and Reference*.

With Java, you can use the `ncomp` tool to compile your own packages and classes. See Java Developer's Guide, Java Tools Reference.

Remote Dependencies

Dependencies among PL/SQL program units can be handled in two ways:

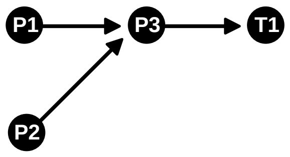
- Timestamps
- Signatures

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 9–1 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 9–1 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 Server 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 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 9-29.

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.

Now, assume that on another server in California, some PL/SQL code calls `get_emp_name` identifying it using a DBlink called `BOSTON_SERVER`, as follows:

```
CREATE OR REPLACE PROCEDURE print_ename (emp_number IN NUMBER) AS
   hire_date    VARCHAR2(12);
   ename        VARCHAR2(10);
BEGIN
   get_emp_name@BOSTON_SERVER(emp_number, hire_date, ename);
   dbms_output.put_line(ename);
   dbms_output.put_line(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?

Datatypes 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 table below, 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, you change

```
PROCEDURE P1 (Param1 NUMBER);
```

to

```
PROCEDURE P1 (Param1 IN NUMBER);
```

This 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 9-5, 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;
```

Then, the specification of the procedure has not changed, and, therefore, 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, then the signature does *not* change. Changing the *name* of a formal parameter does not change the signature of the unit.

Consider the following example:

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_type IS RECORD (
        Emp_number NUMBER,
        Hire_date  VARCHAR2(12),
        Emp_name   VARCHAR2(10));
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type);
```

```
END;

CREATE OR REPLACE PACKAGE BODY Emp_package AS
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type) IS
   BEGIN
       SELECT Empno, Ename, TO_CHAR(Hiredate, 'DD/MON/YY')
           INTO Emp_data
           FROM Emp_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:

  ```
  ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =
      {SIGNATURE | TIMESTAMP}
  ```

  The above example alters the dependency model for the current session.

  ```
  ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =
      {SIGNATURE | TIMESTAMP}
  ```

  The above example alters the dependency model on a system-wide basis after startup.

If the REMOTE_DEPENDENCIES_MODE parameter is not specified, either in the init.ora parameter file or using the ALTER SESSION or ALTER SYSTEM DDL 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, you should be aware of the following:

- 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 9-26), then an error is returned to the calling session.

Suggestions for Managing Dependencies

Oracle recommends that you follow these guidelines for setting the `REMOTE_DEPENDENCIES_MODE` parameter:

- Server-side PL/SQL users can set the parameter to `TIMESTAMP` (or let it default to that) to get the timestamp dependency mode.
- Server-side PL/SQL users can choose to use the signature dependency mode if they have a distributed system and 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: The *PL/SQL User's Guide and Reference* has a complete discussion of 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++ Precompiler Programmer's Guide*
- *Pro*COBOL Precompiler Programmer's Guide*
- *Oracle Call Interface Programmer's Guide*
- *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: See the *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 CHARWIDTH statement to get long lines on output. The value 132 is usually a good choice. For example:

```
SET CHARWIDTH 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:** *Oracle9i Database Reference* for more information about these data dictionary views.

Handling Run-Time PL/SQL Errors

Oracle allows user-defined errors in PL/SQL code to be handled so that user-specified error numbers and messages are returned to the client application. After received, the client application can handle the error based on the user-specified error number and message returned by Oracle.

User-specified error messages are returned using the RAISE_APPLICATION_ERROR procedure. 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-supplied packages, such as DBMS_OUTPUT, DBMS_DESCRIBE, and DBMS_ALERT, use application error numbers in the range -20000 to -20005. See the descriptions of these packages for more information.

The RAISE_APPLICATION_ERROR procedure is often used in exception handlers or in the logic of PL/SQL code. For example, the following exception handler selects the string for the associated user-defined error message and calls the RAISE_APPLICATION_ERROR procedure:

```
...
WHEN NO_DATA_FOUND THEN
   SELECT Error_string INTO Message
   FROM Error_table,
   V$NLS_PARAMETERS V
   WHERE Error_number = -20101 AND Lang = v.value AND
      v.parameter = "NLS_LANGUAGE";
   Raise_application_error(-20101, Message);
...
```

> **See Also:** For information on exception handling when calling remote procedures, see"Handling Errors in Remote Procedures" on page 9-40.

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.

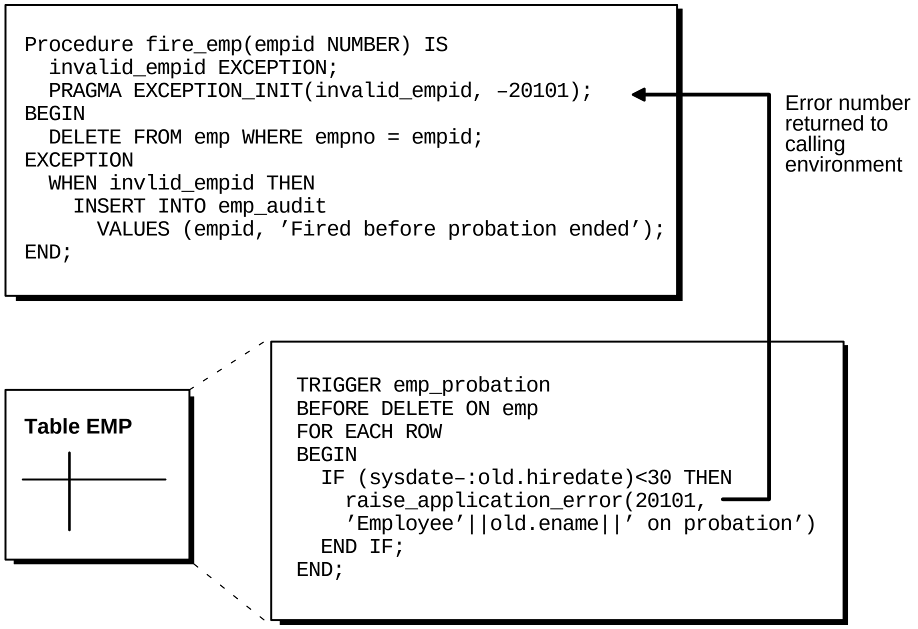
You can also define an exception handler to handle user-specified error messages. For example, Figure 9–2 on page 9-39 illustrates the following:

- 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 9–2 Exceptions and User-Defined Errors



```
Procedure fire_emp(empid NUMBER) IS
  invalid_empid EXCEPTION;
  PRAGMA EXCEPTION_INIT(invalid_empid, –20101);
BEGIN
  DELETE FROM emp WHERE empno = empid;
EXCEPTION
  WHEN invlid_empid THEN
    INSERT INTO emp_audit
      VALUES (empid, ’Fired before probation ended’);
END;
```

```
TRIGGER emp_probation
BEFORE DELETE ON emp
FOR EACH ROW
BEGIN
  IF (sysdate–:old.hiredate)<30 THEN
    raise_application_error(20101,
    ’Employee’||old.ename||’ on probation’)
  END IF;
END;
```

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 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 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

The DBMS_DEBUG API, provided starting with Oracle8*i*, implements server-side debuggers, and it provides a way to debug server-side PL/SQL program units. Several of the debuggers currently available, such as Oracle Procedure Builder and various third-party vendor solutions, use this API.

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.

See Also: *Oracle Procedure Builder Developer's Guide*

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.

See Also: See *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information about the `DBMS_DEBUG` and the `DBMS_OUTPUT` packages.

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 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 9-49.

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 Tools

A procedure can be called interactively from an Oracle 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: See the *SQL*Plus User's Guide and Reference* for SQL*Plus information. See 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 more information about calling PL/SQL procedures from within 3GL applications, see the following manuals:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide,*
- *SQL*Module for Ada Programmer's Guide*

Name Resolution When 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, "Managing 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 owner's name in the call. For example:

> **Note:** 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
> ```

```
EXECUTE Jward.Fire_emp (1043);

EXECUTE Jward.Hire_fire.Fire_emp (1043);
```

> **Note:** A stored subprogram or package runs in the privilege domain of the owner of the procedure. The owner must be explicitly granted the necessary object privileges to all objects referenced within the body of the code.

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.

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's 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:** For information on exception handling when calling remote procedures, see "Handling Errors in Remote Procedures" on page 9-40.

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 list below explains how to properly call remote procedures, depending on the calling environment.

- Remote procedures (standalone and packaged) can be called from within a procedure, 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 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 9-49

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 proceeds as follows:

- Oracle first checks for the Payroll package in the current schema.

- If the PAYROLL package is found in the current schema, then Oracle 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 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 currently 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.

Meeting Basic Requirements

To be callable from SQL expressions, a user-defined PL/SQL function must meet the following basic requirements:

- It must be a stored function, *not* a function defined within a PL/SQL block or subprogram.
- It must be a row function, *not* a column (group) function; 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 Server internal types, such as `CHAR`, `DATE`, or `NUMBER`, *not* PL/SQL types, such as `BOOLEAN`, `RECORD`, or `TABLE`.
- Its return type (the datatype of its result value) must be an Oracle Server internal type.

For example, the following stored function meets the basic requirements:

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 function refers to the side effects of that function 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 function is called from a SQL query or DML statement.

In previous releases, Oracle leveraged the PL/SQL compiler to enforce restrictions during the compilation of a stored function or a SQL statement. Starting in Oracle8*i*, the compile-time restrictions have been relaxed, and a smaller set of restrictions are enforced during execution.

See Also: "Restrictions" on page 9-55.

This change provides uniform support for stored functions written in PL/SQL, Java, and C, and it allows programmers the most flexibility possible.

PL/SQL Compilation Checking

A user-written function can now be called from a SQL statement *without* any compile-time checking of its purity: PRAGMA RESTRICT_REFERENCES is no longer required on functions called from SQL statements.

PRAGMA RESTRICT_REFERENCES remains available as a means of asking the PL/SQL compiler to verify that a function has only the side effects that you expect. SQL statements, package variable accesses, or calls to functions that violate the declared restrictions will continue to raise PL/SQL compilation errors to help you isolate the code that has unintended effects.

Because Oracle no longer requires that the pragma on functions called from SQL statements, different applications may choose different style standards on whether and where to use PRAGMA RESTRICT REFERENCES. An existing PL/SQL application will most likely want to continue using the pragma even on new functionality, to ease integration with the existing code. A newly created Java application will most likely not want to use the pragma at all, because the Java compiler does not have the functionality to assist in isolating unintended effects.

> **See Also:** "Using PRAGMA RESTRICT_REFERENCES" on page 9-59.

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 function 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:

- A function 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 function called from a query (SELECT) statement or from a parallelized DML statement may not execute a DML statement or otherwise modify the database.
- A function 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 function or trigger. For example:

- They apply to a SQL statement called from PL/SQL, whether embedded directly in a function or trigger body, run using the new 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 new autonomous transactions provide one escape. Another escape is available using 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 meaningfully 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's declaration in a CREATE PACKAGE statement, or on a method's 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. However, the database has no reasonable way to recognize if the function's behavior indeed is truly 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.

Two relatively new features require that any function used with them is declared DETERMINISTIC.

- Any function used in a function-based index is required to be DETERMINISTIC.
- Any function used in a materialized view must be DETERMINISTIC if that view is to be 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.

It is also preferable that only functions declared DETERMINISTIC are used in any materialized view or snapshot that is declared REFRESH FAST. Oracle allows in REFRESH FAST snapshots those functions that have a PRAGMA RESTRICT_REFERENCES noting that they are RNDS, and those PL/SQL functions defined with a CREATE FUNCTION statement whose code can be examined to determine that they do not read the database nor call any other routine which might, as these have been allowed historically.

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 above. Oracle 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'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 cannot assume that it is safe to parallelize the execution of all user-defined functions.

For query (SELECT) statements, in previous releases, 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 actually pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

See Also: "Using PRAGMA RESTRICT_REFERENCES" on page 9-59.

For DML statements, in previous releases, 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.

In Oracle9*i*, we continue to parallelize those functions that earlier releases would recognize as parallelizable. The PARALLEL_ENABLE keyword is the preferred way now to mark your code as safe for parallel execution. This keyword is syntactically similar to DETERMINISTIC as described above; 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’s declaration in a CREATE PACKAGE statement, or on a method’s 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 9-55.

Using PRAGMA RESTRICT_REFERENCES

To assert the purity level, code the pragma RESTRICT_REFERENCES in the package specification (not in the package body). The pragma must follow the function declaration, but it does not need to follow it immediately. Only one pragma can reference a given function declaration.

To code the pragma RESTRICT_REFERENCES, use the following syntax:

```
PRAGMA RESTRICT_REFERENCES (
    Function_name, WNDS [, WNPS] [, RNDS] [, RNPS] [, TRUST] );
```

Where:

WNDS	Writes no database state (does not modify database tables).
RNDS	Reads no database state (does not query database tables).
WNPS	Writes no package state (does not change the values of packaged variables).
RNPS	Reads no package state (does not reference the values of packaged variables).
TRUST	Allows easy calling from functions that do have RESTRICT_REFERENCES declarations to those that do not.

You can pass the arguments in any order. If any SQL statement inside the function body violates a rule, then you get an error when the statement is parsed.

In the example below, 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 function allows. That way, the PL/SQL compiler never rejects the function unnecessarily.

Note: You may need to set up the following data structures for certain examples 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 trusted 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 function. 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 function 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's executed dynamically, but it does *not* violate RNDS if it's 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 foregoing function declaration.

In the following example, the pragma applies to the second declaration of valid:

```
CREATE PACKAGE Tests AS
    FUNCTION Valid (x NUMBER) RETURN CHAR;
    FUNCTION Valid (x DATE) RETURN CHAR;
    PRAGMA RESTRICT_REFERENCES (valid, WNDS);
 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's **state** includes global variables, cursors, and so on.

The state of a serially reusable package persists only for the lifetime of a call to the server. On a subsequent call to the server, if a reference is made to the serially reusable package, then Oracle creates a new **instantiation** (described below) of the serially reusable package and initializes all the global variables to NULL or to the default values provided. Any changes made to the serially reusable package state in the previous calls to the server are not visible.

Note: Creating a new instantiation of a serially reusable package on a call to the server does not necessarily imply that Oracle allocates memory or configures the instantiation object. Oracle simply looks for an available instantiation work area (which is allocated and configured) for this package in a least-recently used (LRU) pool in SGA.

At the end of the call to the server, this work area is returned back to the LRU pool. The reason for keeping the pool in the SGA is that the work area can be reused across users who have requests for the same package.

Why Serially Reusable Packages?

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 within triggers. If you attempt to access a serially reusable package from a trigger, then Oracle issues the error message "cannot access Serially Reusable package <*string*> in the context of a trigger."

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;
```

The above program prints:

```
5
```

> **Note:** If the package had not had the pragma SERIALLY_REUSABLE, then 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;
   -- now 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 Across 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. In the past, such transformations required either significant memory overhead, or storing the data in tables between each stage of the transformation.

A low-overhead way to perform such transformations is to use PL/SQL table functions. 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: *Oracle9i Data Cartridge Developer's Guide* for complete details of this process and the requirements for the member functions.

10

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 lets you work in different languages:

- **PL/SQL**, as described in th*e 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++ Precompiler Programmer's Guide*
- **COBOL**, by means of the Pro*COBOL precompiler, as described in the *Pro*COBOL Precompiler Programmer's Guide*
- **Visual Basic**, by means of Oracle Objects For OLE (OO4O), as described in the *Oracle Objects for OLE/ActiveX Programmer's Guide*
- **Java**, by means of the JDBC Application Programmers Interface (API), as described in *Oracle8 Database Programming with Java*.

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:

- 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 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.

Until Oracle 8.0, the Oracle RDBMS supported SQL and the stored procedure language PL/SQL. In Oracle 8.0, PL/SQL introduced **external procedures**, which allowed the capability of writing C functions as PL/SQL bodies. These C functions are callable from PL/SQL and SQL (via PL/SQL). With 8.1, Oracle 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:

- Move computation-bound programs from client to server where they execute faster (because they avoid the roundtrips entailed in across-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

In previous releases, you published an external procedure to Oracle through an `AS EXTERNAL` clause in a PL/SQL wrapper. This wrapper defined the mapping to, and allowed the calling of, external C procedures. The current way to publish external procedures is 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, as before, but also Java class methods.

> **Note:** Call specifications also allow you to publish with the `AS EXTERNAL` clause, introduced in Oracle 8.0. For new applications, however, you should use the `AS LANGUAGE` clause.

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.

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

For help in creating a DLL, look in the RDBMS subdirectory `/public`, where a template `makefile` can be found.

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 `BFILE`s 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';
```

Now, 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.

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

Loading External C Procedures

In order to set up to use external procedures written in C, or callable by C, you and your DBA take the following steps:

Note: This feature is available only on platforms that support DLLs or dynamically loadable shared libraries such as Solaris `.so` libraries.

---

1. Set Up the Environment

Your DBA sets up the environment for calling external procedures by adding entries to the files `tnsname.ora` and `listener.ora` and by starting a Listener process exclusively for external procedures.

By default, the agent that handles external procedures is named `extproc` and runs on the same database server as your main application. In situations where reliability is critical, you might want to run external procedures on a separate database server, so that any crashes in the procedures do not bring everything down. You might also run external procedures on other machines to perform operations specific to those machines, such as invoking third-party software applications. When the agent process and the code for external procedures reside on a separate database server, you can specify the server using the name of a database link.

See Also: *Oracle9i Database Administrator's Guide*.

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 machine.

2. Identify the DLL

In this context, a DLL is any dynamically loadable operating-system file that stores external procedures.

For safety, 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'];
```

You must specify the full path to the DLL, because the linker cannot resolve references to 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';
```

When the external procedures are executed on a different server, the full path name might differ according to the operating system convention. To allow flexibility in moving the DLLs on this machine, 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 server, such as `/usr/bin/dll`.

```
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';
```

Notes:

- **a.** On a Windows system, you would specify the path using a drive letter and backslash characters (\) in the path.
- **b.** This technique is not applicable for VMS systems, where the `ENVS` section of listener.ora is not supported.

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 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 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, introduced for Java class methods, has now 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 | MAXLEN | CHARSETID | CHARSETFORM}
```

Note: Unlike Java, C doesn't 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.

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

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 Windows NT calling standard (C or Pascal) under which the external procedure was compiled. (Under the Pascal Calling Standard, arguments are reversed on the stack, and the called function must pop the stack.) 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 one external procedure crashes its agent process. 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 machines, the external procedure is invoked on the other machine. 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 machine 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.

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

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 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 *Invoker-rights* section of this manual for rules on their placement and defaults.

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 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;
```

Example: Running External Procedures on Different Machines

This example declares an external procedure that shuts down a machine cleanly. It calls this procedure on multiple other machines, then on the local machine. Because no AGENT clause is used in the CREATE LIBRARY statement, the default agent for this library is EXTPROC on the local machine. The calls to remote_shutdown supply the agent name in the SERVER parameter. The call to local_shutdown omits the agent name, so the default for the library is used.

```
CREATE DATABASE LINK dept_server_link USING 'dept_server_tns_alias';
CREATE DATABASE LINK sandbox_link USING 'sandbox_server_tns_alias';

CREATE LIBRARY shut_lib IS '/usr/bin/dll/cleanup.so';

CREATE OR REPLACE PROCEDURE local_shutdown(server VARCHAR2) AS LANGUAGE C
```

```
LIBRARY ’shut_lib’ NAME ’shutdown’;
CREATE OR REPLACE PROCEDURE remote_shutdown(server VARCHAR2) AS LANGUAGE C
LIBRARY shut_lib NAME shutdown AGENT IN(server);

BEGIN
-- This call is executed on the machine specified in the corresponding
-- database link.
  remote_shutdown(’dept_server_link’);
-- This call is also executed on another machine, specified by a different
-- database link.
  remote_shutdown(’sandbox_link’);
-- This call is executed using the agent specified in the library declaration,
-- in this case EXTPROC on the local machine.
  local_shutdown();
END;
/
```

Passing Parameters to Java Class Methods with Call Specifications

See Also: *Oracle9i Java Stored Procedures Developer’s Guide*

Passing Parameters to External C Procedures with Call Specifications

Call specifications allows a mapping between PL/SQL and C datatypes. Datatype mappings are shown below.

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 characterset 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. Each PL/SQL datatype maps to a default external datatype, as shown in Table 10–1.

Table 10–1 Parameter Datatype Mappings

PL/SQL Type	Supported External Types	Default External Type
BINARY_INTEGER BOOLEAN PLS_INTEGER	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	INT
NATURAL NATURALN POSITIVE POSITIVEN SIGNTYPE	[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

Table 10–1 Parameter Datatype Mappings (Cont.)

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 DECIMAL INT INTEGER NUMERIC SMALLINT	OCINUMBER	OCINUMBER
DATE	OCIDATE	OCIDATE
TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime	OCIDateTime
INTERVAL DAY TO SECOND INTERVAL YEAR TO MONTH	OCIInterval	OCIInterval
composite object types: ADTs	dvoid	dvoid
composite object types: collections (varrays, nested tables, index-by tables	OCICOLL	OCICOLL

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 10–2, 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 10–2 External Datatype Mappings

	Datatypes Used in C Prototype		
External Datatype	**IN, RETURN**	**IN by REFERENCE, RETURN by REFERENCE**	**IN OUT, OUT**
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 *
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 **

Table 10–2 (Cont.) External Datatype Mappings

OCINUMBER	OCINumber *	OCINumber *	OCINumber *
OCISTRING	OCIString *	OCIString *	OCIString *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *
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'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 can not 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 the above 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 and/or maximum length of a parameter
- `NULL`/`NOT NULL` indicators for parameters

- Characterset 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 must specify the parameter RETURN, and it must be in the last position.

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
MAXLEN
CHARSETID
CHARSETFORM
SELF
```

The following table 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 10–3 Property Datatype Mappings

Property	C Parameter		PL/SQL Parameter		
	Allowed External Types	Default External Type	Allowed Types	Allowed Modes	Default Passing Method
INDICATOR	SHORT INT LONG	SHORT	all scalars	IN IN OUT OUT RETURN	BY VALUE BY REFERENCE BY REFERENCE BY REFERENCE
LENGTH	[UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG	INT	CHAR LONG RAW RAW VARCHAR2	IN IN OUT OUT RETURN	BY VALUE BY REFERENCE BY REFERENCE BY REFERENCE
MAXLEN	[UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG	INT	CHAR LONG RAW RAW VARCHAR2	IN OUT OUT RETURN	BY REFERENCE BY REFERENCE BY REFERENCE
CHARSETID CHARSETFORM	[UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG	[UNSIGNED] INT	CHAR CLOB VARCHAR2	IN IN OUT OUT RETURN	BY VALUE BY REFERENCE BY REFERENCE 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 above.

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 above, 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 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 same host name and the exact same $ORACLE_HOME value, the agent uses the same national language 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), or an entirely different machine from the server, the agent uses the same national language 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: For more information about using national language data with the OCI, see *Oracle Call Interface Programmer's Guide* and the *Oracle9i Globalization and National Language Support Guide*.

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 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,
  PRAGMA RESTRICT_REFERENCES(calcAge_func, WNDS)
);
```

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 doesn’t 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.

Now 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
```

Sample C code, implementing the "external" member function, and the Object-Type-Translator (OTT)-generated struct definitions are included below.

```
#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.

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

Rules for PL/SQL and C Parameter Modes are listed above.

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, call the PL/SQL subprogram that published the external procedure. Such calls, which you code like a call to a regular PL/SQL procedure or function, can appear in the following:

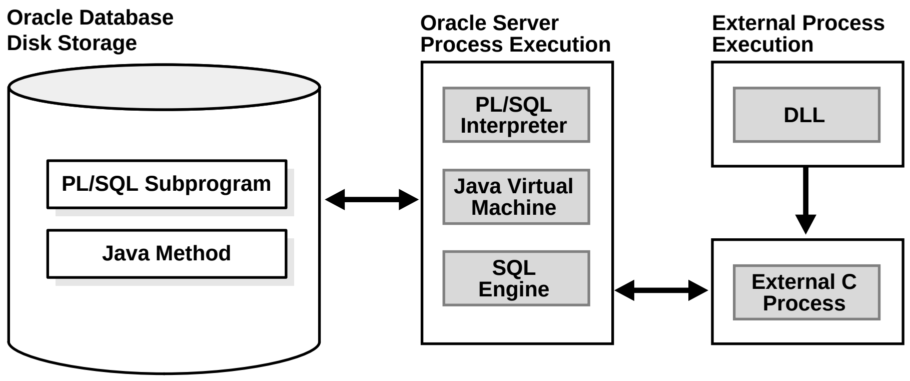
- Anonymous blocks
- Standalone and packaged subprograms
- Methods of an object type
- Database triggers
- SQL statements (calls to packaged functions only).

Although the `CALL` statement, described below, is confined to `SELECT`s, it can appear in either the `WHERE` clause or the `SELECT` list.

Note: To call a packaged function from SQL statements, you must use the pragma `RESTRICT_REFERENCES`, which asserts the purity level of the function (the extent to which the function is free of side effects). PL/SQL cannot check the purity level of the corresponding external procedure. Therefore, make sure that the procedure does not violate the pragma. Otherwise, you might get unexpected results.

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 10–1 shows how Oracle and external procedures interact.

Figure 10–1 Oracle and External Procedures



Preliminaries

Before you call your external procedure, you might want to make sure you understand the execution environment. Specifically, you might be interested in privileges, permissions, and synonyms.

Privileges

When external procedures are called through CALL specifications, they execute with *definer's privileges*, rather than with the privileges of their invoker.

An invoker's-privileges program 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 definer's privileges program 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:

- *Oracle9i SQL Reference*
- *Oracle9i Java Stored Procedures Developer's Guide*

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 example below, 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 which uses dynamic SQL to call `plsToC_demoExternal_proc`, which we published above. 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;
```

Now, call `J_calcFactorial`:

```
CALL J_calcFactorial(:x) INTO :y;
PRINT y
```

The result:

```
Y
------
   120
```

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

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 procedure declaration's `AS LANGUAGE` clause.

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 the Oracle server. 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 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.

You can run the agent on a separate machine from your database server. For details, see "Loading External C Procedures" on page 10-6.

Here, we call PL/SQL function `plsCallsCdivisor_func`, which we published above, 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.

Java Exception Handling

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

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 */
(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 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 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 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 example below, 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. It is only used for callbacks, and, furthermore, it is the only callback routine used. If you use the OCI handles obtained by this function for standard OCI calls, then the handles establish a new connection to the database and cannot be used for callbacks in the same transaction. In other words, during an external procedure call, you can use OCI handles for callbacks or a new connection but not for both.

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 10-50.

Java exceptions:

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

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 the Oracle server 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 `LOB`s. 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 now 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 objects' 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

See Also: *Oracle9i Java Stored Procedures Developer's Guide*

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 tables above.

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 Installation or User's Guide.)

To use the package, follow the instructions in `dbgextp.sql`. Your Oracle 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 current non-threaded configuration of the agent process, there is only one function active at a time. In the future, however, Oracle might thread the agent process, which would mean that multiple functions can be active at the same time. In that case, it is possible that two or more functions concurrently would try to access the global variable 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 for the above two reasons. 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 invocation of the same function. But, because of the DLL caching feature mentioned above, the DLL might be unloaded and reloaded between invocations, which means that the internal static variable would lose its value.

> **See Also:** For help in creating a dynamic link library, look in the RDBMS subdirectory /*public*, where a template *makefile* can be found.

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

Currently, 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

Application Security

This part contains the following chapters:

11

Database Security Overview for Application Developers

This chapter provides a basic understanding of application and database security policies. The following security policy topics are included in this chapter:

- Introduction to Database Security Policies
- Introduction to Application Security Policies

See Also: *Oracle Security Overview*

Introduction to Database Security Policies

This section briefly introduces security policies. It covers:

- Security Threats and Countermeasures
- What Information Security Policies Can Cover
- Features to Use in Establishing Security Policies

Security Threats and Countermeasures

Organizations should create a written security policy to enumerate the security threats they are trying to guard against, and the specific measures the organization must take. Security threats can be addressed with different types of measures:

- Procedural, such as requiring data center employees to display security badges
- Personnel, such as performing background checks or "vetting" key personnel
- Physical, such as securing computers in restricted-access facilities
- Technical, such as implementing strong authentication requirements for critical business systems

Consider whether the appropriate response to a threat is procedural, physical, technical, or personnel-related, or whether the threat can be met by a combination of the above measures.

For example, one possible security threat is disruption of critical business systems caused by a malicious person damaging a computer. A physical response to this threat is to secure key business computers in a locked facility. A procedural response is to create system backups at regular intervals. Personnel measures could include background checks on employees who access or manage key business systems.

Oracle9*i* offers many mechanisms which can implement the *technical* measures of a good security policy.

What Information Security Policies Can Cover

In addition to requirements unique to your environment, you should design and implement information security policies to address the following important issues:

- The level of security at the application level
- System and object privileges
- Database roles
- Enterprise roles
- How to grant and revoke privileges and roles
- How to create, alter, and drop roles
- How to control role use
- Level of granularity of access control
- User attributes which govern access to the database
- Whether to use encryption
- How to implement security in three-tier applications

Features to Use in Establishing Security Policies

The following elements of Oracle9*i* enable you to address security issues of a technical nature:

Application Security	Use this feature to attach privileges and roles to each application, while making sure that users do not misuse those roles and privileges when they are not using the application.
Fine-Grained Access Control	Use this feature to implement security policies at a high level of granularity; for example, to enforce row level security. Do this by creating security policy functions attached to the table or view on which you have based your application. Then, when a user enters a DML statement on that object, Oracle modifies that statement dynamically and transparently to the user.
Application Context	Use this feature to set up session-based attributes securely. For example, you can securely store such user attributes as a user name, employee number, and your position in the management hierarchy. You can retrieve that information later in the session and use it for fine-grained access control.
Secure Application Role	Use this feature to base use of roles on user-defined criteria. For example, you could allow use of a role by a user connecting only from a particular IP address, or accessing the database only through a particular middle tier.
Fine-Grained Auditing	Use this feature to monitor query access based on content. For example, you can monitor users accessing a specific row within a table. This feature can also be used to detect data misuse or serve as an intrusion detection system.

Oracle Label Security	Use this Oracle9*i* data server option to enforce fine-grained access control and label-based access control automatically. For example, you can label data Company Confidential or Partner Releaseable to automatically limit access to data based on the label of the data and the labels of data a user is permitted to access. By using Oracle Label Security, organizations can implement fine-grained and label-based access control quickly, in many cases without additional programming
Proxy Authentication	Use this feature to preserve user identity through a middle tier to the database, without the overhead of a separate database connection. It is able to proxy the user identity and credentials, such as a password or X.509 certificate, through the middle tier to the database. Also, on behalf of a user, it supports auditing of connections.
Data Encryption	Use this feature to encrypt information, as an extra measure of security.

See Also: *Oracle Security Overview*

Recommended Application Design Practices to Reduce Risk

To avoid potential problems, use the following recommended practices when implementing database roles. Each practice is explained in detail below.

- Tip 1: Enable and Disable Roles Promptly
- Tip 2: Encapsulate Privileges in Stored Procedures
- Tip 3: Use Role Passwords Unknown to the User
- Tip 4: Use Proxy Authentication and a Secure Application Role
- Tip 5: Use Secure Application Role to Verify IP Address
- Tip 6: Use Application Context and Fine-Grained Access Control

Tip 1: Enable and Disable Roles Promptly

Enable the proper role when the application starts, and disable it when the application terminates. To do this, you must take the following approach:

- Give each application distinct roles, where one role should contain *all* privileges necessary to use the application successfully. Depending on the situation, there might be several roles that contain various privileges, to provide tighter or less restrictive security while executing the application. Each database role should be protected by a password (or by operating system authentication) to prevent unauthorized use.

  Another role should contain only non-destructive privileges associated with the application (`SELECT` privileges for specific tables or views associated with the application). The read-only role allows the application user to generate custom reports using ad hoc tools, such as SQL*Plus. However, this role does not allow the application user to modify table data outside the application itself. A role designed for an ad hoc query tool may or may not be protected by a password (or by operating system authentication).

- At startup, each application should use the `SET ROLE` statement to enable one of the database roles associated with that application. If a password is used to authorize the role, then the password must be included in the `SET ROLE` statement within the application (encrypted by the application, if possible). If the role is authorized by the operating system, then the system administrator must have set up user accounts and applications so that application users get the appropriate operating system privileges when using the application.

- Upon termination, each application should disable the previously enabled database role.

- Application users should be granted database roles, as required.

Note: Database roles granted to users can nonetheless be enabled by users *outside the application*. Such use is not controlled by application-based security. Again, virtual private database is the best way to solve this problem. Also, in three-tier systems, it is possible to restrict the users from using the role outside of the application, by using a secure application role.

Additionally, you can:

- Specify the roles to enable when a user starts SQL*Plus, using the `PRODUCT_USER_PROFILE` table. This functionality is similar to that of a precompiler or Oracle Call Interface (OCI) application that issues a `SET ROLE` statement to enable specific roles upon application startup.
- Disable the use of the `SET ROLE` statement for SQL*Plus users with the `PRODUCT_USER_PROFILE` table. This allows a SQL*Plus user only the privileges associated with the roles enabled when the user started SQL*Plus.

  Other ad hoc query and reporting tools can also make use of the `PRODUCT_USER_PROFILE` table to restrict the roles and commands that each user can use while running that product.

  See Also: "Ways to Use Application Context with Fine-Grained Access Control" on page 12-8

  "Using Secure Application Role to Ensure Database Connection" on page 11-16

  The appropriate tool manual, such as the *SQL*Plus User's Guide and Reference*

Tip 2: Encapsulate Privileges in Stored Procedures

Another way to restrict users from exercising application privileges by way of ad hoc query tools is to encapsulate privileges into stored procedures. Grant users execute privileges on the procedures, rather than issuing them direct privilege grants. In this way, the logic goes with the privilege.

This allows users to exercise privileges only in the context of well-formed business applications. For example, consider authorizing users to update a table only by executing a stored procedure, rather than by updating the table directly. By doing this, you avoid the problem of the user having the SELECT privilege and using it outside the application.

> **See Also:** "Example 3: Event Triggers, Application Context, Fine-Grained Access Control, and Encapsulation of Privileges" on page 12-23

Tip 3: Use Role Passwords Unknown to the User

Grant privileges through roles that require a password unknown to the user.

If there are privileges which the user should use only within the application, you can enable the role by a password *known only by the creator of the role*. Use the application to issue a SET ROLE statement. Because the user does not have the password, you should either embed the password in the application or use a stored procedure to retrieve the role password from a database table. This measure discourages users from avoiding use of the application. However, while it does improve application security, it is not foolproof.

A user with access to application code could potentially find the password that is embedded in the application. This *security by obscurity* is not a good security practice. Embedding a password in the application protects against a user who wants to bypass the application (a lazy user). It does not protect against the user who deliberately wants to misuse privileges by accessing data and bypassing the application (a malicious user). Since client code can be decompiled and the embedded password recovered, you should only use the *embedded password* method to protect against the lazy users.

To use the stored procedure to retrieve the role password from a database table, a user would need EXECUTE permission, then execute the procedure, retrieve the password, and use the role outside of the application.

Tip 4: Use Proxy Authentication and a Secure Application Role

In three-tier systems, it is possible to enable a role only when the user accesses the database through a middle-tier application. This requires the use of proxy authentication and a secure application role. Proxy authentication distinguishes between a middle tier creating a session on behalf of a user and the user connecting directly. Both the proxy user (the middle tier) and the *real* user information are captured in the user session. A secure application role, which is implemented by a package, can do desired validation before allowing the user to assume the privileges in the role. When the application uses proxy authentication, the secure application role can validate that the user session was created by proxy, and that the user is connecting to the database through an application, and not directly.

Consider a situation in which you want to restrict use of an HR administration role to users accessing the database (by proxy) through the middle tier HRSERVER. You could create the following secure access role:

```
CREATE ROLE admin_role IDENTIFIED USING hr.admin;
```

Here, hr.admin is a package which performs desired validation. The package can determine if a user is connected by proxy using `SYS_CONTEXT (’userenv’, ’proxy_userid’)`, or `SYS_CONTEXT (userenv’, ’proxy_user’)`, or both return the ID and name of the proxy user (HRSERVER, in this case). If the user attempts to connect directly to the database, the hr.admin package will not allow the role to be set.

Tip 5: Use Secure Application Role to Verify IP Address

The secure application role can use additional information in the user session in order to restrict access. IP-address based security is not foolproof, since IP addresses can be spoofed. Therefore, you should never use IP address to make primary access control decisions, but you could use IP address to further restrict access, in addition to other controls. For example, you may want to ensure that a user session was created by proxy and that a middle tier user connecting from a particular IP address created the user session. Of course, the middle tier must authenticate itself to the database before creating a lightweight session, and the database ensures that the middle tier has privilege to create a session on behalf of the user. Your secure application role could verify the IP address of the incoming connection to ensure that the HRSERVER connection (or the lightweight user session) is coming from the appropriate IP address using `SYS_CONTEXT (userenv’,' 'ip_address')` before allowing `SET ROLE` to succeed. This provides an additional layer of security.

Tip 6: Use Application Context and Fine-Grained Access Control

In this scenario, you combine server-enforced fine-grained access control and, through application context, session-based attributes.

See Also: "Ways to Use Application Context with Fine-Grained Access Control" on page 12-8

Introduction to Application Security Policies

You should draft security policies for each database application. For example, each database application should have one or more database roles that provide different levels of security when executing the application. The database roles can be granted to user roles, or directly to specific usernames.

Applications that potentially allow unrestricted SQL statement execution (through tools such as SQL*Plus) also need security policies that prevent malicious access to confidential or important schema objects.

This section describes the following aspects of application security policies:

- Considerations for Using Application-Based Security
- Security-Related Tasks of Application Administrators
- Managing Application Privileges
- Creating Secure Application Roles
- Associating Privileges with the User’s Database Role
- Protecting Database Objects Through Use of Schemas
- Managing Object Privileges
- Enabling and Disabling Roles
- Granting and Revoking System Privileges and Roles
- Granting and Revoking Schema Object Privileges and Roles
- Granting to, and Revoking from, the User Group PUBLIC

Considerations for Using Application-Based Security

There are many issues to consider when you formulate and implement application security. Two of the main considerations are these:

- Are Application Users Also Database Users?
- Is Security Enforced in the Application or in the Database?

Are Application Users Also Database Users?

Oracle Corporation recommends that, where possible, you build applications in which application users are database users. In this way you can leverage the intrinsic security mechanisms of the database.

For many commercial packaged applications, application users are not database users. For these applications, multiple users authenticate themselves to the application, and the application then connects to the database as a single, highly-privileged user. We will call this the "One Big Application User" model.

Applications built in this fashion generally cannot use many of the intrinsic security features of the database, because the identity of the user is not known to the database.

For example, use of the following features is compromised by the One Big Application User model:

Auditing	A basic principle of security is accountability through auditing. However, if all actions in the database are performed by One Big Application User, then database auditing cannot hold individual users accountable for their actions. The application must implement its own auditing mechanisms to capture individual users' actions.
Oracle Advanced Security enhanced authentication	Strong forms of authentication supported by Oracle Advanced Security (such as, client authentication over SSL, tokens, and so on) cannot be used if the client authenticating to the database is the application, rather than an individual user.

Roles	Roles are assigned to database users. Enterprise roles are assigned to enterprise users who, though not created in the database, are known to the database. If application users are not database users, then the usefulness of roles is diminished. Applications must then craft their own mechanisms to distinguish between the privileges which various application users need to access data within the application.
Enterprise user management feature of Oracle Advanced Security	This feature enables users and their authorizations to be centrally managed in an LDAP-based directory such as Oracle Internet Directory. While enterprise users do not need to be created in the database, they do need to be known to the database. The One Big Application User model cannot take advantage of user and authorization management in LDAP.

Is Security Enforced in the Application or in the Database?

Applications whose users are also database users can either build security into the application, or rely upon intrinsic database security mechanisms such as granular privileges, virtual private database (fine-grained access control with application context), roles, stored procedures, and auditing (including fine-grained auditing). To the extent possible, Oracle recommends that applications utilize the security enforcement mechanisms of the database.

When security is enforced in the database itself, rather than in the application, it cannot be bypassed. The main shortcoming of application-based security is that security is bypassed if the user bypasses the application to access data. For example, a user who has SQL*Plus access to the database can execute queries without going through the Human Resources application. The user thus bypasses all of the security measures in the application.

Applications that use the One Big Application User model must build security enforcement into the application rather than using database security mechanisms. In this case, since it is the application—and not the database—which recognizes users, the application must enforce any per-user security measures itself.

This approach means that each and every application which accesses data must reimplement security. For example, if an organization implements a new report-writing tool, then it must also implement security to ensure that users do not get more data access through the report-writing tool than they would have in the application itself. Security becomes expensive because organizations must implement the same security policies in multiple applications. Each new application requires an expensive reimplementation.

See Also: "Use of Ad Hoc Tools a Potential Security Problem" on page 12-57

Security-Related Tasks of Application Administrators

In large database systems with many applications, you may decide to have application administrators. An application administrator is responsible for the following:

- Creating roles for the database application and managing the privileges of each database role
- Creating and managing the objects used by the application
- Maintaining and updating the application code, and Oracle procedures and packages, as necessary

Managing Application Privileges

Most database applications involve different privileges on different schema objects. Keeping track of which privileges are required for each application can be complex. In addition, authorizing users to run an application can involve many `GRANT` operations. This section provides some features to managing application privileges. Managing application privileges includes the following:

- Creating Roles to Simplify Application Privilege Management
- Advantages of Grouping Application Privileges in Roles

Creating Roles to Simplify Application Privilege Management

To simplify application privilege management, you can create a role for each application and grant that role all the privileges a user needs to run the application. In fact, an application might have a number of roles, each granted a specific subset of privileges that allow greater or lesser capabilities while running the application.

For example, suppose that every administrative assistant uses the Vacation application to record vacation taken by members of the department. To best manage this application, you should:

1. Create a `VACATION` role.
2. Grant all privileges required by the Vacation application to the `VACATION` role.

3. Grant the `VACATION` role to all administrative assistants or to a role named `ADMIN_ASSISTS` (if previously defined).

Advantages of Grouping Application Privileges in Roles

Grouping application privileges in a role aids privilege management. Consider the following administrative options:

- You can grant the role, rather than many individual privileges, to those users who run the application. Then, as employees change jobs, you need to grant or revoke only one role, rather than many privileges.
- You can change the privileges associated with an application by modifying only the privileges granted to the role, rather than the privileges held by all users of the application.
- You can determine which privileges are necessary to run a particular application by querying the `ROLE_TAB_PRIVS` and `ROLE_SYS_PRIVS` data dictionary views.
- You can determine which users have privileges on which applications by querying the `DBA_ROLE_PRIVS` data dictionary view.

Creating Secure Application Roles

Database access is based on privileges, which are often grouped into roles. Once grouped, the roles are granted to the application user. In previous releases, one would embed a password inside the application to ensure that users only enable the granted roles within the application. Roles secured by embedding passwords inside their applications are called 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. These application roles, those enabled by PL/SQL packages, are called secure application roles.

Within the package that implements the secure application role:

- The application must do the necessary validation. For example, the application must validate that the user is in a particular department, check that the user session was created by proxy, from a particular IP address, or that the user was authenticated using an X.509 certificate. To perform the validation, applications can use session information accessible through `SYS_CONTEXT` (`'userenv', <session_attribute>`). The accessible information indicates the way the user was authenticated, the IP address of the client, and whether the user was proxied.

- The application must issue a SET_ROLE command using dynamic SQL (DBMS_SESSION.SET ROLE).

Topics in this section include:

- Example of Creating a Secure Application Role
- Using Secure Application Role to Ensure Database Connection

---

Note: Because users can not change security domain inside Definer's Right procedures, secure application roles can only be enabled inside Invoker's Right procedures.

Example of Creating a Secure Application Role

To create a secure application role:

1. Create the roles as application roles and specify the authorized package that will enable the roles. In this example, hr.hr_admin is the specified authorized package.

```
CREATE ROLE admin_role IDENTIFIED USING hr.hr_admin;
CREATE ROLE staff_role IDENTIFIED USING hr.hr_admin;
```

2. Create an invoker’s right procedure.

```
CREATE OR REPLACE PACKAGE BODY hr_admin IS
PROCEDURE hr_app_report
AUTHID CURRENT_USER AS
BEGIN
    /* set application context in ’responsibility’ namespace */
    hr_logon.hr_set_responsibility;
    /* authentication check here */
    if (Hr.MySecurityCheck = TRUE)
    then
        /* check ’responsibility’ being set, then enable the roles without
        supplying the password */
        if (sys_context(’hr’,’role’) = ’admin’ )
    then
        dbms_session.set_role('admin_role');
    else
```

```
            dbms_session.set_role('staff_role');
            end if;
        end if;
    END;
    /* Create a dedicated authentication function for manageability so that
    changes in authentication policies would not affect the source code of the
    application - this design is up the application developers */
    /* the only policy in this function is that current user must have been
    authenticated using the proxy user 'SCOTT' */
    CREATE OR REPLACE FUNCTION hr.MySecurityCheck RETURN BOOLEAN
    AS
    BEGIN
        /* a simple check to see if current session is authenticated
        by the proxy user 'SCOTT' */
        if (sys_context('userenv','proxy_user') = 'SCOTT')
        then
            return TRUE;
        else
            return FALSE;
        end;
    END;
```

When enabling the secure application role, Oracle verifies that the authorized PL/SQL package is on the calling stack. This step verifies that the authorized PL/SQL package is issuing the command to enable the role. Also, when enabling the user's default roles, no checking will be performed for application roles.

Using Secure Application Role to Ensure Database Connection

Since a secure application role is a role implemented by a package, the package can do desired validation, such as ensuring that users can connect to the database through a middle tier or from a specific IP address. In this way, users are prevented from accessing data outside an application. They are forced to work within the framework of the application privileges that they have been granted.

Associating Privileges with the User's Database Role

A single user can use many applications and associated roles. However, you should ensure that the user has only the privileges associated with the running database role. Consider the following scenario:

- The `ORDER` role (for the Order application) contains the `UPDATE` privilege for the `INVENTORY` table

- The INVENTORY role (for the Inventory application) contains the SELECT privilege for the INVENTORY table
- Several order entry clerks have been granted both the ORDER and INVENTORY roles

In this scenario, an order entry clerk who has been granted both roles, can presumably use the privileges of the ORDER role when running the INVENTORY application to update the INVENTORY table. The problem is that updating the INVENTORY table is not an authorized action when using the INVENTORY application, but only when using the ORDER application.

To avoid such problems, consider using either the SET ROLE statement or the SET_ROLE procedure as explained below. You can also use the secure application role feature to allow roles to be set based on criteria you define.

Topics in this section include:

- Using the SET ROLE Statement
- Using the SET_ROLE Procedure
- Examples of Assigning Roles with Static and Dynamic SQL

Using the SET ROLE Statement

Use a SET ROLE statement at the beginning of each application to automatically enable its associated role and, consequently, disable all others. In this way, each application dynamically enables particular privileges for a user only when required.

The SET ROLE statement facilitates privilege management because, in addition to letting you control what information a user can access, it allows you to control when a user can access it. In addition, the SET ROLE statement keeps users operating in a well-defined privilege domain. If a user obtains privileges only from roles, then the user cannot combine these privileges to perform unauthorized operations.

See Also: "Enabling and Disabling Roles" on page 11-25

Using the SET_ROLE Procedure

The PL/SQL package DBMS_SESSION.SET_ROLE is functionally equivalent to the SET ROLE statement in SQL.

A limitation of roles is the inability to `SET ROLE` within a definer's rights procedure. The reason is that, for a definer's rights procedure, the database checks privileges at compilation time, not at execution time. That is, the database verifies that the owner of the procedure has necessary privileges—granted to him directly, not through a role—at the time the procedure is compiled. A `SET ROLE` statement does not work because the role is not enabled at compilation time, when the database checks privileges. At execution time, when the role is to be enabled, the database does not check the owner's privileges; instead, the database merely ensures that a user of the procedure has `EXECUTE` privilege on the procedure.

In cases where the database checks privileges at execution time rather than at compilation time, it is possible to issue a `SET ROLE`. Thus, the `DBMS_SESSION.SET_ROLE` command can be called from the following:

- Anonymous PL/SQL blocks
- Invoker’s rights stored procedures (except those invoked from within definer’s rights procedures)

In both the above cases, the database checks privileges at execution time, not at compilation time. Therefore, the database can validate that a user has the appropriate privileges (that is, that the user has been granted the role that is being set).

Note: If you use `DBMS_SESSION.SET_ROLE` within an invoker's rights procedure, the role remains in effect until you explicitly disable it. In keeping with the *least privilege* principle, (that users should have the fewest privileges they need to do their jobs), you should explicitly disable roles set within an invoker's rights procedure, at the end of the procedure.

Because PL/SQL performs the security check on SQL when an anonymous block is compiled, `SET_ROLE` will not affect the security role (in other words, it will not affect the roles enabled) for embedded SQL statements or procedure calls.

Examples of Assigning Roles with Static and Dynamic SQL

This section shows how static and dynamic SQL affect the assignment of roles.

Note: You may need to set up data structures for the following example, and certain others, to work. Set up the following:

```
CONNECT system/manager
DROP USER joe CASCADE;
CREATE USER joe IDENTIFIED BY joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO scott;
DROP ROLE acct;
CREATE ROLE acct;
GRANT acct TO scott;

CONNECT joe/joe;
CREATE TABLE finance (empno NUMBER);
GRANT SELECT ON finance TO acct;
CONNECT scott/tiger
```

Suppose you have a role named ACCT that has been granted privileges allowing you to select from table FINANCE in the JOE schema. In this case, the following block fails:

```
DECLARE
    n NUMBER;
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct');
    SELECT empno INTO n FROM JOE.FINANCE;
END;
```

The block fails because the security check which verifies that you have the SELECT privilege on table JOE.FINANCE occurs at compile time. At compile time, however, the ACCT role is not yet enabled. The role is not enabled until the block is executed.

The DBMS_SQL package, by contrast, is not subject to this restriction. When you use this package, the security checks are performed at runtime. Thus, a call to SET_ROLE would affect the SQL executed using calls to the DBMS_SQL package. The following block is, therefore, successful:

```
CREATE OR REPLACE PROCEDURE dynSQL_proc
AUTHID CURRENT_USER AS
   n NUMBER;
BEGIN
   SYS.DBMS_SESSION.SET_ROLE('acct');
   EXECUTE IMMEDIATE 'select empno from joe.finance' INTO n;
    --other calls to SYS.DBMS_SQL
```

```
END;
```

See Also: "Choosing Between Native Dynamic SQL and the DBMS_SQL Package" on page 8-11

Protecting Database Objects Through Use of Schemas

A *schema* is a security domain that can contain database objects. The privileges granted to each user or role control access to these database objects. This section covers:

- Unique Schemas
- Shared Schemas

Unique Schemas

Most schemas can be thought of as usernames: the accounts which enable users to connect to a database and access the database objects. However, *unique schemas* do not allow connections to the database, but are used to contain a related set of objects. Schemas of this sort are created as normal users, yet are not granted the `CREATE SESSION` system privilege (either explicitly or through a role). However, you must temporarily grant the `CREATE SESSION` and `RESOURCE` privilege to such schemas, if you want to use the `CREATE SCHEMA` statement to create multiple tables and views in a single transaction.

For example, the schema objects for a specific application might be owned by a given schema. Application users can connect to the database using typical database usernames and use the application and the corresponding objects, if they have the privileges to do so. However, no user can connect to the database using the schema set up for the application. This configuration prevents access to the associated objects through the schema, and provides another layer of protection for schema objects. In this case, the application could issue an `ALTER SESSION SET CURRENT_SCHEMA` statement to connect the user to the correct application schema.

Shared Schemas

For many applications, users do not need their own accounts—or their own schemas—in a database. These users merely need to access an application schema. For example, users John, Firuzeh and Jane are all users of the Payroll application, and they need access to the `Payroll` schema on the `Finance` database. None of them need to create their own objects in the database; in fact, they need only access

Payroll objects. To address this issue, Oracle Advanced Security provides enterprise users (schema-independent users).

Enterprise users, users managed in a directory service, can access a shared schema. They do not need to be created as database users; they are shared schema users of the database. Instead of creating a user account (that is, a user schema) in each database an enterprise user needs to access, as well as creating the user in the directory, an administrator can create an enterprise user once, in the directory, and point the user at a shared schema that many other enterprise users can also access.

In the previous example, if John, Firuzeh and Jane all access the `Sales` database as well as the `Finance` database, an administrator need only create a single schema in the `Sales` database, which all three users can access—instead of creating an account for each user on the `Sales` database. In this case, the DBA for the `Sales` database creates a shared schema called `sales_application`, as follows:

```
CREATE USER sales_application IDENTIFIED GLOBALLY AS ‘ ‘;
```

The mapping between enterprise users and a schema is done in the directory by means of one or more mapping objects. A mapping object maps the Distinguished Name (DN) of a user to a database schema that the user will access. This can be done in one of two ways:

- A full DN mapping maps the DN of a single directory user to a database schema, thus associating this user with a particular schema on a database.
- A partial DN mapping maps all users who share part of a DN to a database schema. A partial DN mapping is useful if multiple enterprise users that have something in common are already grouped under some common root in the directory tree. For example, all enterprise users in the directory subtree corresponding to the Engineering Division can be mapped to one shared schema on the bug database. Multiple enterprise users, who share part of their DN, can access the same shared schema.

When the database tries to determine the enterprise user’s schema in the directory (that is, the schema to which the database will connect the user), it searches for a full DN mapping. If it does not find a full DN mapping, then it searches for a partial DN. A full DN mapping thus takes precedence over a partial one.

For users authenticated by SSL to the database, or whose X.509 certificate or DN from a certificate is proxied to the database, the database uses the DN to search for the user in the directory. For password-authenticated enterprise users, the DN is obtained from the directory. That is, when a username is presented to the database for authentication (for example, `JANE`), the database searches internally to find if there is a local user Jane. If not, the database searches the directory for Jane, and retrieves an associated DN for Jane. Afterwards, the database refers to a mapping object as above to determine the correct shared schema to which Jane connects.

If a set of privileges should be granted to a group of users, this can be done by granting roles and privileges to a shared schema. Every user sharing the schema gets these local roles and local privileges in addition to the enterprise roles.

Each enterprise user can be mapped to a shared schema on each database that the user needs to access. These schema-independent users thus need not have a dedicated database schema on each database. Shared schemas therefore lowers the cost of managing users in an enterprise.

> **See Also:** "Switching to a Different Schema" on page 2-35
>
> *Oracle Advanced Security Administrator's Guide*

Managing Object Privileges

As part of designing your application, you need to determine the types of users who will be working with the application, and the level of access they need to accomplish their designated tasks. You must categorize these users into role groups, and then determine the privileges that must be granted to each role. This section covers:

- Object Privileges
- SQL Statements Permitted by Object Privileges

Object Privileges

End users are typically granted object privileges. An object privilege allows a user to perform a particular action on a specific table, view, sequence, procedure, function, or package. Table 11–1 summarizes the object privileges available for each type of object.

Table 11–1 Object Privileges

Object Privilege	Table	View	Sequence	Procedure (1)
ALTER	3		3	
DELETE	3	3		
EXECUTE				3
INDEX	3 (2)			
INSERT	3	3		
REFERENCES	3 (2)			
SELECT	3	3 (3)	3	
UPDATE	3	3		

Notes:

1 Stand-alone stored procedures, functions, and public package constructs

2 Privilege that cannot be granted to a role

3 Can also be granted for snapshots

SQL Statements Permitted by Object Privileges

As you implement and test your application, you should create each necessary role. Test the usage scenario for each role to be certain that the users of your application will have proper access to the database. After completing your tests, coordinate with the administrator of the application to ensure that each user is assigned the proper roles.

Table 11–2 lists the SQL statements permitted by the object privileges shown in Table 11–1.

Table 11–2 SQL Statements Permitted by Database Object Privileges

Object Privilege	SQL Statements Permitted
ALTER	ALTER object (table or sequence)
	CREATE TRIGGER ON object (tables only)
DELETE	DELETE FROM object (table or view)
EXECUTE	EXECUTE object (procedure or function)
	References to public package variables
INDEX	CREATE INDEX ON object (table or view)
INSERT	INSERT INTO object (table or view)
REFERENCES	CREATE or ALTER TABLE statement defining a FOREIGN KEY integrity constraint on object (tables only)
SELECT	SELECT...FROM object (table, view, or snapshot) SQL statements using a sequence

Creating a Role and Protecting Its Use

This section explains how to create a new role and protect its use. This section covers:

- Creating and Implementing a New Role
- Managing Roles
- Protecting Role Use

Creating and Implementing a New Role

To create a role, you must have the CREATE ROLE system privilege.

The name of a new role must be unique among existing usernames and role names of the database. Roles are not contained in the schema of any user.

Immediately after creation, a role has no privileges associated with it. To associate privileges with a new role, you must grant it privileges or other roles.

Managing Roles

You can create roles such that their use is authorized using information from the operating system, from a network authentication service, or from an LDAP-based directory. This enables role management to be centralized.

Central management of roles provides many benefits. If an employee leaves, for example, all of her roles and permissions can be changed in a single place.

Protecting Role Use

The use of a role can be protected by an associated password. For example:

```
CREATE ROLE Clerk IDENTIFIED BY Bicentennial;
```

A user who is granted a role protected by a password can enable or disable the role only by supplying the proper password for the role using a SET ROLE statement. If a role is created without any protection, then any grantee can enable or disable it.

Separate SET ROLE statements can be used to enable one database role, and disable all other roles of a user. This way, the user cannot use privileges (from a role) which were intended for use with another application. With ad hoc query tools such as SQL*Plus or Enterprise Manager, users can explicitly enable only the roles for which they are authorized.

A secure application role can incorporate additional logic to determine under what conditions the role is enabled. The conditions can reference any information available in the user session. This means information accessible through the USERENV application context namespace, such as the IP address from which the session connected, the method of authentication, and whether the user was proxied or not (that is, connected through a middle tier).

.

See Also:

- "Explicitly Enabling Roles" on page 11-27
- *Oracle9i Database Administrator's Guide*
- For information about network authentication services, see *Oracle Advanced Security Administrator's Guide*

Enabling and Disabling Roles

When a user has been granted a role, the role must be enabled before the privileges associated with it become available in the user's current session. Some, all, or none of the user's roles can be enabled or disabled. The following sections discuss when roles should be enabled and disabled, and the different ways in which a user can have roles enabled or disabled. Topics in this section include:

- When to Enable Roles

- Default Roles
- Explicitly Enabling Roles
- Enabling and Disabling Roles When OS_ROLES=TRUE
- Dropping Roles

When to Enable Roles

In general, a user's security domain should permit him to perform the task at hand, yet limit him from having privileges which are not necessary for the current job. For example, a user should have all the privileges to work with the database application currently in use, but not have any privileges required for any other database applications. Having too many privileges might allow users to access information through unintended methods.

Privileges granted directly to a user are always available to him; therefore, directly granted privileges cannot be selectively enabled and disabled, depending on his current task. By contrast, privileges granted to a role can be selectively made available to any user granted the role. The enabling of roles *never* affects privileges explicitly granted to the user. The following sections explain how a user's roles can be selectively enabled (and disabled).

Default Roles

A default role is automatically enabled for a user when the user creates a session. A user's list of default roles should include those which correspond to his or her typical job function.

Each user has a list of zero, one, or more default roles. Any role directly granted to a user can potentially be a default role. An indirectly granted role (a role that is granted to a role) cannot be a default role.

The number of default roles for a user should not exceed the maximum number of enabled roles that are allowed per user (as specified by the initialization parameter `MAX_ENABLED_ROLES`). If the number of default roles for a particular user exceeds this maximum, then errors are returned when the user attempts a connection, and the connection is not allowed.

Note: A default role is automatically enabled for a user when the user creates a session. Placing a role in a user's list of default roles bypasses authentication for the role, whether it is authorized using a password or through the operating system.

A user's list of default roles can be set and altered using the SQL statement ALTER USER. If the user's list of default roles is specified as ALL, then every role granted to her is automatically added to her list of default roles. Only subsequent modification of the user's default role list can remove newly granted roles from her list of default roles.

Modifications to a user's default role list only apply to sessions created after the alteration or role grant; neither method applies to a session in progress at the time of the user alteration or role grant.

Explicitly Enabling Roles

Any user (or application) can use the SET ROLE statement to enable any granted roles, provided the grantee supplies role passwords, when necessary.

A SET ROLE statement enables all specified roles, provided that they have been granted to the user. All roles granted to the user that are not explicitly specified in a SET ROLE statement are disabled, including any roles previously enabled.

When you enable a role that contains other roles, all the indirectly granted roles are specifically enabled. Each indirectly granted role can be explicitly enabled or disabled for a user.

If a role is protected by a password, then the role can only be enabled by indicating its password in the SET ROLE statement. If the role is not protected by a password, then it can be enabled with a simple SET ROLE statement.

The following examples illustrate how roles can be enabled and disabled.

Assume that user Morris' security domain is as follows:

- He is granted three roles:

  PAYROLL_CLERK (password BICENTENNIAL)

  ACCTS_PAY (password GARFIELD)

  ACCTS_REC (identified externally)

- The PAYROLL_CLERK role includes the indirectly granted role PAYROLL_REPORT (identified externally)
- Morris' only default role is PAYROLL_CLERK

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

```
CREATE ROLE Payroll_clerk;
CREATE ROLE Payroll_report;

GRANT PAYROLL_CLERK TO hr;
GRANT ACCTS_PAY TO hr;
GRANT ACCTS_REC TO hr;
```

Morris' currently enabled roles can be changed from his default role, PAYROLL_CLERK, to ACCTS_PAY and ACCTS_REC, by the following statements:

```
SET ROLE accts_pay IDENTIFIED BY garfield;
SET ROLE accts_pay IDENTIFIED BY accts_rec;
```

Notice that in the first statement, multiple roles can be enabled in a single SET ROLE statement. The ALL and ALL EXCEPT options of the SET ROLE statement also allow several roles granted directly to the user to be enabled in one statement:

```
SET ROLE ALL EXCEPT Payroll_clerk;
```

This statement shows the use of the ALL EXCEPT option of the SET ROLE statement. Use this option when you want to enable most of a user's roles and only disable one or more. Similarly, all of Morris' roles can be enabled by the following statement:

```
SET ROLE ALL;
```

When using the ALL or ALL EXCEPT options of the SET ROLE statement, all roles to be enabled either must not require a password, or must be authenticated using the operating system. If a role requires a password, then the SET ROLE ALL or ALL EXCEPT statement is rolled back and an error is returned. A user can also explicitly enable any indirectly granted roles granted to him or her through an explicit grant of another role. Morris can thus issue the following statement:

```
SET ROLE Payroll_report;
```

Enabling and Disabling Roles When OS_ROLES=TRUE

If OS_ROLES is set to TRUE, then any role granted by the operating system can be dynamically enabled using the SET ROLE statement. However, any role not identified in a user's operating system account cannot be specified in a SET ROLE statement. Such a role is ignored, even if a it has been granted using a GRANT statement.

When OS_ROLES is set to TRUE, a user can enable as many roles as are specified by the initialization parameter MAX_ENABLED_ROLES.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about use of the operating system for role authorization

Dropping Roles

When you drop a role, the security domains of all users and roles granted that role are immediately changed to reflect the absence of the dropped role's privileges. All indirectly granted roles of the dropped role are also removed from affected security domains. Dropping a role automatically removes the role from all users' default role lists.

Because the creation of objects is not dependent upon the privileges received through a role, no cascading effects regarding objects need to be considered when dropping a role. For example, tables or other objects are not dropped when a role is dropped.

You can drop a role using the SQL statement DROP ROLE. For example:

```
DROP ROLE clerk;
```

To drop a role, you must have the DROP ANY ROLE system privilege or have been granted the role with the ADMIN OPTION.

Granting and Revoking System Privileges and Roles

The following sections explain how to grant and revoke system privileges and roles.

- Granting System Privileges and Roles
- Granting System Privileges and Roles with the ADMIN OPTION
- Revoking System Privileges and Roles

Granting System Privileges and Roles

System privileges and roles can be granted to other roles or users using the SQL command GRANT, as shown in the following example:

Note: 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;
CREATE USER tsmith IDENTIFIED BY tsmith;
CREATE USER finance IDENTIFIED BY finance;
CREATE USER michael IDENTIFIED BY michael;
CREATE ROLE Payroll_report;
GRANT CREATE TABLE, Accts_rec TO finance IDENTIFIED BY finance;
GRANT CREATE TABLE, Accts_rec TO tsmith IDENTIFIED BY tsmith;
GRANT REFERENCES ON Dept_tab TO jward;
CONNECT scott/tiger
CREATE VIEW Salary AS SELECT Empno,Sal from Emp_tab;
```

```
GRANT CREATE SESSION, Accts_pay TO jward, finance;
```

Schema object privileges cannot be granted along with system privileges and roles in the same GRANT statement.

Granting System Privileges and Roles with the ADMIN OPTION

A system privilege or role can be granted with the ADMIN OPTION. A grantee with this option has several expanded capabilities:

- The grantee can grant or revoke the system privilege or role to or from *any* user or other role in the database. However, a user cannot revoke a role from himself.
- The grantee can further grant the system privilege or role with the ADMIN OPTION.

- The grantee of a role can alter or drop the role.

A grantee without the ADMIN OPTION cannot perform the above operations. Note also that this option is not valid when granting a role to another role.

When a user creates a role, the role is automatically granted to the creator with the ADMIN OPTION.

Assume that you grant the NEW_DBA role to MICHAEL with the following statement:

```
GRANT new_dba TO michael WITH ADMIN OPTION;
```

Not only can the user MICHAEL use all of the privileges implicit in the NEW_DBA role, but he can grant, revoke, or drop the NEW_DBA role, as necessary.

Privileges Required to Grant System Privileges or Roles To grant a system privilege or role, the grantor requires the ADMIN OPTION for all system privileges and roles being granted. Additionally, any user with the GRANT ANY ROLE system privilege can grant any role in a database.

Revoking System Privileges and Roles

System privileges and roles can be revoked using the SQL command REVOKE. For example:

```
REVOKE CREATE TABLE, Accts_rec FROM tsmith, finance;
```

The ADMIN OPTION for a system privilege or role cannot be selectively revoked. To do so, you must first revoke the privilege or role, and then grant it without the ADMIN OPTION.

Privileges Required to Revoke System Privileges and Roles Any user with the ADMIN OPTION for a system privilege or role can revoke the privilege or role from any other database user or role. The user who revokes a privilege or role need not be the user who originally granted it. Additionally, any user with the GRANT ANY ROLE privilege can revoke *any* role.

Cascading Effects of Revoking System Privileges There are no cascading effects when revoking a system privilege related to DDL operations, regardless of whether the privilege was granted with or without the ADMIN OPTION. For example, assume the following:

1. You grant the CREATE TABLE system privilege to JWARD with the WITH ADMIN OPTION.

2. JWARD creates a table.
3. JWARD grants the CREATE TABLE system privilege to TSMITH.
4. TSMITH creates a table.
5. You revoke the CREATE TABLE privilege from JWARD.
6. JWARD's table continues to exist. TSMITH continues to have the CREATE TABLE system privilege, and his table still exists.

Cascading effects can be observed when revoking a system privilege related to a DML operation. For example, if SELECT ANY TABLE is granted to a user, and if that user has created any procedures, then all procedures contained in the user's schema must be reauthorized before they can be used again (after the revoke).

Granting and Revoking Schema Object Privileges and Roles

You can grant schema object privileges to roles or users using the SQL command GRANT. The following statement grants the SELECT, INSERT, and DELETE object privileges for all columns of the EMP_TAB table to the users JWARD and TSMITH:

```
GRANT SELECT, INSERT, DELETE ON Emp_tab TO jward, tsmith;
```

To grant the INSERT object privilege for only the ENAME and JOB columns of the EMP_TAB table to the users JWARD and TSMITH, enter the following statement:

```
GRANT INSERT(Ename, Job) ON Emp_tab TO jward, tsmith;
```

To grant all schema object privileges on the SALARY view to the user WALLEN, use the ALL shortcut. For example:

```
GRANT ALL ON Salary TO wallen;
```

System privileges and roles cannot be granted along with schema object privileges in the same GRANT statement.

The following section explains granting and revoking schema object privileges. It includes:

- Granting and Revoking Schema Object Privileges with the GRANT OPTION
- Revoking Schema Object Privileges

Granting and Revoking Schema Object Privileges with the GRANT OPTION

A schema object privilege can be granted to a user with the GRANT OPTION. This special privilege allows the grantee several expanded privileges:

- The grantee can grant the schema object privilege to any user or any role in the database.
- The grantee can also grant the schema object privilege to other users, with or without the GRANT OPTION.
- If the grantee receives schema object privileges for a table with the GRANT OPTION, and the grantee has the CREATE VIEW or the CREATE ANY VIEW system privilege, then the grantee can create views on the table and grant the corresponding privileges on the view to any user or role in the database.

The user whose schema contains an object is automatically granted all associated schema object privileges with the GRANT OPTION.

> **Note:** The GRANT OPTION is not valid when granting a schema object privilege to a role. Oracle prevents the propagation of schema object privileges through roles, so that grantees of a role cannot propagate object privileges received through roles.

Privileges Required to Grant Schema Object Privileges To grant a schema object privilege, the grantor must either

- Be the owner of the schema object being specified, or
- Have received, with the GRANT OPTION, the schema object privileges in question

Revoking Schema Object Privileges

Schema object privileges can be revoked using the SQL command REVOKE. For example, the original grantor can revoke the SELECT and INSERT privileges on the EMP_TAB table from the users JWARD and TSMITH by entering:

```
REVOKE SELECT, INSERT ON Emp_tab FROM jward, tsmith;
```

For the table DEPT_TAB, a grantor could also revoke all privileges that he or she granted to the role HUMAN_RESOURCES by entering the following statement:

```
REVOKE ALL ON Dept_tab FROM human_resources;
```

The statement is valid even if only one privilege was granted. Note that this statement would only revoke the privileges that the grantor authorized, not the grants made by other users. The GRANT OPTION for a schema object privilege cannot be selectively revoked; the schema object privilege must be revoked and then regranted without the GRANT OPTION. A user cannot revoke schema object privileges from himself.

Revoking Column-Selective Schema Object Privileges Recall that column-specific INSERT, UPDATE, and REFERENCES privileges can be granted for tables or views. However, it is not possible to revoke column-specific privileges selectively with a similar REVOKE statement. Instead, the grantor must first revoke the schema object privilege for all columns of a table or view, and then selectively grant the new column-specific privileges again.

For example, assume the role HUMAN_RESOURCES has been granted the UPDATE privilege on the DEPTNO and DNAME columns of the table DEPT_TAB. To revoke the UPDATE privilege on just the DEPTNO column, enter the following two statements:

```
REVOKE UPDATE ON Dept_tab FROM human_resources;
GRANT UPDATE (Dname) ON Dept_tab TO human_resources;
```

The REVOKE statement revokes the UPDATE privilege on all columns of the DEPT_TAB table from the role HUMAN_RESOURCES. The GRANT statement regrants the UPDATE privilege on the DNAME column to the role HUMAN_RESOURCES.

Revoking the REFERENCES Schema Object Privilege If the grantee of the REFERENCES object privilege has used the privilege to create a foreign key constraint (that currently exists), then the grantor can only revoke the privilege by specifying the CASCADE CONSTRAINTS option in the REVOKE statement:

```
REVOKE REFERENCES ON Dept_tab FROM jward CASCADE CONSTRAINTS;
```

When the CASCADE CONSTRAINTS option is specified, any foreign key constraints currently defined that use the revoked REFERENCES privilege are dropped.

Privileges Required to Revoke Schema Object Privileges To revoke a schema object privilege, the revoker must be the original grantor of the object privilege being revoked.

Cascading Effects of Revoking Schema Object Privileges Revoking a schema object privilege can have several cascading effects that should be investigated before a REVOKE statement is issued:

- Schema object definitions that depend on a DML object privilege can be affected if the DML object privilege is revoked. For example, assume the procedure body of the TEST procedure includes a SQL statement that queries data from the EMP_TAB table. If the SELECT privilege on the EMP_TAB table is revoked from the owner of the TEST procedure, then the procedure can no longer be executed successfully.
- Schema object definitions that require the ALTER and INDEX DDL object privileges are not affected, if the ALTER or INDEX object privilege is revoked. For example, if the INDEX privilege is revoked from a user that created an index on someone else's table, then the index continues to exist after the privilege is revoked.
- When a REFERENCES privilege for a table is revoked from a user, any foreign key integrity constraints defined by the user that require the dropped REFERENCES privilege are automatically dropped. For example, assume that the user JWARD is granted the REFERENCES privilege for the DEPTNO column of the DEPT_TAB table and that she creates a foreign key on the DEPTNO column in the EMP_TAB table that references the DEPTNO column. If the REFERENCES privilege on the DEPTNO column of the DEPT_TAB table is revoked, then the foreign key constraint on the DEPTNO column of the EMP_TAB table is dropped in the same operation.
- If a grantor's object privilege is revoked, then the schema object privilege grants propagated using the GRANT OPTION are revoked. For example, assume that USER1 is granted the SELECT object privilege with the GRANT OPTION, and grants the SELECT privilege on EMP_TAB to USER2. Subsequently, the SELECT privilege is revoked from USER1. This revoke is cascaded to USER2 as well. Any schema objects that depended on USER1's and USER2's revoked SELECT privilege can also be affected.

How Grants Affect Dependent Objects Issuing a GRANT statement against a schema object causes the "last DDL time" attribute of the object to change. This can invalidate any dependent schema objects, in particular PL/SQL package bodies that refer to the schema object. These then must be recompiled.

Granting to, and Revoking from, the User Group PUBLIC

Privileges and roles can be granted to and revoked from the user group PUBLIC. Because PUBLIC is accessible to every database user, all privileges and roles granted to PUBLIC are accessible to every database user.

You should only grant a privilege or role to PUBLIC if every database user requires the privilege or role. This recommendation restates the general rule that at any given time, each database user should only have the privileges required to successfully accomplish the current task.

Note: Certain privileges and roles granted to PUBLIC by default may not be needed in your situation. Oracle Corporation highly recommends that system administrators review grants to PUBLIC and revoke privileges that are not absolutely necessary.

This section explains granting and revoking from the user group PUBLIC. It includes:

- Revoking Security-Vulnerable Packages from PUBLIC
- Cascading Effects of Revokes from PUBLIC
- When Grants and Revokes Take Effect

Revoking Security-Vulnerable Packages from PUBLIC

All unnecessary privileges, grants, and roles should be revoked from PUBLIC. Any database user can exercise privileges that are granted to PUBLIC. Such privileges include EXECUTE on various PL/SQL packages. These packages may allow minimally privileged users to access and execute packages that they may not have permit to access directly. The more potentially security vulnerable packages include:

UTL_SMTP	Allows arbitrary users to send and receive e-mail messages. Granting this package to PUBLIC may permit unauthorized exchange of e-mail messages.
UTL_TCP	Allows the database server to establish outgoing network connections any receiving network service. Thus, the database server and any waiting network service exchange data.
UTL_HTTP	Allows a database to request and retrieve data through HTTP. Granting this package to PUBLIC may permit data to be sent through HTML forms to a malicious web site.

`UTL_FILE`	Allows text level access to any file on the host operating system, if configure improperly. Even when properly configured, this package does not distinguish between its calling applications may write arbitrary data into the same location that is written by another application.
`DBMS_RANDOM`	Allows one to encrypt stored data. Most users should not have the privilege to encrypt data. Encrypted data may be non-recoverable if the keys are not securely generated, stored, and managed. Oracle recommends that `EXECUTE` on this package be granted to a role, and the role granted to those users who need to encrypt data.

These packages are useful to the applications that need them and warrant proper configuration and usage, but they may not be suitable or required for other applications. If necessary, revoke the package from `PUBLIC` and database users.

Cascading Effects of Revokes from PUBLIC

Revokes from `PUBLIC` can cause significant cascading effects, depending on the privilege that is revoked. If any privilege related to a DML operation is revoked from `PUBLIC` (for example, `SELECT ANY TABLE`, `UPDATE ON EMP_TAB`), then all procedures in the database (including functions and packages) must be reauthorized before they can be used again. Therefore, use caution when granting DML-related privileges to `PUBLIC`.

When Grants and Revokes Take Effect

Depending upon what is granted or revoked, a grant or revoke takes effect at different times:

- All grants/revokes of privileges (system and schema object) to users, roles, or `PUBLIC` are immediately observed.
- All grants/revokes of roles to users, other roles, or `PUBLIC` are observed only when a current user session issues a `SET ROLE` statement to re-enable the role after the grant/revoke, or when a new user session is created after the grant/revoke.

See Also: "Listing Privilege and Role Information" in the *Oracle9i Database Administrator's Guide*

12

Implementing Application Security Policies

This chapter explains how to implement application security policies. Topics in this chapter include:

Introduction to Application Context

Application context allows you to write applications which draw upon certain aspects of a user's session information. It provides a way to define, set, and access attributes that an application can use to enforce access control—specifically, fine-grained access control.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, customers would be limited to access their own orders (`ORDER_NUMBER`) and customer number (`CUSTOMER_NUMBER`). These can be used as security attributes.

Consider a user running a Human Resource application. Part of the application's initialization process is to determine the kind of responsibility that the user can assume, based on the user's identity. This responsibility ID becomes part of the Human Resource application context; it will affect what data the user can access throughout the session.

This section explains the use of application context. It includes:

- Features of Application Context
- Ways to Use Application Context with Fine-Grained Access Control
- User Models and Virtual Private Database
- Creating a Virtual Private Database Policy with Oracle Policy Manager
- How to Use Application Context
- Examples: Application Context Within a Fine-Grained Access Control Function
- Automatic Reparse
- Introduction to Application Context Accessed Globally
- Initializing Application Context Externally
- Initializing Application Context Globally

Features of Application Context

Application context provides important security features:

- Specifying Attributes for Each Application
- Providing Security Validation
- Providing Access to Predefined Attributes Through the USERENV Namespace

- Externalized Application Contexts

Specifying Attributes for Each Application

Each application can have its own context with its own attributes. Suppose, for example, you have three applications: General Ledger, Order Entry, and Human Resources. You can specify different attributes for each application. Thus,

- For the General Ledger application context, you can specify the attributes `SET_OF_BOOKS` and `TITLE`.
- For the Order Entry application context, you can specify the attribute `CUSTOMER_NUMBER`.
- For the Human Resources application context, you can specify the attributes `ORGANIZATION_ID`, `POSITION`, and `COUNTRY`.

In each case, you can adapt the application context to your precise security needs.

Providing Security Validation

Suppose you have a General Ledger application, which has access control based upon the set of books being used. If a user accessing this application changes the set of books he is working on from *01* to *02*, the application context can ensure that:

- Set *02* is a valid set of books.
- The user has privileges to access set of books *02*.

The validation function can check application metadata tables to make this determination and ensure that the attributes in combination are in line with the overall security policy. To prevent users from changing a context attribute without the above security validation, Oracle ensures that an attribute can be changed only by the designated package which implements the context.

Providing Access to Predefined Attributes Through the USERENV Namespace

Oracle9*i* provides a built-in application context namespace, USERENV, which provides access to predefined attributes. These attributes are session primitives—information which the database captures regarding a user's session. For example, the IP address from which a user connected, the username, and a proxy username (in cases where a user connection is proxied through a middle tier), are all available as predefined attributes through the USERENV application context.

Predefined attributes can be very useful for access control. For example, if you are using a three-tier application which creates lightweight user sessions through OCI or thick JDBC, you can access the PROXY_USER attribute in the USERENV application context to determine whether the user's session was created by a middle tier application. Your policy function could allow a user to access data only for connections where the user is proxied. If the user is not proxied (that is, when the user connects directly to the database), the user would not be able to access any data.

While you can use the PROXY_USER attribute within VPD to ensure that users only access data through a particular middle-tier application, a different approach would be to develop a secure application role. Rather than have each policy ensure that the user accesses the database by being proxied through HRAPPSERVER, you could have the secure application role enforce this.

Although predefined attributes can be accessed through the USERENV application context, they cannot not be changed. They are listed in Table 12–1.

Use the following syntax to return information about the current session.

```
SYS_CONTEXT('userenv', 'attribute')
```

Note: The USERENV application context namespace is intended to replace the USERENV function provided in earlier database releases.

See Also: SYS_CONTEXT in the *Oracle9i SQL Reference* for complete details about the USERENV namespace and its predefined attributes

Table 12–1 Key to Predefined Attributes in USERENV Namespace

Predefined Attribute	Meaning
TERMINAL	Returns the operating system identifier for the client of the current session. "Virtual" in TCP/IP
LANGUAGE	Returns the language and territory currently used by the session, along with the database character set in the form: *language_territory.characterset*
LANG	Returns abbreviation for the language name
SESSIONID	Returns auditing session identifier
INSTANCE	Returns instance identification number of the current instance
ENTRYID	Returns available auditing entry identifier
ISDBA	Returns TRUE if you currently have the DBA role enabled and FALSE if you do not.
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package
NLS_TERRITORY	Returns the territory of the current session
NLS_CURRENCY	Returns the currency symbol of the current session
NLS_CALENDAR	Returns NLS calendar used for dates in the current session
NLS_DATE_FORMAT	Returns the current date format of the current session
NLS_DATE_LANGUAGE	Returns language used to express dates in the current session
NLS_SORT	Indicates whether the sort base is binary or linguistic
CURRENT_USER	Returns name of user under whose privilege the current session runs. Can be different from SESSION_USER from within a stored procedure (such as an invoker-rights procedure).
CURRENT_USERID	Returns the user ID of the user under whose privilege the current session runs. Can can be different from SESSION_USERID from within a stored procedure (such as an invoker-rights procedure).
SESSION_USER	Returns the database user name by which the current user is authenticated
SESSION_USERID	Returns the identifier of the database user name by which the current user is authenticated

Table 12–1 Key to Predefined Attributes in USERENV Namespace (Cont.)

Predefined Attribute	Meaning
CURRENT_SCHEMA	Returns the name of the default schema being used in the current session. This can be changed with an ALTER SESSION SET SCHEMA statement.
CURRENT_SCHEMAID	Returns the identifier of the default schema being used in the current session. This can be changed with an ALTER SESSION SET SCHEMAID statement.
PROXY_USER	Returns the name of the database user (typically middle tier) who opened the current session on behalf of SESSION_USER
PROXY_USERID	Returns identifier of the database user (typically middle tier) who opened the current session on behalf of SESSION_USER
DB_DOMAIN	Returns the domain of the database as specified in the DB_DOMAIN initialization parameter
DB_NAME	Returns the name of the database as specified in the DB_NAME initialization parameter
HOST	Returns the name of the host machine on which the database is running
OS_USER	Returns the operating system username of the client process that initiated the database session
EXTERNAL_NAME	Returns the external name of the database user
IP_ADDRESS	Returns the IP address (when available) of the machine from which the client is connected
NETWORK_PROTOCOL	Returns the protocol named in the connect string (PROTOCOL=*protocol*)
BG_JOB_ID	Returns the background job ID
FG_JOB_ID	Returns the foreground job ID
AUTHENTICATION_TYPE	Shows how the user was authenticated (DATABASE, OS, NETWORK, PROXY)
AUTHENTICATION_DATA	Returns the data being used to authenticate the login user. If the user has been authenticated through SSL, or if the user's certificate was proxied to the database, this includes the user's X.509 certificate
CURRENT_SQL	SQL text of the query that triggers fine-grained audit event handler. Only valid inside the event handler
CLIENT_IDENTIFIER	User-defined client identifier for the session

Table 12–1 Key to Predefined Attributes in USERENV Namespace (Cont.)

Predefined Attribute	Meaning
`GLOBAL_CONTEXT_MEMORY`	Amount of shared memory used by global application context, in bytes

Externalized Application Contexts

Many applications store attributes used for fine-grained access control within a database metadata table that they use for access control. For example, an `EMPLOYEES` table could include cost center, title, signing authority, and other information useful for fine-grained access control. However, many organizations centralize user information and user management in an LDAP-based directory such as Oracle Internet Directory. These organizations also wish to centralize the information about users that is used for access control. Application context attributes can be stored in Oracle Internet Directory and assigned to one or more enterprise users. They can be retrieved automatically upon login for an enterprise user, and used to initialize an application context.

Note: Enterprise User Management is a feature of Oracle Advanced Security.

See Also: "Initializing Application Context Globally" on page 12-35

Oracle Advanced Security Administrator's Guide

Ways to Use Application Context with Fine-Grained Access Control

To simplify the implementation of a security policy, you have the option of using application context within a fine-grained access control function.

Note: Using application context with fine-grained access control is called virtual private database (VPD).

Application context can be used in the following ways with fine-grained access control:

- Using Application Context as a Secure Data Cache
- Using Application Context to Return a Specific Predicate (Security Policy)
- Using Application Context to Provide Attributes Similar to Bind Variables in a Predicate

Using Application Context as a Secure Data Cache

Accessing an application context inside your fine-grained access control policy function is like writing down an often-used phone number and posting it next to your phone, where you can find it easily—rather than looking it up every time you need it.

For example, suppose you base access to the `ORDERS_TAB` table upon customer number. Rather than querying the customer number for a logged-in user each time you need it, you could store the number in the application context. In this way, the customer number is available when you need it.

Application context is especially helpful if your security policy is based upon multiple security attributes. For example, a policy function which bases a predicate on four attributes (such as employee number, cost center, position, spending limit) would have to execute multiple subqueries to retrieve this information. If all of this data is already available through application context, then performance will be much faster.

Using Application Context to Return a Specific Predicate (Security Policy)

You can use application context to return the correct predicate—that is, the correct security policy.

Consider an order entry application which enforces the rules, "customers only see their own orders, and clerks see all orders for all customers." These are two different policies. You could define an application context with a `position` attribute, and this attribute could be accessed within the policy function to return the correct predicate, depending on the value of the attribute. Thus, you can enable a user in the `Clerk` position to retrieve all orders, but a user in the `Customer` position to see his own records only.

To design a fine-grained access control policy to return a specific predicate for an attribute, access the application context within the function that implements the policy. For example, to limit customers to seeing their own records only, use fine-grained access control to dynamically modify the user's query from this:

```
SELECT * FROM Orders_tab
```

to this:

```
SELECT * FROM Orders_tab
   WHERE Custno = SYS_CONTEXT ('order_entry', 'cust_num');
```

Using Application Context to Provide Attributes Similar to Bind Variables in a Predicate

Continuing with the example above, suppose you have 50,000 customers, and you do not want to have a different predicate returned for each customer. Customers all share the same policy. That is, they can only see their own orders. It is merely their customer numbers which are different.

Using application context, you can return one predicate within a policy function which applies to 50,000 customers. As a result, there is one shared cursor which nonetheless executes differently for each customer, because the customer number is evaluated at execution time. This value is, of course, different for every customer. Use of application context in this case provides optimum performance, as well as fine-grained security.

Note that the `SYS_CONTEXT` function works much like a bind variable, but only if the `SYS_CONTEXT` arguments are constants.

User Models and Virtual Private Database

Applications may have differing user models, but you may want to use virtual private database (VPD) to limit access by user. Whether the user is a database user or an application user unknown to the database, Oracle provides different ways in which applications can enforce per-user fine-grained access control.

For applications in which the application users are also database users, VPD enforcement is relatively simple; users connect to the database, and the application can set up per-session application contexts. Each session is initiated under a different username, so that it is simple to enforce different fine-grained access control conditions for users Jane and John. This is also possible with use of proxy authentication, since each "lightweight" session in OCI or thick JDBC is still a distinct database session, and can have its own application context.

Since proxy authentication can be integrated with Enterprise User Security, user roles can be retrieved from Oracle Internet Directory, as well as other attributes that can be used for VPD enforcement.

For applications in which a single user (for example, One Big Application User) connects to the database on behalf of all users, per-user fine-grained access control is still possible. An application developer can create a context attribute to represent the application user (for example, realuser). While all database sessions (and thus all audit records) are initiated as One Big Application User, each session can nonetheless have attributes that vary, depending on who the real user is. This model works best for applications with a limited number of users where there is no requirement for session reuse. Of course, each session, from the database standpoint, is created as the same database user, so that the ability to use roles, database auditing, and others is greatly diminished for reasons previously enumerated.

Web-based applications typically have hundreds if not thousands of users, and the web is stateless. There may be a persistent connection to the database (to support data retrieval for a number of user requests), but these connections are not specific to each web-based user. Web-based applications typically set up and reuse connections instead of having different sessions for each user, to provide scalability. For example, web user Jane and Ajit connect to a middle tier application, which establishes a session in the database used by the application on behalf of both users. Typically, neither Jane nor Ajit are known to the database. The application is responsible for switching the username on the connection, so that, at any given time, it’s either Jane or Ajit using the session.

Oracle9*i* VPD capabilities facilitate connection pooling by allowing multiple connections to access one or more global application contexts, instead of setting up an application context for each distinct user session.

Applications use a `CLIENT_IDENTIFIER` (which could be an individual application username, or a group) to reference the global application context. Global application contexts provide additional flexibility for web-based applications to use Virtual Private Database, as well as enhanced performance through reuse of common application contexts among multiple sessions instead of setting up per-session application contexts. The `CLIENT_IDENTIFIER` is also viewable in the user session and accessible in the USERENV naming context.

The use of a `CLIENT_IDENTIFIER` thus functions as an application user proxy, since the `CLIENT_IDENTIFIER` can be used to capture the 'application username.' The ability to pass a `CLIENT_IDENTIFIER` to the database for use with global application context is supported in OCI, thick JDBC, and thin JDBC. For OCI-based connections, a change in `CLIENT_IDENTIFIER` is automatically piggybacked on the next OCI call, for additional performance benefits.

Application user proxy authentication can be used with global application context for additional flexibility and high performance in building applications. For example, suppose a web-based application that provides information to business partners has three types of users: gold partner, silver partner, and bronze partner, representing different levels of information available. Instead of each user having his own session — with individual application contexts — set up, the application could set up global application contexts for gold partner, silver partner, or bronze partner and use the client identifier to point the session at the correct context, in order to retrieve the appropriate type of data. The application need only initialize the three global contexts once, and use the client identifier to access the correct application context to limit data access. This provides performance improvements through session reuse, and through accessing global application contexts set up once, instead of having to initialize application contexts for each session individually.

Creating a Virtual Private Database Policy with Oracle Policy Manager

Developers implementing virtual private database (VPD) can use the `DBMS_RLS` package to apply security policies to tables and views. Also, Developers can use the `CREATE CONTEXT` command to create application contexts.

Alternatively, developers can use the Oracle Policy Manager graphical user interface, accessed from Oracle Enterprise Manager, to apply security policies to schema objects, such as tables and views, and to create application contexts. Oracle

Policy Manager provides an easy-to-use interface to manage security policies and application contexts, and therefore makes VPD easier to develop.

Oracle Policy Manager is the administration tool for Oracle Label Security. Oracle Label Security provides a functional, out-of-the-box VPD policy which enhances your ability to implement row-level security. It supplies an infrastructure--a label-based access control framework--whereby you can specify labels for users and data. It also enables you to create one or more custom security policies to be used for label access decisions. You can implement these policies without any knowledge of a programming language. There is no need to write additional code; in a single step you can apply a security policy to a given table. In this way, Oracle Label Security provides a straightforward, efficient way to implement fine-grained security policies using data labeling technology. Finally, the structure of Oracle Label Security labels provides a degree of granularity and flexibility which cannot easily be derived from the application data alone. Oracle Label Security is thus a generic solution which can be used in many different circumstances.

To create VPD policies, users must provide the schema name, table (or view) name, policy name, the function name that generates the predicate, and the statement types to which the policy applies (that is, `SELECT`, `INSERT`, `UPDATE`, `DELETE`). Oracle Policy Manager then executes the function `DBMS_RLS.ADD_POLICY`. You create an application context by providing the name of the context and the package that implements the context.

See Also: *Oracle Label Security Administrator’s Guide*

How to Use Application Context

To use application context, you perform the following tasks:

- Task 1: Create a PL/SQL Package that Sets the Context for Your Application
- Task 2: Create a Unique Context and Associate It with the PL/SQL Package
- Task 3: Set the Context Before the User Retrieves Data
- Task 4. Use the Context in a Policy Function

Task 1: Create a PL/SQL Package that Sets the Context for Your Application

Begin by creating a PL/SQL package with functions that set the context for your application. This section presents an example for creating the PL/SQL package, followed by a discussion of SYS_CONTEXT syntax and behavior.

Note: A login trigger can be used because the user's context (information such as EMPNO, GROUP, MANAGER) should be set before the user accesses any data.

SYS_CONTEXT Example The following example creates the package app_security_context.

```
CREATE OR REPLACE PACKAGE App_security_context IS
   PROCEDURE Set_empno;
END;

CREATE OR REPLACE PACKAGE BODY App_security_context IS
   PROCEDURE Set_empno
   IS
   Emp_id NUMBER;
   BEGIN
    SELECT Empno INTO Emp_id FROM Emp_tab
       WHERE Ename = SYS_CONTEXT('USERENV',
                                 'SESSION_USER');
    DBMS_SESSION.SET_CONTEXT('app_context', 'empno', Emp_id);
   END;
END;
```

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference*

SYS_CONTEXT Syntax The syntax for this function is:

```
SYS_CONTEXT ('namespace', 'attribute', [length])
```

This function returns the value of `attribute` as defined in the package currently associated with the context namespace. It is evaluated once for each statement execution, and is treated like a constant during type checking for optimization. You can use the pre-defined namespace `USERENV` to access primitive contexts such as userid and NLS parameters.

Note: If you try to execute SYS_CONTEXT in a parallel query environment, you will receive a query error.

See Also: "Providing Access to Predefined Attributes Through the USERENV Namespace" on page 12-3

Oracle9i SQL Reference for details about attributes

Using Dynamic SQL with SYS_CONTEXT

Note: This feature is applicable when COMPATIBLE is set to either 8.0 or 8.1.

During a session in which you expect a change in policy between executions of a given query, that query must use dynamic SQL. You must use dynamic SQL because static SQL and dynamic SQL parse statements differently.

- Static SQL statements are parsed at compile time; for performance reasons, they are not reparsed at execution.
- Dynamic SQL statements are parsed every time they are executed.

Consider a situation in which policy A is in force when you compile a SQL statement—and then you switch to policy B and execute the statement. With static SQL, policy A remains in force: the statement is parsed at compile time and not reparsed upon execution. With dynamic SQL, however, the statement is parsed upon execution, and so the switch to policy B is carried through.

For example, consider the following policy:

```
EMPLOYEE_NAME = SYS_CONTEXT ('userenv', 'session_user')
```

The policy "Employee name matches database user name" is represented in the form of a SQL predicate: the predicate is basically a policy. If the predicate changes, the statement must be reparsed in order to produce the correct result.

See Also: "Automatic Reparse" on page 12-27

Parallel Query Requires SYS_CONTEXT Directly Within Query If SYS_CONTEXT is used inside a SQL function which is embedded in a parallel query, the function cannot pick up the application context. This is true because the application context exists only in the user session. To use these features in combination, you must call SYS_CONTEXT directly from the query.

Consider a user-defined function within a SQL statement, which sets the user's ID to 5:

```
CREATE FUNC proc1 AS RETURN NUMBER;
BEGIN
     IF SYS_CONTEXT ('hr', 'id') = 5
     THEN RETURN 1; ELSE RETURN 2;
     END
END;
```

Now consider the statement:

```
SELECT * FROM EMP WHERE proc1( ) = 1;
```

If this statement is run as a single query (that is, if one process is used to run the entire query), there will be no problem.

However, if this statement is run as a parallel query, the parallel execution servers (query slave processes) do not have access to the user session which contains the application context information. The query will not produce the expected results.

By contrast, if you use the SYS_CONTEXT function within a query, there is no problem. For example,

```
SELECT * FROM EMP WHERE SYS_CONTEXT ('hr', 'id') = 5;
```

In this way, it works like a bind variable: the query coordinator can access the application context information and pass it on to the parallel execution servers.

Versioning in Application Context When you execute a statement, Oracle9*i* takes a snapshot of the entire application context being set up by SYS_CONTEXT. Within the duration of a query, the context remains the same for all fetches of the query.

If you (or a function) attempt to change the context within a query, the change will not take effect in the current query. In this way, SYS_CONTEXT enables you to store variables in a session.

Task 2: Create a Unique Context and Associate It with the PL/SQL Package

To perform this task, use the CREATE CONTEXT statement. Each context must have a unique attribute and belong to a namespace. That is, context names must be unique within the database, not just within a schema. Contexts are always owned by the schema SYS.

For example:

```
CREATE CONTEXT order_entry USING oe_context;
```

where order_entry is the context namespace, and oe_context is the trusted package that can set attributes in the context namespace.

After you have created the context, you can set or reset the context attributes by using the DBMS_SESSION.SET_CONTEXT package. The values of the attributes you set remain either until you reset them, or until the user ends the session.

You can only set the context attributes inside the trusted procedure you named in the CREATE CONTEXT statement. This prevents a malicious user from changing context attributes without proper attribute validation.

Alternatively, you can use the Oracle Policy Manager graphical user interface to create a context and associate it with a PL/SQL package. Oracle Policy Manager, accessed from Oracle Enterprise Manager, enables you to apply policies to database objects and create application contexts. It also can be used to create and manage Oracle Label Security policies.

Task 3: Set the Context Before the User Retrieves Data

Always use an event trigger on login to pull session information into the context. This sets the user’s security-limiting attributes for the database to evaluate, and thus enables it to make the appropriate security decisions.

Other considerations come into play if you have a changing set of books, or if positions change constantly. In these cases, the new attribute values may not be picked up right away, and you must force a cursor reparse to pick them up.

See Also: "Features of Application Context" on page 12-2

"Introduction to Application Context Accessed Globally" on page 12-28

Task 4. Use the Context in a Policy Function

Now that you have set up the context and the PL/SQL package, you can go ahead and have your policy functions use the application context to make policy decisions based on different context values.

Examples: Application Context Within a Fine-Grained Access Control Function

This section provides three examples that use application context within a fine-grained access control function.

- Example 1: Implementing the Policy
- Example 2: Controlling User Access by Way of an Application
- Example 3: Event Triggers, Application Context, Fine-Grained Access Control, and Encapsulation of Privileges

Example 1: Implementing the Policy

This example uses application context to implement the policy, "Customers can see their own orders only."

This example guides you through the following steps in building the application:

- Step 1. Create a PL/SQL Package Which Sets the Context for the Application
- Step 2. Create an Application Context
- Step 3. Access the Application Context Inside the Package
- Step 4. Create the New Security Policy

The procedure in this example assumes a one-to-one relationship between users and customers. It finds the user's customer number (`Cust_num`), and caches the customer number in the application context. You can later refer to the `cust_num` attribute of your order entry context (`order_entry_ctx`) inside the security policy function.

Note that you could use a login trigger to set the initial context.

Step 1. Create a PL/SQL Package Which Sets the Context for the Application

Create the package as follows:

```
CREATE OR REPLACE PACKAGE apps.oe_ctx AS
   PROCEDURE set_cust_num ;
END;

CREATE OR REPLACE PACKAGE BODY apps.oe_ctx AS
   PROCEDURE set_cust_num IS
     custnum NUMBER;
   BEGIN
      SELECT cust_no INTO custnum FROM customers WHERE username =
         SYS_CONTEXT('USERENV', 'session_user');
     /* SET cust_num attribute in 'order_entry' context */
         DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
         DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
   END set_cust_num;
 END;
```

Note: This example does not treat error handling.

You can access predefined attributes—such as session user—by using `SYS_CONTEXT('userenv', `*`session_primitive`*`)`.

For more information, see *Oracle9i SQL Reference*

Step 2. Create an Application Context

Create an application context by entering:

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

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

Step 3. Access the Application Context Inside the Package

Access the application context inside the package that implements the security policy on the database object.

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

```
CREATE PACKAGE Oe_security AS
FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2)
RETURN VARCHAR2;
END;
```

The package body appends a dynamic predicate to SELECT statements on the ORDERS_TAB table. This predicate limits the orders returned to those of the user's customer number by accessing the cust_num context attribute, instead of a subquery to the customers table.

```
CREATE OR REPLACE PACKAGE BODY Oe_security AS

/* limits select statements based on customer number: */
FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2) RETURN VARCHAR2
IS
    D_predicate VARCHAR2 (2000)
    BEGIN
     D_predicate = 'cust_no = SYS_CONTEXT("order_entry", "cust_num")';
     RETURN D_predicate;
    END Custnum_sec;
END Oe_security;
```

Step 4. Create the New Security Policy

Create the policy as follows:

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

```
CONNECT sys/change_on_install AS sysdba;
CREATE USER secusr IDENTIFIED BY secusr;
```

```
DBMS_RLS.ADD_POLICY ('scott', 'orders_tab', 'oe_policy', 'secusr',
                     'oe_security.custnum_sec', 'select')
```

This statement adds a policy named OE_POLICY to the ORDERS_TAB table for viewing in schema SCOTT. The SECUSR.OE_SECURITY.CUSTNUM_SEC function implements the policy, is stored in the SECUSR schema, and applies to SELECT statements only.

Now, any select statement by a customer on the ORDERS_TAB table automatically returns only that customer's orders. In other words, the dynamic predicate modifies the user's statement from this:

```
SELECT * FROM Orders_tab;
```

to this:

```
SELECT * FROM Orders_tab
   WHERE Custno = SYS_CONTEXT('order_entry','cust_num');
```

Note the following with regard to this example:

- In reality, you might have several predicates based on a user's position. For example, a sales representative would be able to see records only for his customers, and an order entry clerk would be able to see any customer order. You could expand the custnum_sec function to return different predicates based on the user's position context value.
- The use of application context in a fine-grained access control package effectively gives you a bind variable in a parsed statement. For example:

  ```
  SELECT * FROM Orders_tab
     WHERE Custno = SYS_CONTEXT('order_entry', 'cust_num')
  ```

  This is fully parsed and optimized, but the evaluation of the user's CUST_NUM attribute value for the ORDER_ENTRY context takes place at execution. This means that you get the benefit of an optimized statement which executes differently for each user who executes the statement.

  Note: You can improve the performance of the function in this example even more by indexing CUST_NO.

- You could set your context attributes based on data from a database table or tables, or from a directory server using LDAP (Lightweight Directory Access Protocol).

See Also: Compare and contrast this example, which uses an application context within the dynamically generated predicate, with "How Fine-Grained Access Control Works" on page 12-44, which uses a subquery in the predicate

Chapter 15, "Using Triggers"

Example 2: Controlling User Access by Way of an Application

This example uses application context to control user access by way of a Human Resources application. It guides you through the following three tasks, each of which is described more fully below.

- Step 1. Create a PL/SQL Package to Set the Context
- Step 2. Create the Context and Associate It with the Package
- Step 3. Create the Initialization Script for the Application

In this example, assume that the application context for the Human Resources application is assigned to the HR_CTX namespace.

Step 1. Create a PL/SQL Package to Set the Context

Create a PL/SQL package with a number of functions that set the context for the application

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

```
CREATE OR REPLACE PACKAGE apps.hr_sec_ctx IS
   PROCEDURE set_resp_id (respid NUMBER);
   PROCEDURE set_org_id (orgid NUMBER);
  /* PROCEDURE validate_respid (respid NUMBER); */
  /* PROCEDURE validate_org_id (orgid NUMBER); */
END hr_sec_ctx;
```

APPS is the schema owning the package.

```
CREATE OR REPLACE PACKAGE BODY apps.hr_sec_ctx IS
/* function to set responsibility id */
PROCEDURE set_resp_id (respid NUMBER) IS
BEGIN
```

```
/* validate respid based on primitive and other context */
/*    validate_respid (respid); */

/* set resp_id attribute under namespace 'hr_ctx'*/
    DBMS_SESSION.SET_CONTEXT('hr_ctx', 'resp_id', respid);
END set_resp_id;

/* function to set organization id */
PROCEDURE set_org_id (orgid NUMBER) IS

BEGIN
/* validate organization ID */
/*    validate_org_id(orgid); /*
/* set org_id attribute under namespace 'hr_ctx' */
    DBMS_SESSION.SET_CONTEXT('hr_ctx', 'org_id', orgid);
END set_org_id;

/* more functions to set other attributes for the HR application */
END hr_sec_ctx;
```

Step 2. Create the Context and Associate It with the Package

For example:

```
CREATE CONTEXT Hr_ctx USING Apps.Hr_sec_ctx;
```

Step 3. Create the Initialization Script for the Application

Suppose that the execute privilege on the package `HR_SEC_CTX` has been granted to the schema running the application. Part of the script will make calls to set various attributes of the `HR_CTX` context. Here, we do not show how the context is determined. Normally, it is based on the primitive context or other derived context.

```
APPS.HR_SEC_CTX.SET_RESP_ID(1);
APPS.HR_SEC_CTX.SET_ORG_ID(101);
```

The `SYS_CONTEXT` function can be used for data access control based on this application context. For example, the base table `HR_ORGANIZATION_UNIT` can be secured by a view that restricts access to rows based on attribute `ORG_ID`:

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

```
CREATE TABLE hr_organization_unit (organization_id NUMBER);
CREATE VIEW Hr_organization_secv AS
   SELECT * FROM hr_organization_unit
      WHERE Organization_id = SYS_CONTEXT('hr_ctx','org_id');
```

Example 3: Event Triggers, Application Context, Fine-Grained Access Control, and Encapsulation of Privileges

This example illustrates use of the following security features in Oracle9*i*:

- Event triggers
- Application context
- Fine-grained access control
- Encapsulation of privileges in stored procedures

In this example, we associate a security policy with the table called `DIRECTORY` which has the following columns:

`EMPNO`	identification number for each employee
`MGRID`	employee identification number for the manager of each employee
`RANK`	position of the employee in the corporate hierarchy

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,
   Empno  NUMBER,
   Name   VARCHAR2(20));
CREATE TABLE Directory_u(
   Empno NUMBER,
   Mgrno NUMBER,
   Rank  NUMBER);
CREATE SEQUENCE Empno_seq
CREATE SEQUENCE Rank_seq
```

The security policy associated with this table has two elements:

- All users can find the MGRID for a specific EMPNO. To implement this, we create a definer's right package in the Human Resources schema (HR) to perform SELECT on the table.
- Managers can update the positions in the corporate hierarchy of only their direct subordinates. To do this they must use only the designated application. You can implement this as follows:
  - Define fine-grained access policies on the table based on EMPNO and application context.
  - Set EMPNO by using a logon trigger.
  - Set the application context by using the designated package for processing the updates (event triggers and application context).

Note: In this example, we grant UPDATE privileges on the table to PUBLIC, because fine-grained access control prevents an unauthorized user from wrongly modifying a given row.

```
CONNECT system/manager AS sysdba
GRANT CONNECT,RESOURCE,UNLIMITED TABLESPACE,CREATE ANY CONTEXT, CREATE
PROCEDURE, CREATE ANY TRIGGER TO HR IDENTIFIED BY HR;
CONNECT hr/hr;
CREATE TABLE Directory (Empno   NUMBER(4) NOT NULL,
                        Mgrno   NUMBER(4) NOT NULL,
                        Rank    NUMBER(7,2) NOT NULL);

CREATE TABLE Payroll (Empno  NUMBER(4) NOT NULL,
                      Name   VARCHAR(30) NOT NULL );

/* seed the tables with a couple of managers: */
INSERT INTO Directory VALUES (1, 1, 1.0);
INSERT INTO Payroll VALUES (1, 'KING');
INSERT INTO Directory VALUES (2, 1, 5);
INSERT INTO Payroll VALUES (2, 'CLARK');

/* Create the sequence number for EMPNO: */
CREATE SEQUENCE Empno_seq START WITH 5;

/* Create the sequence number for RANK:  */
CREATE SEQUENCE Rank_seq START WITH 100;

CREATE OR REPLACE CONTEXT Hr_app USING Hr.Hr0_pck;
CREATE OR REPLACE CONTEXT Hr_sec USING Hr.Hr1_pck;

CREATE or REPLACE PACKAGE Hr0_pck IS
PROCEDURE adjustrankby1(Empno NUMBER);
END;

CREATE or REPLACE PACKAGE BODY Hr0_pck IS
/* raise the rank of the empno by 1:  */
PROCEDURE Adjustrankby1(Empno NUMBER)
IS
   Stmt   VARCHAR2(100);
   BEGIN

   /*Set context to indicate application state */
   DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',1);
   /* Now we can issue DML statement:  */
   Stmt := 'UPDATE SET Rank := Rank +1 FROM Directory d WHERE d.Empno = '
   || Empno;
   EXECUTE IMMEDIATE STMT;

/* Re-set application state: */
   DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',0);
   END;
END;

CREATE or REPLACE PACKAGE hr1_pck IS PROCEDURE setid;
END;
/
/* Based on userid, find EMPNO, and set it in application context */

CREATE or REPLACE PACKAGE BODY Hr1_pck IS
PROCEDURE setid
  IS
  id NUMBER;
  BEGIN
    SELECT Empno INTO id FROM Payroll WHERE Name =
      SYS_CONTEXT('userenv','session_user') ;
    DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
    DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
  EXCEPTION
    /* For purposes of demonstration insert into payroll table
    /  so that user can continue on and run example. */
    WHEN NO_DATA_FOUND THEN
      INSERT INTO Payroll (Empno, Name)
         VALUES (Empno_seq.NEXTVAL, SYS_CONTEXT('userenv','session_user'));
      INSERT INTO Directory (Empno, Mgrno, Rank)
         VALUES (Empno_seq.CURRVAL, 2, Rank_seq.NEXTVAL);
      SELECT Empno INTO id FROM Payroll WHERE Name =
        sys_context('userenv','session_user') ;
      DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
      DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
    WHEN OTHERS THEN
      NULL;
    /* If this is to be fired via a "logon" trigger,
    /  you need to handle exceptions if you want the user to continue
    /  logging into the database. */
  END;
END;

GRANT EXECUTE ON Hr1_pck TO public;

CONNECT system/manager AS sysdba

CREATE OR REPLACE TRIGGER Databasetrigger

AFTER LOGON
ON DATABASE
BEGIN
   hr.Hr1_pck.Setid;
END;

/* Creates the package for finding the MGRID for a particular EMPNO
using definer's right (encapsulated privileges). Note that users are
granted EXECUTE privileges only on this package, and not on the table
(DIRECTORY) it is querying. */

CREATE or REPLACE PACKAGE hr2_pck IS
   FUNCTION Findmgr(Empno NUMBER) RETURN NUMBER;
END;

CREATE or REPLACE PACKAGE BODY hr2_pck IS
   /* insert a new employee record: */
   FUNCTION findmgr(empno number) RETURN NUMBER IS
   Mgrid NUMBER;
   BEGIN
      SELECT mgrno INTO mgrid FROM directory WHERE mgrid = empno;
   RETURN mgrid;
   END;
END;
```

```
CREATE or REPLACE FUNCTION secure_updates(ns varchar2,na varchar2)
  RETURN VARCHAR2 IS
     Results VARCHAR2(100);
    BEGIN
       /* Only allow updates when designated application has set the session
       state to indicate we are inside it. */
       IF (sys_context('hr_sec','adjstate') = 1)
          THEN results := 'mgr = SYS_CONTEXT("hr_sec","empno")';
       ELSE results := '1=2';
      END IF;
      RETURN Results;
   END;

/* Attaches fine-grained access policy to all update operations on
hr.directory */

CONNECT system/manager AS sysdba;
BEGIN
   DBMS_RLS.ADD_POLICY('hr','directory_u','secure_update','hr',
                       'secure_updates','update',TRUE,TRUE);
END;
```

Automatic Reparse

Note: This feature is applicable when `COMPATIBLE` is set to 9.0.1.

Any SQL table or view, that is VPD-policy based, will enable automatic reparse to support dynamic adjustments to policies. Oracle will execute a policy and get the latest policy.

For example, users can develop policies that are based on time of day. When a parsed cursor is executed, Oracle will execute a policy function automatically to get the up-to-date predicate.

Note: For policy function that returns the same predicate, we recommend that you mark the policy function `DETERMINISTIC` to enhance.

See Also: "Using Dynamic SQL with SYS_CONTEXT" on page 12-14

Introduction to Application Context Accessed Globally

In many application architectures, the middle tier application is responsible for managing session pooling for application users. That is, users authenticate themselves to the application, which uses a single identity to log into the database and maintains all the connections. In this environment, it is not possible to maintain application attributes using session-dependent secure application context because of the sessionless model of the application.

Another scenario is when a user is connected to the database through an application (such as Oracle Forms) which then spawns other applications (such as Oracle Reports) to connect to the database. These applications may need to share the session attributes such that they appear to be sharing the same database session.

Global application context is a type of secure application context that can be shared among trusted sessions. In addition to driving the enforcement of the fine-grained access control policies, applications (especially middle-tier products) can use this support to manage application attributes securely and globally.

Note: Global application context is not available in Real Application Clusters.

Using the DBMS_SESSION Interface to Manage Application Context in Client Sessions

The `DBMS_SESSION` interface for managing application context has a client identifier for each application context. In this way, application context can be managed globally, yet each client sees only his or her assigned application context. The following interfaces in `DBMS_SESSION` enable the administrator to manage application context in client sessions:

- `SET_CONTEXT`
- `CLEAR_CONTEXT`
- `SET_IDENTIFIER`
- `CLEAR_IDENTIFIER`

The middle-tier application server can use SET_CONTEXT to set application context for a specific client ID. Then, when assigning a database connection to process the client request, the application server needs to issue a SET_IDENTIFIER to denote the ID of the application session. From then on, every time the client invokes SYS_CONTEXT, only the context that was associated with the set identifier is returned. In other words, the application server uses SET_IDENTIFIER to associate the database session with a particular user or a group. Then, the CLIENT_IDENTIFIER is an attribute of the session and can be viewed in session information. Also, CLIENT_IDENTIFIER is the key to accessing the global application context. For example, suppose a web-based application that provides information to business partners has three types of users: gold partner, silver partner, and bronze partner. These users represent different levels of available information. Instead of each user having their own setup session with application contexts, the application could set up global application contexts for gold partner, silver partner, and bronze partner. Afterwards, one can do the following:

- Use SET_IDENTIFIER to set a particular session to gold partner, silver partner, or bronze partner.
- Associate the session with the correct global application context, in order to retrieve the appropriate data for gold, silver, and bronze partners, respectively.

The application need only initialize the three global contexts once, and use CLIENT_IDENTIFIER to access the correct application context to limit data access. This provides performance improvements through session reuse, and through accessing global application contexts setup once, instead of having to initialize application contexts for each session.

Example 1: Global Access of Application Context

For an application context accessed globally, the scenario is as follows:

1. Consider an application server that has assigned the client identifier 12345 to client SCOTT. It then issues the following statement to indicate that, for this client identifier, there is an application context RESPONSIBILITY with a value of 13 in the HR namespace.

   ```
   DBMS_SESSION.SET_CONTEXT( 'HR', 'RESPONSIBILITY' , '13', 'SCOTT', '12345' );
   ```

   Note that HR must be a global context namespace created as follows:

   ```
   CREATE CONTEXT hr USING hr.init ACCESSED GLOBALLY;
   ```

2. Then, for each client session using APPSMGR to establish a connection to database, the following command should be issued when client SCOTT is assigned to a new database session to indicate identity:

   ```
   DBMS_SESSION.SET_IDENTIFIER('12345');
   ```

3. Within the database session, when there is a SYS_CONTEXT('HR','RESPONSIBILITY') call, the database engine will match the client identifier 12345 to the global context, and return the value 13.
4. When exiting this database session, middle tier can clear the client identifier by issuing:

   ```
   DBMS_SESSION.CLEAR_IDENTIFIER( );
   ```

After the client identifier in a session is clear, it takes on a NULL value, implying that any subsequent SYS_CONTEXT call will only retrieve application context with a NULL client identifier, until the client identifier is set again using the SET_IDENTIFIER interface.

Note: Versioning is not available for application context accessed globally. This will return a point in time SYS_CONTEXT value. Since multiple client sessions could be accessing the same global application context values at any time, versioning is not possible. Simple application context is per session, and can be versioned.

Example 2: Global Access of Application Context for Proxy Authentication Applications

For a proxy authentication application, the scenario is as follows:

1. The administrator creates the global context namespace by issuing:

   ```
   CREATE CONTEXT hr USING hr.init ACCESSED GLOBALLY;
   ```

2. The HR application server (AS) starts up and establishes multiple connections to the HR database as user APPSMGR.
3. User SCOTT logs on to the HR application server.
4. AS authenticates SCOTT into the application.
5. AS assigns a temporary session ID (or simply uses the application user ID), 12345, for this connection.

6. The session ID is returned to SCOTT's browser as part of a cookie or maintained by AS.

   Note: If the application generates a session ID for use as a CLIENT_IDENTIFIER, the session ID must be suitably random, and protected over the network through encryption. If the session ID is not random, then a malicious user could guess the session ID and access another user's data. If the session ID is unencrypted over the network, then a malicious user could retrieve the session ID and access the connection.

7. AS initializes application context for this client calling the HR.INIT package, which issues:

   ```
   DBMS_SESSION.SET_CONTEXT( 'hr', 'id', 'scott', 'APPSMGR', 12345 );
   DBMS_SESSION.SET_CONTEXT( 'hr', 'dept', 'sales', 'APPSMGR', 12345 );
   ```

8. AS assigns a database connection to this session, and initializes the session by issuing:

   ```
   DBMS_SESSION.SET_IDENTIFIER( 12345 );
   ```

9. All SYS_CONTEXT calls within this database session will return application context values belonging to the client session only. For example, SYS_CONTEXT('hr','id') will return the value SCOTT.
10. When done with the session, AS can issue the following statement to clean up the client identity:

    ```
    DBMS_SESSION.CLEAR_IDENTIFIER ( );
    ```

Note that even if another database user (ADAMS) had logged into the database, he cannot access the global context set by AS because AS has specified that only the application with logged in user APPSMGR can see it. If AS has used the following, then any user session with client ID set to 12345 can see the global context.

```
DBMS_SESSION.SET_CONTEXT( 'hr', 'id', 'scott', NULL , 12345 );
DBMS_SESSION.SET_CONTEXT( 'hr', 'dept', 'sales', NULL , 12345 );
```

This approach enables different users to share the same context.

The users, however, should be aware of the security implication of different settings of the global context. Basically, NULL in the username means that any user can access the global context. A `NULL` client ID in the global context means that only a session with an uninitialized client ID can access the global context.

Users can query the client identifier set in the session as follows:

```
SYS_CONTEXT('USERENV','CLIENT_IDENTIFIER')
```

The DBA can see which sessions have the client identifier set by querying the `V$SESSION` view's `CLIENT_IDENTIFIER` and `USERNAME`.

When a user wants to see how much global context area (in bytes) is being used, she can use `SYS_CONTEXT('USERENV', 'GLOBAL_CONTEXT_MEMORY')`

> **See Also:** *Oracle9i SQL Reference*
>
> *Oracle9i Supplied PL/SQL Packages and Types Reference*
>
> *Oracle9i JDBC Developer's Guide and Reference* and *Oracle Call Interface Programmer's Guide* for client identifier information

Initializing Application Context Externally

This feature lets you specify a special type of namespace that accepts initialization of attribute values from external resources. This enhances performance and enables the automatic propagation of attributes from one session to the other. For example, many organizations want to manage user information centrally, in an LDAP-based directory. Oracle9*i* Enterprise User Security feature supports centralized user and authorization management in Oracle Internet Directory. However, there may be additional attributes an application wishes to retrieve from LDAP to use for VPD enforcement:

- The user's title
- The users' organization
- The user's physical location

The ability to initialize application context from external sources such as LDAP helps organizations leverage existing information they have for VPD enforcement, that is centrally managed, without requiring replication or duplication of this information in database tables.

This section contains these topics:

- Obtaining Default Values from Users
- Obtaining Values from Other External Resources

- Obtaining Values for Users Not Known to the Database

Obtaining Default Values from Users

In some situations it is desirable to obtain default values from users. These default values may serve as hints or preferences initially, and may become trusted context after the values are validated. Similarly, clients may want a convenient way to initialize some default values, and then rely on a login event trigger or applications to validate the values.

For job queues, administrators may expect the job submission routine to record all the context being set at the time the job is submitted, and restore it when executing the batched job. To maintain the integrity of context, job queues cannot bypass the designated PLSQL package to set the context. Rather, externally initialized application context accepts initialization of context values from the job queue process.

Whereas automatic propagation of context to a remote session may create security problems, developers or administrators can effectively handle this new type of context that takes default values from resources other than the designated PLSQL procedure. In addition, performance is enhanced because this feature provides an extensible interface for the OCI client to bundle more information to the server in one `OCISessionBegin()` call.

Obtaining Values from Other External Resources

In addition to using the designated trusted package, externally initialized application context can also accept initialization of attributes and values through external resources such as an OCI interface, a job queue process, or a database link. It provides:

- For remote sessions, automatic propagation of context values that are in external initialized context namespace
- For job queues, restoration of context values that are in externally initialized context namespace
- For OCI interface, a mechanism to initialize context values that are in externally initialized context namespace

Although this new type of namespace can be initialized by any client program using OCI, there are login event triggers that can verify the values. It is up to the application to interpret and trust the values of the attributes.

Note that with this feature, the middle-tier server can actually initialize context values on behalf of database users. Context attributes are propagated for the remote session at initiation time, and the remote database accepts the values if the namespace is externally initialized.

Obtaining Values for Users Not Known to the Database

Externally initialized application context is especially useful for cases in which users are not known to the database. In these situations, the application typically connects as a single database user, and all actions are taken as that user. Since all user sessions are created as the same user, this security model normally makes it very difficult, if not impossible, to use the virtual private database capability to achieve per user or per customer data separation. However, these applications can use the client identifier as an application user proxy. In this way, the application uses the client identifier to proxy the "real" application user name to the database.

This approach has several advantages. With application user proxy, the sessions can be reused by multiple users merely by changing the client identifier (which here is employed to capture the name of the real application user). This avoids the overhead of setting up a separate session and separate attributes for the user, and enables reuse of sessions by the application merely by changing the client identifier (to represent the new application user name). When a client changes the client identifier, the change is piggybacked on the next OCI (or thick JDBC) call, for additional performance gains. Application user proxy (via client identifier) is available in available in OCI, thick JDBC, and thin JDBC.

Suppose, for example, that user Daniel connects to a Web Expense application. Daniel is not a database user, he is a typical Web Expense application user. The application sets up a global application context for a typical web user and sets `DANIEL` as the client identifier. Daniel completes his Web Expense form and exits the application. Ajit now connects to the Web Expense application. Instead of setting up a new session for Ajit, the application reuses the session that currently exists for Daniel, merely by changing the client identifier to `AJIT`. This avoids both the overhead of setting up a new connection to the database, and the overhead of initializing a new application context.

Note that the client identifier can be anything the application wishes to base access control upon; it need not be an application username.

Another way in which the client identifier can be used for applications whose users are not database users, is to use the client identifier as a type of group or role mechanism. For example, suppose a Marketing application has three types of users: standard partners, silver partners, and gold partners. The application could use the global application context feature to set up three types of contexts (standard, silver, and gold). The application then determines which type of partner a user is, and, passes the client identifier to the database for a session. The client identifier (standard, silver, or gold) here acts like a pointer to the correct application context. There may be multiple sessions that are *silver*, for example, and yet they all share the same application context.

See Also:

Oracle9i JDBC Developer's Guide and Reference

Oracle Call Interface Programmer's Guide

Initializing Application Context Globally

This feature provides a centralized location to store the user's application context, enabling applications to set up the user's contexts during initialization based upon the user's identity. In particular, it supports Oracle Label Security labels and privileges. This feature makes it much easier for the administrator to manage contexts for large numbers of users and databases.

This section contains these topics:

- Application Context Utilizing LDAP
- How Globally Initialized Application Context Works
- Example: Initializing Application Context Globally

Application Context Utilizing LDAP

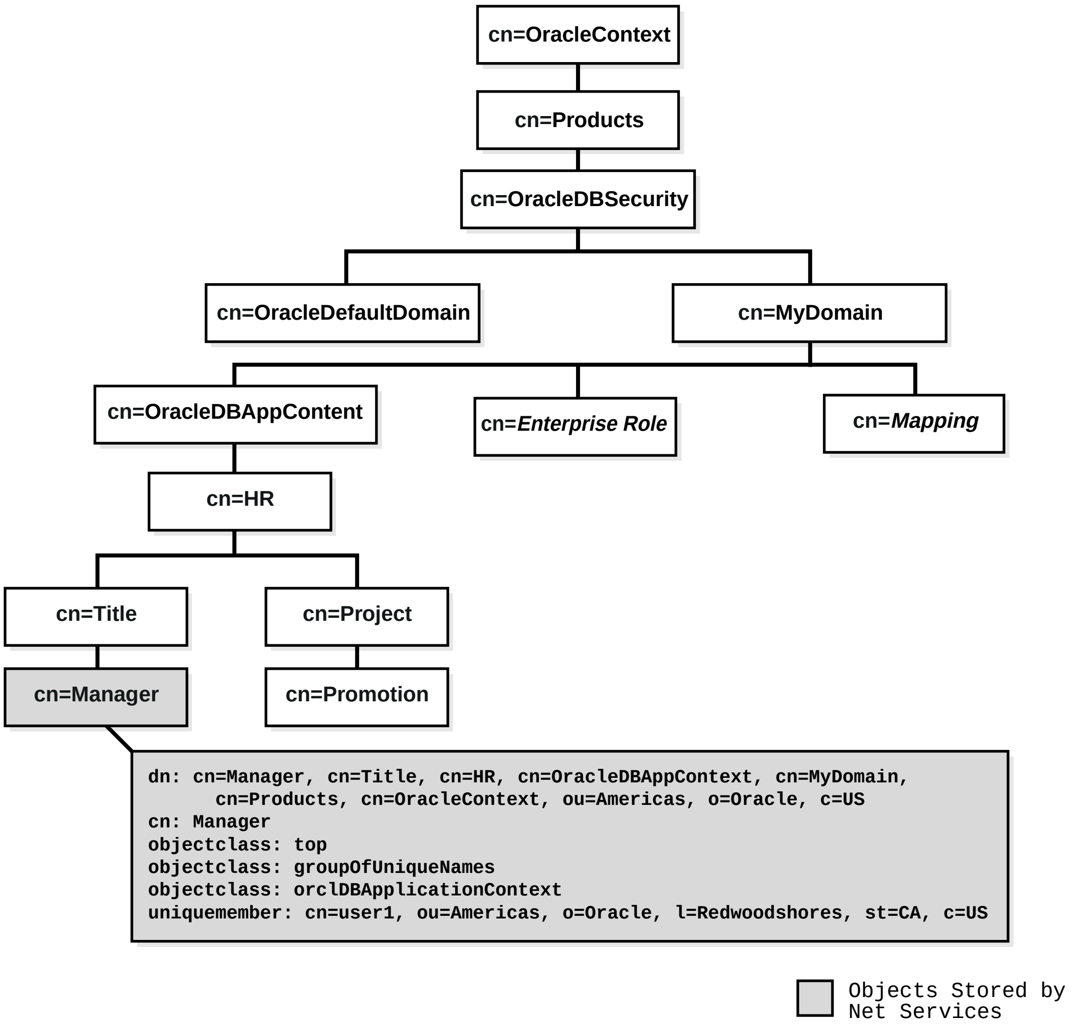
Application context initialized globally utilizes the Lightweight Directory Access Protocol (LDAP). LDAP is a standard, extensible, and efficient directory access protocol. The LDAP directory stores a list of users to which this application is assigned. Oracle9*i* can use Oracle Internet Directory as the directory service for authentication and authorization of enterprise users. (Note that enterprise user security requires Oracle Advanced Security.)

The LDAP object `orclDBApplicationContext` (a subclass of `groupOfUniqueNames`) has been defined to store the application context values in the directory. The location of the application context object is described in Figure 12–1, which is based upon the Human Resources example.

Note that an internal C function is required to retrieve the `orclDBApplicationContext` value. A list of application context values is returned to RDBMS.

Note: In this example, HR is the namespace, Title and Project are the attributes, and Manager and Promotion are the values.

Figure 12–1 Location of Application Context in LDAP Directory Information Tree (DIT)



```
dn: cn=Manager, cn=Title, cn=HR, cn=OracleDBAppContext, cn=MyDomain,
      cn=Products, cn=OracleContext, ou=Americas, o=Oracle, c=US
cn: Manager
objectclass: top
objectclass: groupOfUniqueNames
objectclass: orclDBApplicationContext
uniquemember: cn=user1, ou=Americas, o=Oracle, l=Redwoodshores, st=CA, c=US
```

How Globally Initialized Application Context Works

The administrator sets up the user's global application context values at the database and the directory.

When a global user connects to the database, the Oracle Advanced Security option performs authentication to verify the identity of the user connecting to the database. Once the identification is completed, the user's global roles are retrieved from LDAP. Then the user's global application context is retrieved from LDAP. Thus, when the user logs on to the database, her global roles and initial application context are already set up.

Example: Initializing Application Context Globally

The initial application context for a user, such as department name, level (title) can be set up and stored in the LDAP directory. The values are retrieved during user login so that the context is set properly. In addition, any information related to the user is retrieved and stored in the application context namespace SYS_USER_DEFAULTS. The following example shows how this is done.

1. Create an application context in the database.

   ```
   CREATE CONTEXT HR USING hrapps.hr_manage_pkg INITIALIZED GLOBALLY;
   ```

2. Create and add new entries in the LDAP directory.

   An example of the entries added to the LDAP directory follows. These entries create an attribute name Title with attribute value Manager for the application (namespace) HR, and assign usernames user1 and user2.

   ```
   dn:
   cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Products,cn=OracleC
   ontext,ou=Americas,o=oracle,c=US
   changetype: add
   cn: OracleDBAppContext
   objectclass: top
   objectclass: orclContainer

   dn:
   cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Products,cn=O
   racleContext,ou=Americas,o=oracle,c=US
   changetype: add
   cn: HR
   objectclass: top
   objectclass: orclContainer
   ```

```
dn:
cn=Title,cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Prod
ucts,cn=OracleContext,ou=Americas,o=oracle,c=US
changetype: add
cn: Title
objectclass: top
objectclass: orclContainer

dn:
cn=Manager,cn=Title,cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecur
ity,cn=Products,cn=OracleContext,ou=Americas,o=oracle,c=US
cn: Manager
objectclass: top
objectclass: groupofuniquenames
objectclass: orclDBApplicationContext
uniquemember: CN=user1,OU=Americas,O=Oracle,L=Redwoodshores,ST=CA,C=US
uniquemember: CN=user2,OU=Americas,O=Oracle,L=Redwoodshores,ST=CA,C=US
```

3. If an LDAP `inetOrgPerson` object entry exists for the user, the connection will also retrieve all the attributes from `inetOrgPerson` and assign them to the namespace `SYS_LDAP_USER_DEFAULT`. The following is an example of an `inetOrgPerson` entry:

```
dn: cn=user1,ou=Americas,O=oracle,L=redwoodshores,ST=CA,C=US
changetype: add
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: user1
sn: One
givenName: User
initials: UO
title: manager, product development
uid: uone
mail: uone@us.oracle.com
telephoneNumber: +1 650 123 4567
employeeNumber: 00001
employeeType: full time
```

4. Connect to the database.

   When `user1` connects to a database that belongs to domain `myDomain`, `user1` will have his `Title` set to `Manager`. Any information related to `user1` will be retrieved from the LDAP directory. The value can be obtained using the syntax

   ```
   SYS_CONTEXT('namespace','attribute name')
   ```

   For example:

   ```
   DECLARE
   tmpstr1 VARCHAR2(30);
   tmpstr2 VARCHAR2(30);
   BEGIN
   tmpstr1 = SYS_CONTEXT('HR','TITLE);
   tmpstr2 = SYS_CONTEXT('SYS_LDAP_USER_DEFAULT','telephoneNumber');
   DBMS_OUTPUT.PUT_LINE('Title is ' || tmpstr1);
   DBMS_OUTPUT.PUT_LINE('Telephone Number is ' || tmpstr2);
   END;
   ```

   The output of the above example is:

   ```
   Title is Manager
   Telephone Number is +1 650 123 4567
   ```

Introduction to Fine-Grained Access Control

Fine-grained access control allows you to build applications that enforce security policies at a low level of granularity. You can use it, for example, to restrict a customer who is accessing an Oracle server to see only his own account, a physician to see only the records of her own patients, or a manager to see only the records of employees who work for him.

When you use fine-grained access control, you create security policy functions attached to the table or view on which you have based your application. Then, when a user enters a DML statement (SELECT, INSERT, UPDATE, or DELETE) on that object, Oracle dynamically modifies the user's statement—transparently to the user—so that the statement implements the correct access control.

This section covers:

- Features of Fine-Grained Access Control
- How Fine-Grained Access Control Works
- How to Establish Policy Groups
- How to Add a Policy to a Table or View
- How to Check for Policies Applied to Statement
- EXEMPT ACCESS POLICY System Privilege
- Use of Ad Hoc Tools a Potential Security Problem

Features of Fine-Grained Access Control

Fine-grained access control provides the following capabilities:

- Table- Or View-Based Security Policies
- Multiple Policies for Each Table or View
- Grouping of Security Policies
- High Performance
- Default Security Policies

Table- Or View-Based Security Policies

Attaching security policies to tables or views, rather than to applications, provides greater security, simplicity, and flexibility.

Security	Attaching a policy to a table or view overcomes a potentially serious application security problem. Suppose a user is authorized to use an application, and then, drawing on the privileges associated with that application, wrongfully modifies the database by using an ad hoc query tool, such as SQL*Plus. By attaching security policies to tables or views, fine-grained access control ensures that the same security is in force, no matter how a user accesses the data.
Simplicity	Adding the security policy to the table or view means that you make the addition only once, rather than repeatedly adding it to each of your table- or view-based applications.
Flexibility	You can have one security policy for `SELECT` statements, another for `INSERT` statements, and still others for `UPDATE` and `DELETE` statements. For example, you might want to enable a Human Resources clerk to `SELECT` all employee records in her division, but to `UPDATE` only salaries for those employees in her division whose last names begin with "A" through "F".

Note: Although you can define a policy against a table, you cannot select that table from within the policy that was defined against the table.

Multiple Policies for Each Table or View

You can establish several policies for the same table or view. Suppose, for example, you have a base application for Order Entry, and each division of your company has its own special rules for data access. You can add a division-specific policy function to a table without having to rewrite the policy function of the base application.

Note that all policies applied to a table are enforced with `AND` syntax. Thus, if you have three policies applied to the `CUSTOMERS` table, each policy is applied to any access of the table. You can use policy groups and a driving application context to partition fine-grained access control enforcement so that different policies apply, depending upon which application is accessing data. This eliminates the requirement for development groups to collude on policies and simplifies application development. You can also have a default policy group that always applies (for example, to enforce data separate by subscriber, in a hosting environment).

Grouping of Security Policies

Since multiple applications, with multiple security policies, can share the same table or view, it is important to identify those policies which should be in effect when the table or view is accessed.

For example, in a hosting environment, Company A can host the `BENEFIT` table for Company B and Company C. The table is accessed by two different applications, HUMAN RESOURCES and FINANCE, with two different security policies. The HUMAN RESOURCES application authorizes users based on ranking in the company, and the FINANCE application authorizes users based on department. To integrate these two policies into the BENEFIT table would require joint development of policies between the two companies, which is not a feasible option. By defining an application context to drive the enforcement of a particular set of policies to the base objects, each application can implement a private set of security policies.

To do this, you can organize security policies into groups. By referring to the application context, the Oracle server determines which group of policies should be in effect at runtime. The server enforces all the policies which belong to that policy group.

High Performance

With fine-grained access control, each policy function for a given query is evaluated only once, at statement parse time. Also, the entire dynamically modified query is optimized and the parsed statement can be shared and reused. This means that rewritten queries can take advantage of Oracle's high performance features, such as dictionary caching and shared cursors.

Default Security Policies

While partitioning security policies by application is desirable, it is also useful to have security policies that are always in effect. In the previous example, a hosted application can always enforce data separation by `subscriber_ID`, whether you are using the Human Resources application or the Finance application. Default security policies allow developers to have base security enforcement under all conditions, while partitioning of security policies by application (using security groups) enables layering of additional, application-specific security on top of default security policies. To implement default security policies, you add the policy to the `SYS_DEFAULT` policy group.

How Fine-Grained Access Control Works

Fine-grained access control is based on dynamically modified statements, similar to the example described in this section. Suppose you want to attach to the `ORDERS_TAB` table the following security policy: "Customers can see only their own orders." The process is described in this section.

1. Create a function to add a predicate to a user's DML statement.

   Note: A predicate is the `WHERE` clause and, more explicitly, a selection criterion clause based on one of the operators (=, !=, `IS`, `IS NOT`, >, >=).

   In this case, you might create a function that adds the following predicate:

   ```
   Cust_no = (SELECT Custno FROM Customers WHERE Custname =
             SYS_CONTEXT ('userenv','session_user'))
   ```

2. A user enters the statement:

   ```
   SELECT * FROM Orders_tab;
   ```

3. The Oracle server calls the function you created to implement the security policy.
4. The function dynamically modifies the user's statement to read:

```
SELECT * FROM Orders_tab WHERE Custno = (
   SELECT Custno FROM Customers
       WHERE Custname = SYS_CONTEXT('userenv', 'session_user'))
```

5. The Oracle server executes the dynamically modified statement.

Upon execution, the function employs the username returned by `SYS_CONTEXT ('userenv','session_user')` to look up the corresponding customer and to limit the data returned from the `ORDERS_TAB` table to that customer's data only.

See Also: For more information on using fine-grained access control, see "Introduction to Application Context Accessed Globally" on page 12-28, as well as *Oracle9i Supplied PL/SQL Packages and Types Reference*.

How to Establish Policy Groups

A policy group is a set of security policies which belong to an application. You can designate an application context (known as a driving context) to indicate the policy group in effect. Then, when the tables or views are accessed, the server looks up the driving context (which are also known as policy context) to determine the policy group in effect. It enforces all the associated policies which belong to that policy group.

This section contains the following topics:

- The Default Policy Group: SYS_DEFAULT
- New Policy Groups
- Using Oracle Policy Manager to Establish Policy Groups
- How to Implement Policy Groups
- Validation of the Application Used to Connect

The Default Policy Group: SYS_DEFAULT

In the Oracle Policy Manager tree structure, the Fine-Grained Access Control Policies folder contains the Policy Groups folder. The Policy Groups folder contains an icon for each policy group, as well as an icon for the SYS_DEFAULT policy group.

By default, all policies belong to the SYS_DEFAULT policy group by default. Policies defined in this group for a particular table or view will always be executed along with the policy group specified by the driving context. The SYS_DEFAULT policy group may or may not contain policies. If you attempt to drop the SYS_DEFAULT policy group, an error will be raised.

If, to the SYS_DEFAULT policy group, you add policies associated with two or more objects, then each such object will have a separate SYS_DEFAULT policy group associated with it. For example, the EMP table in the SCOTT schema has one SYS_DEFAULT policy group, and the DEPT table in the SCOTT schema has a different SYS_DEFAULT policy group associated with it. These are displayed in the tree structure as follows:

```
SYS_DEFAULT
  - policy1 (SCOTT/EMP)
  - policy3 (SCOTT/EMP)
SYS_DEFAULT
  - policy2 (SCOTT/DEPT)
```

Note: Policy groups with identical names are supported. When you select a particular policy group, its associated schema and object name are displayed in the property sheet on the right-hand side of the screen

New Policy Groups

When adding the policy to a table or view, you can use the DBMS_RLS.ADD_GROUPED_POLICY interface to specify the group to which the policy belongs. To specify which policies will be effective, you add a driving context using the DBMS_RLS.ADD_POLICY_CONTEXT interface. If the driving context returns an unknown policy group, an error is returned.

If the driving context is not defined, then all policies are executed. Likewise, if the driving context is NULL, then policies from all policy groups are enforced. In this way, an application accessing the data cannot bypass the security setup module (which sets up application context) to avoid any applicable policies.

You can apply multiple driving contexts to the same table or view, and each of them will be processed individually. In this way you can configure multiple active sets of policies to be enforced.

Consider, for example, a hosting company that hosts Benefits and Financial applications, which share some database objects. Both applications are striped for hosting using a `SUBSCRIBER` policy in the `SYS_DEFAULT` policy group. Data access is partioned first by subscriber ID, then by whether the user is accessing the Benefits or Financial applications (determined by a driving context). Suppose that Company A, which uses the hosting services, wants to apply a custom policy which relates only to its own data access. You could add an additional driving context (such as COMPANY A SPECIAL) to ensure that the additional, special policy group is applied for Company A's data access only. You would not apply this under the `SUBSCRIBER` policy, since the policy relates only to Company A, and it is cleaner to segregate the basic hosting policy from other policies.

Using Oracle Policy Manager to Establish Policy Groups

Alternatively, with the Oracle Policy Manager graphical user interface, accessed from Oracle Enterprise Manager, you can create a policy group by using the `DBMS_RLS.CREATE_POLICY_GROUP` command line procedure.

Note: You may use Oracle Policy Manager to create a policy context.

How to Implement Policy Groups

To create policy groups, the administrator must do two things:

- Set up a driving context to identify the effective policy group.
- Add policies to policy groups, as required.

The following example shows how to perform these tasks.

Step 1: Set Up a Driving Context

Begin by creating a namespace for the driving context. For example:

```
CREATE CONTEXT appsctx USING apps.apps_security_init;
```

Create the package that administers the driving context. For example:

```
CREATE OR REPLACE PACKAGE BODY apps.apps_security_init
PROCEDURE setctx ( policy_group varchar2 )
```

```
BEGIN

REM  Do some checking to determine the current application.
REM  You can check the proxy if using the proxy authentication feature.
REM  Then set the context to indicate the current application.
.
.
.
DBMS_SESSION.SET_CONTEXT('APPSCTX','ACTIVE_APPS', policy_group);
END;
END;
```

Define the driving context for the table APPS.BENEFIT.

```
DBMS_RLS.ADD_POLICY_CONTEXT('apps','benefit','APPSCTX','ACTIVE_APPS')
```

Step 2: Add a Policy to the Default Policy Group.

Create a security function to return a predicate to divide the data by company.

```
CREATE OR REPLACE FUNCTION by_company (schema varchar2, table varchar2)
RETURN VARCHAR2;
BEGIN
  RETURN 'COMPANY = SYS_CONTEXT(''ID'',''MY_COMPANY'')';
END;
```

Since policies in SYS_DEFAULT are always executed (except for SYS, or users with the EXEMPT ACCESS POLICY system privilege), this security policy (named SECURITY_BY_COMPANY), will always be enforced regardless of the application running. This achieves the universal security requirement on the table: namely, that each company should see its own data, regardless of the application running. The function APPS.APPS_SECURITY_INIT.BY_COMPANY returns the predicate to make sure that you can only see your company's data.

```
DBMS_RLS.ADD_GROUPED_POLICY('apps','benefit','SYS_DEFAULT',
'security_by_company',
'apps','by_company');
```

Step 3: Add a Policy to the HR Policy Group

First, create the HR group:

```
CREATE OR REPLACE FUNCTION hr.security_policy
RETURN VARCHAR2;
AS
BEGIN
```

```
 RETURN 'SYS_CONTEXT(''ID'',''TITLE'') = ''MANAGER'' ';
END;
DBMS_RLS.CREATE_POLICY_GROUP('apps','benefit','HR');
```

The following adds a policy named `HR_SECURITY` to the `HR` policy group. The function `HR.SECURITY_POLICY` returns the predicate to enforce HR's security on the table `APPS.BENEFIT`:

```
DBMS_RLS.ADD_GROUPED_POLICYS('apps','benefit','HR',
'hr_security','hr','security_policy');
```

Step 4: Add a Policy to the FINANCE Policy Group

Create the `FINANCE` policy:

```
CREATE OR REPLACE FUNCTION finance.security_policy
RETURN VARCHAR2;
AS
BEGIN
 RETURN 'SYS_CONTEXT(''ID'',''DEPT'') = ''FINANCE'' ';
END;
```

Create a policy group named `FINANCE`:

```
DBMS_RLS.CREATE_POLICY_GROUP('apps','benefit','FINANCE');
```

Add the `FINANCE` policy to the `FINANCE` group:

```
DBMS_RLS.ADD_GROUPED_POLICY('apps','benefit','FINANCE',
'finance_security','finance', 'security_policy');
```

As a result, when the database is accessed, the application initializes the driving context after authentication. For example, with the `HR` application:

```
execute apps.security_init.setctx('HR');
```

Validation of the Application Used to Connect

In this regard, one factor is extremely important: The package implementing the driving context must correctly validate the application which is being used. Although the database always ensures that the package implementing the driving context sets context attributes (by checking the call stack), this fact cannot protect against poor or inadequate validation within the package.

For example, in applications where database users or enterprise users are known to the database, the user needs `EXECUTE` privilege on the package which sets the driving context. Consider a user who knows that:

- The company’s BENEFITS application allows more liberal access than its HR application, and
- The setctx procedure (which sets the correct policy group within the driving context) does not perform any validation to determine which application is actually connecting. That is, the procedure does not check the IP address of the incoming connection (for a three-tier system), or, even better, the proxy_user attribute of the user session.

In this situation, the user could pass to the driving context package an argument (BENEFITS) which would set the context to the more liberal BENEFITS policy group—even though this user will, in fact, access the HR application. In this way the user can bypass the intended, more restrictive security policy simply because the package does inadequate validation.

If, by contrast, you implement proxy authentication with VPD, you can determine the identity of the middle tier (and thus, the application) which is actually connecting to the database on a user's behalf. In this way, the correct, per-application policy will be applied to mediate data access. For example, a developer using the proxy authentication feature could determine that the application (that is, the middle tier) connecting to the database is HRAPPSERVER. The package which implements the driving context can thus verify that the proxy_user in the user session is HRAPPSERVER before setting the driving context to use the HR policy group.

In this case, when the following query is executed

```
SELECT * FROM APPS.BENEFIT;
```

Oracle picks up policies from the default policy group (SYS_DEFAULT) and active namespace HR. The query is internally rewritten as follows:

```
SELECT * FROM APPS.BENEFIT WHERE COMPANY = SYS_CONTEXT(’ID’,’MY_COMPANY’) and
SYS_CONTEXT(’ID’,’TITLE’) = ’MANAGER’;
```

How to Add a Policy to a Table or View

The DBMS_RLS package enables you to administer security policies. These procedures allow you to specify the table or view to which you are adding a policy, the name of the policy, the name of the policy group, the function which implements the policy, the type of statement to which the policy applies (that is, SELECT, INSERT, UPDATE, or DELETE), and additional information. The package includes the following procedures:

Table 12–2 DBMS_RLS Procedures

Procedure	**Purpose**
`DBMS_RLS.ADD_POLICY`	Use this procedure to add a policy to a table or view.
`DBMS_RLS.DROP_POLICY`	Use this procedure to drop a policy from a table or view.
`DBMS_RLS.REFRESH_POLICY`	Use this procedure to force a reparse of open cursors associated with a policy, so that a new policy or change to a policy can take effect immediately.
`DBMS_RLS.ENABLE_POLICY`	Use this procedure to enable (or disable) a policy you previously added to a table or view.
`DBMS_RLS.CREATE_POLICY_GROUP`	Use this procedure to create a policy group.
`DBMS_RLS.ADD_GROUPED_POLICY`	Use this procedure to add a policy to the specified policy group.
`DBMS_RLS.ADD_POLICY_CONTEXT`	Use this procedure to add the context for the active application.
`DBMS_RLS.DELETE_POLICY_GROUP`	Use this procedure to drop a policy group.
`DBMS_RLS.DROP_GROUPED_POLICY`	Use this procedure to drop a policy which is a member of the specified group.
`DBMS_RLS.DROP_POLICY_CONTEXT`	Use this procedure to drop the context for the application.
`DBMS_RLS.ENABLE_GROUPED_POLICY`	Use this procedure to enable a policy within a group.
`DBMS_RLS.REFRESH_GROUPED_POLICY`	Use this procedure to reparse the SQL statements associated with a refreshed policy.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference*

Alternatively, you can use Oracle Policy Manager to administer security policies.

How to Check for Policies Applied to Statement

V$VPD_POLICY allows one to perform a dynamic view in order to check what policies are being applied to a SQL statement. When debugging, in your attempt to find which policy corresponds to a particular SQL statement, you should use the following table.

Table 12–3 V$VPD_POLICY

Column Name	Type
ADDRESS	RAW(4)
PARADDR	RAW(4)
SQL_HASH	NUMBER
CHILD_NUMBER	NUMBER
OBJECT_OWNER	VARCHAR2(30)
OBJECT_NAME	VARCHAR2(30)
POLICY_GROUP	VARCHAR2(30)
POLICY	VARCHAR2(30)
POLICY_FUNCTION_OWNER	VARCHAR2(30)
PREDICATE	VARCHAR2(30)
DBMS_RLS.REFRESH_GROUPED_POLICY	VARCHAR2(4096)

EXEMPT ACCESS POLICY System Privilege

The system privilege EXEMPT ACCESS POLICY allows a user to be exempted from all fine-grained access control policies on any DML operation (SELECT, INSERT, UPDATE, and DELETE). This provides ease of use for such administrative activities as installation, and import and export of the database through a non-SYS schema.

Also, regardless of the utility or application that is being used, if a user is granted the EXEMPT ACCESS POLICY privilege, then the user is exempt from VPD and Oracle Label Security policy enforcement. That is, the user will not have any VPD or Oracle Label Security policies applied to their data access.

Since EXEMPT ACCESS POLICY negates the effect of fine-grained access control, this privilege should only be granted to users who have legitimate reasons for bypassing fine-grained security enforcement. This privilege should not be granted WITH ADMIN OPTION, so that users cannot pass on the EXEMPT ACCESS POLICY privilege to other users, and thus propagate the ability to bypass fine-grained access control.

Fine-Grained Auditing

This section describes fine-grained auditing in the context of Oracle9*i* auditing capabilities. It contains the following sections:

- Introduction to Standard Auditing and Fine-Grained auditing
- Standard Oracle9i Auditing Techniques
- Fine-Grained Auditing Techniques

Introduction to Standard Auditing and Fine-Grained auditing

Standard Oracle9*i* auditing monitors privileges and objects, and provides triggers to monitor DML operations such as INSERT, UPDATE, and DELETE. By contrast, monitoring SELECT statements is facilitated by fine-grained auditing, which allows the monitoring of data access based on content. In this way, you can specify auditing conditions, and obtain more specific information about the environment and query result. This additional information helps you reconstruct audited events, and determine whether access rights have been violated.

For example, a drug enforcement agency needs to track in detail access to its informants database. Likewise, a central tax authority needs to track access to tax returns in order to guard against employee snooping. Such agencies need enough detail to determine what data was accessed, not simply that the SELECT privilege was used by SCOTT on the INFORMANTS table.

Standard Oracle9i Auditing Techniques

Oracle provides over 170 configurable auditing options for accountability of users and servers. The Oracle9*i* audit facility allows you to audit database activity by statement, by use of system privilege, by object, or by user. For example, you can audit activity as general as all user connections to the database, and as specific as a particular user creating a table. You can audit only successful operations, or only unsuccessful operations. Auditing unsuccessful SELECT statements may find users attempting to access data that they are not privileged to see.

Although auditing is highly configurable, standard audit options do not include a lot of detail about the audited events. Typically, an audit record identifies the user, the object accessed, the privilege used, whether the access was successful or unsuccessful, and a timestamp.

You can use triggers to record customized information that is not automatically included in audit records. In this way, you can further design your own audit auditing conditions and audit record contents. For example, you could define a trigger on the EMP table to generate an audit record whenever an employee's salary is increased by more than 10 percent. This can include selected information, such as before and after values of SALARY:

```
CREATE TRIGGER audit_emp_salaries
AFTER INSERT OR DELETE OR UPDATE ON employee_salaries
for each row
begin
if (:new.salary> :old.salary * 1.10)
      then
      insert into emp_salary_audit values (
      :employee_no,
      :old.salary,
      :new.salary,
      user,
      sysdate);
      endif;
end;
```

Furthermore, you can use event triggers to enable auditing options for specific users on login, and disable them upon logoff.

In some cases, businesses may actually need to capture the statement executed as well as the result set from a query. Fine-grained auditing provides an extensible auditing mechanism that supports definition of key conditions for granular audit, as well as an event handler to actively alert administrators to misuse of data access rights.

Oracle9*i* also gives you the option of sending audit records to the database audit trail or your operating system's audit trail, when the operating system is capable of receiving them. This option, coupled with the broad selection of audit options and the ability to customize auditing with triggers or stored procedures, gives you the flexibility of implementing an auditing scheme that suits your specific business needs.

Fine-Grained Auditing Techniques

A more granular level of auditing can be achieved with a fine-grained auditing mechanism. This employs simple, user-defined SQL predicates on table objects as conditions for selective auditing. During fetching, whenever policy conditions are met for returning a row from a query block, the query is audited.

Fine-grained auditing allows organizations to define audit policies, which specify the data access conditions that trigger the audit event, and use a flexible event handler to notify administrators that the triggering event has occurred. For example, an organization may allow HR clerks to access employee salary information, but trigger an audit event when salaries are greater than $500K are accessed. The audit policy (where SALARY > 500000) is applied to the EMPLOYEES table through an audit policy interface (a PL/SQL package).

For additional flexibility in implementation, organizations can employ a user-defined function to determine the policy condition, and identify an audit column to further refine the audit policy. For example, the function could allow unaudited access to any salary as long as the user is accessing data within the intranet, but audit access to executive-level salaries when they are accessed from the internet. A relevant column helps reduce the instances of false or unnecessary audit records, because the audit need only be triggered when a particular column is referenced in the query. For example, an organization may only wish to audit executive salary access when an employee name is accessed, because accessing salary information alone is not meaningful unless an HR clerk also selects the corresponding employee name.

You can use the PLSQL package DBMS_FGA to administer these fine-grained audit policies. If any rows returned from a query block match the audit condition, these rows are identified as *interested* rows. An audit event entry, including username, SQL text, policy name, session id, timestamp, and other attributes, is inserted into the audit trail. You can optionally define an *audit event handler* to process the event. For example, the event handler could send an alert page to the administrator.

The following example shows how you can audit SELECT statements on table *hr.emp* to monitor any query that accesses the *salary* column of the employee records which belong to *sales* department:

```
DBMS_FGA.ADD_POLICY(
object_schema => 'hr',
object_name   => 'emp',
policy_name   => 'chk_hr_emp',
audit_condition => 'dept = ''SALES'' ',
audit_column => 'salary');
```

Then, either of the following SQL statements will cause the database to log an audit event record.

```
SELECT count(*) FROM hr.emp WHERE dept = ’SALES’ and salary > 10000000;
```

or

```
SELECT salary FROM hr.emp WHERE dept = ’SALES’;
```

With all the relevant information available, and a trigger-like mechanism to use, the administrator can define what to record and how to process the audit event.

Consider what happens when the following commands are issued. After the fetch of the first interested row, the event is recorded, and the audit function `SEC.LOG_ID` is executed. The audit event record generated is stored in `DBA_FGA_AUDIT_TRAIL`, which has reserved columns for recording SQL text, policy name, and other information.

```
/* create audit event handler */
CREATE PROCEDURE sec.log_id (schema varchar2, table varchar2, policy varchar2)
AS
BEGIN
UTIL_ALERT_PAGER(schema, table, policy);      -- send an alert note to my pager
END;

/* add the policy */
DBMS_FGA.ADD_POLICY(
object_schema => ’hr’,
object_name   => ’emp’,
policy_name   => ’chk_hr_emp’,
audit_condition => ’dept = ’’SALES’’ ’,
audit_column => ’salary’,
handler_schema => ’sec’,
handler_module => ’log_id’,
enable              =>  TRUE);
```

Note: Fine-grained auditing is supported only with cost-based optimization. For query, using rule-based optimization, audit will check before applying row filtering, which could result in an unnecessary audit event trigger.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference*

Enforcing Application Security

This section contains information about enforcing application security. This section consists of the following topics:

- Use of Ad Hoc Tools a Potential Security Problem
- Restricting Database Roles from SQL*Plus Users

Use of Ad Hoc Tools a Potential Security Problem

Prebuilt database applications explicitly control the potential actions of a user, including the enabling and disabling of the user's roles while using the application. By contrast, ad hoc query tools, such as SQL*Plus, allow a user to submit any SQL statement (which may or may not succeed), including the enabling and disabling of any granted role.

An application user can potentially exercise the privileges attached to that application to issue destructive SQL statements against database tables by using an ad hoc tool.

For example, consider the following scenario:

- The Vacation application has a corresponding `VACATION` role.
- The `VACATION` role includes the privileges to issue `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements against the `EMP_TAB` table.
- The Vacation application controls the use of privileges obtained through the `VACATION` role.

Now, consider a user who has been granted the `VACATION` role. Suppose that, instead of using the Vacation application, the user executes SQL*Plus. At this point, the user is restricted only by the privileges granted to him explicitly or through roles, including the `VACATION` role. Because SQL*Plus is an ad hoc query tool, the user is not restricted to a set of predefined actions, as with designed database applications. The user can query or modify data in the `EMP_TAB` table as he or she chooses.

Restricting Database Roles from SQL*Plus Users

This section presents features that you may use in order to restrict database roles from SQL*Plus users and thus, prevent serious security problems. These features include the following:

- Limiting Roles Through PRODUCT_USER_PROFILE
- Using Stored Procedures to Encapsulate Business Logic
- Using Virtual Private Database for Highest Security
- Virtual Private Database and Oracle Label Security

Limiting Roles Through PRODUCT_USER_PROFILE

Oracle9*i* offers some capability to limit what roles a user accesses through an application, through the `PRODUCT_USER_PROFILE` table.

DBAs can use `PRODUCT_USER_PROFILE` to disable certain SQL and SQL*Plus commands in the SQL*Plus environment on a per-user basis. SQL*Plus, not Oracle, enforces this security. DBAs can even restrict access to the `GRANT`, `REVOKE`, and `SET ROLE` commands in order to control users' ability to change their database privileges.

The `PRODUCT_USER_PROFILE` table enables you to list roles which you do not want users to activate with an application. You can also explicitly disable use of various commands, such as `SET ROLE`. For example, you could create an entry in the `PRODUCT_USER_PROFILE` table to:

- Disallow use of the `CLERK` and `MANAGER` roles with SQL*Plus
- Disallow use of `SET ROLE` with SQL*Plus

Suppose user Jane connects to the database using SQL*Plus. Jane has the `CLERK`, `MANAGER`, and `ANALYST` roles. As a result of the above entry in `PRODUCT_USER_PROFILE`, Jane is only able to exercise her `ANALYST` role with SQL*Plus. Also, when Jane attempts to issue a `SET ROLE` statement, she is explicitly prevented from doing so because of the entry in the `PRODUCT_USER_PROFILE` table prohibiting use of `SET ROLE`.

Use of the `PRODUCT_USER_PROFILE` table does not completely guarantee security, for multiple reasons. In the above example, while `SET ROLE` is disallowed with SQL*Plus, if Jane had other privileges granted to her directly, she could exercise these using SQL*Plus.

.

See Also: *SQL*Plus User's Guide and Reference* for more information about the `PRODUCT_USER_PROFILE` table

Using Stored Procedures to Encapsulate Business Logic

Stored procedures encapsulate use of privileges with business logic so that privileges are only exercised in the context of a well-formed business transaction. For example, an application developer might create a procedure to update employee name and address in `EMPLOYEES` table, which enforces that the data can only be updated in normal business hours. Also, rather than grant an human resources clerk the `UPDATE` privilege on the `EMPLOYEES` table, a developer (or application administrator) may grant the privilege on the procedure only. Then, the human resources clerk can exercise the privilege only in the context of the procedures, and cannot update the `EMPLOYEES` table directly.

Using Virtual Private Database for Highest Security

Oracle9*i* enables you to enforce security, to a fine level of granularity, directly on tables or views by implementing virtual private database (VPD). Because security policies are attached directly to tables or views and automatically applied whenever a user accesses data, there is no way to bypass security.

Strong security policies, centrally managed and applied directly to data, can enforce security no matter how a user gets to the data: whether through an application, through a query, or by using a report-writing tool.

When a user directly or indirectly accesses a table or view associated with a VPD security policy, the server dynamically modifies the user's SQL statement. The modification is based on a `WHERE` condition (known as a predicate) returned by a function which implements the security policy. The statement is modified dynamically, transparently to the user, using any condition which can be expressed in, or returned by a function.

Functions which return predicates can also include callouts to other functions. Within your PL/SQL package, you can embed a C or Java callout that can either access operating system information, or return `WHERE` clauses from an operating system file or central policy store. A policy function can return different predicates for each user, for each group of users, or for each application.

Application context enables you to securely access the attributes on which you base your security policies. For example, users with the position attribute of `manager` would have a different security policy than users with the position attribute of `employee`.

Consider an HR clerk who is only allowed to see employee records in the Aircraft Division. When the user initiates the query

```
SELECT * FROM emp;
```

the function implementing the security policy returns the predicate `division = 'AIRCRAFT'`, and the database transparently rewrites the query. The query actually executed becomes:

```
SELECT * FROM emp WHERE division = 'AIRCRAFT';
```

The security policy is applied within the database itself, rather than within an application. This means that use of a different application will not bypass the security policy. Security can thus be built once, in the database, instead of being reimplemented in multiple applications. Virtual private database therefore provides far stronger security than application-based security, at a lower cost of ownership.

It may be desirable to enforce different security policies depending on which application is accessing data. Consider a situation in which two applications, Order Entry and Inventory, both access the `ORDERS` table. You may want to have the Inventory application apply to the table a policy which limits access based on type of product. At the same time, you may want to have the Order Entry application apply to the same table a policy which limits access based on customer number.

In this case, you must partition the use of fine-grained access by application. Otherwise, both policies would be automatically `AND`ed together—which is not the desired result. You can specify one or more policy groups, and a driving application context that determines which policy group is in effect for a given transaction. You can also designate default policies which always apply to data access. In a hosted application, for example, data access should always be limited by subscriber ID.

> **See Also:** "Ways to Use Application Context with Fine-Grained Access Control" on page 12-8

Virtual Private Database and Oracle Label Security

Virtual private database and Oracle Label Security are not enforced during `DIRECT` path export. Also, Virtual private database policies and Oracle Label Security policies cannot be applied to objects in schema `SYS`. As a consequence, the `SYS` user and users making a DBA-privileged connection to the database (for example, CONNECT/AS SYSDBA) do not have VPD or Oracle Label Security policies applied to their actions. Database administrators need to be able to administer the database. It would not suffice to export part of a table due to a VPD policy being applied.

Database users who are granted the Oracle9*i* EXEMPT ACCESS POLICY privilege, directly or through a database role, are exempt from Virtual Private Database and Oracle Label Security enforcement. The users are exempt from Virtual Private Database and Oracle Label Security enforcement regardless of the export mode, application, or utility used to extract data from the database. EXEMPT ACCESS POLICY privilege is a powerful privilege and should be carefully managed.

Note: The EXEMPT ACCESS POLICY privilege does not affect the enforcement of object privileges such as SELECT, INSERT, UPDATE, and DELETE. These privileges are enforced even if a user has been granted the EXEMPT ACCESS POLICY privilege.

13 Proxy Authentication

This chapter provides information about proxy authentication in a multi-tier system. Topics in this chapter are presented in the following sections:

- Advantages of Proxy Authentication
- Security Challenges of Three-tier Computing
- Oracle9i Proxy Authentication Solutions

Advantages of Proxy Authentication

In multi-tier environments, proxy authentication allows you to control the security of middle-tier applications by preserving client identities and privileges through all tiers, and auditing actions taken on behalf of clients. For example, this feature allows the identity of a user using a web application (also known as a "proxy") to be passed through the application to the database server.

Three-tier systems provide many benefits to organizations.

- Application servers and web servers enable users to access data stored in legacy applications.
- Users like using a familiar, easy-to-use browser interface.
- Organizations can separate application logic from data storage, partitioning the former in application servers and the latter in databases.
- Organizations can also lower their cost of computing by replacing many "fat clients" with a number of "thin clients" and an application server.

In addition, Oracle proxy authentication delivers the following security benefits:

- A limited trust model, by controlling the users on whose behalf middle tiers can connect, and the roles the middle tiers can assume for the user
- Scalability, by supporting lightweight user sessions through OCI and thick JDBC, and eliminating the overhead of re-authenticating clients
- Accountability, by preserving the identity of the real user through to the database, and enabling auditing of actions taken on behalf of the real user
- Flexibility, by supporting environments in which users are known to the database, and in which users are merely "application users" of which the database has no awareness

Note: Oracle9*i* supports the above functionality in three tiers only; it does not support functionality across multiple middle tiers.

Security Challenges of Three-tier Computing

While three-tier computing provides many benefits, it raises a number of new security issues. These issues are described in the following sections:

- Who Is the Real User?
- Does the Middle Tier Have Too Much Privilege?
- How to Audit? Whom to Audit?
- Can the User Be Re-Authenticated to the Database?

Who Is the Real User?

Most organizations want to know the identity of the actual user who is accessing the database, for reasons of access control or auditing. User accountability is diminished if the identity of the users cannot be traced through all tiers of the application.

Furthermore, if only the application server knows who the user is, then all per-user security enforcement must be done by the application itself. Application-based security is very expensive. If each application that accesses the data must enforce security, then security must be re-implemented in each and every application. It is often preferable to build security on the data itself, with per-user accountability enforced within the database.

Does the Middle Tier Have Too Much Privilege?

Some organizations are willing to accept three-tier systems within the enterprise, in which "all-privileged" middle tiers, such as transaction processing (TP) monitors, can perform all actions for all users. In this architecture, the middle tier connects to the database as the same user for all application users. It therefore needs to have *all* privileges that application users need to do their jobs.

This computing model can be undesirable in the Internet, where the middle tier resides outside, on, or just inside a firewall. More desirable, in this context, is a *limited trust model*, in which the identity of the real client is known to the data server, and the application server (or other middle tier) has a restricted privilege set.

Also useful is the ability to limit the users on whose behalf a middle tier can connect, and the roles the middle tier can assume for the user. For example, many organizations would prefer that users have different privileges depending on where they are connecting from. A user connecting to a web server or application server on the firewall might only be able to use very minimal privileges to access data, whereas a user connecting to a web server or application server within the enterprise might be able to exercise all privileges she is otherwise entitled to have.

How to Audit? Whom to Audit?

Accountability through auditing is a basic principle of information security. Most organizations want to know on whose behalf a transaction was accomplished, not just that a particular application server performed a transaction. A system must therefore be able to differentiate between a user performing a transaction, and an application server performing a transaction on behalf of a user.

Auditing in three-tier systems should be tied to the issue of knowing the real user: if you cannot preserve the user's identity through the middle tier of a three-tier application, then you cannot audit actions on behalf of the user.

Can the User Be Re-Authenticated to the Database?

In client/server systems, authentication tends to be straightforward: the client authenticates to the server. In three-tier systems authentication is more difficult, because there are several potential types of authentication.

- Client to Middle Tier Authentication
- Middle Tier to Database Authentication
- Client Re-Authentication Through Middle Tier to Database

Client to Middle Tier Authentication

Client authentication to the middle tier is clearly required if a system is to conform with basic security principles. The middle tier is typically the first gateway to useful information that the user can access. Users *must*, therefore, authenticate to the middle tier. Note that such authentication can be mutual; that is, the middle tier authenticates to the client just as the client authenticates to the middle tier.

Middle Tier to Database Authentication

Since the middle tier must typically initiate a connection to a database to retrieve data (whether on its own behalf or on behalf of the user), this connection clearly must be authenticated. In fact, the Oracle9*i* database does not allow unauthenticated connections. Again, middle tier to database authentication can also be mutual if using a protocol that supports this, such as SSL.

Client Re-Authentication Through Middle Tier to Database

Client re-authentication from the middle tier to the database is problematic in three-tier systems. The username might not be the same on the middle tier and the database. In this case, users may need to reenter a username and password, which the middle tier uses to connect on their behalf. Or, more commonly, the middle tier may need to map the username provided, to a database username. This mapping is often done in an LDAP-compliant directory service, such as Oracle Internet Directory.

For the client to re-authenticate himself to the database, the middle tier either needs to ask the user for a password (which it then must be trusted to pass to the database), or the middle tier must retrieve a password for the user and use that to authenticate the user. Both approaches involve security risks, because the middle tier is trusted to handle the user’s password properly, and not to use it maliciously.

One of the only cases for which re-authentication does not involve trusting the middle tier occurs when a middle tier downloads an applet to a client, and the client connects *directly* to the database via the applet. In this case, the application server is literally just that: it serves the application (applet) to the user, and has no part in further authentication of the user.

Re-authenticating the client to the back-end database is not always beneficial. First, two sets of authentication handshakes per user involves considerable network overhead. Second, you must trust the middle tier to have authenticated the user. (You clearly must trust the middle tier if it retrieves or otherwise is privy to the user’s password.) It is therefore not unreasonable for the database to simply accept that the middle tier has performed proper authentication. In other words, the database accepts the identity of the real client without requiring the real client to authenticate herself.

For some authentication protocols, client re-authentication is just not possible. For example, many browsers and application servers support the Secure Sockets Layer (SSL) protocol. Both the Oracle9*i* database (through Oracle Advanced Security) and Oracle9*i* Internet Application Server support the use of SSL for client authentication. However, SSL is a point-to-point protocol, not an end-to-end protocol. It cannot be used to re-authenticate a browser client (through the middle tier) to the database.

The reason for this is that a user cannot *securely* give up his private key to the middle tier in order for the re-authentication of the client to occur. Once the user’s private key is compromised, the user’s very identity is compromised. In addition, there is no way to *tunnel* through a middle tier so that the authentication of the browser client to the database can occur directly.

In short, organizations deploying three-tier systems require flexibility as regards re-authentication of the client. In some cases, they cannot re-authenticate the client; in other cases, they may choose whether or not to re-authenticate the client.

Oracle9*i* Proxy Authentication Solutions

The following sections explain how Oracle9*i* provides solutions to specific authentication issues.

- Passing Through the Identity of the Real User
- Limiting the Privilege of the Middle Tier
- Re-authenticating the Real User
- Auditing Actions Taken on Behalf of the Real User
- Support for Application User Models

Passing Through the Identity of the Real User

Many organizations want to know who the user is through all tiers of an application, without sacrificing the benefits of a middle tier. Oracle9*i* supports multiple ways of preserving user identity through the middle tier of an application. For enterprise users, users managed in Oracle Internet Directory who access a shared schema in the database, or database users, Oracle9*i* provides proxy authentication in OCI or thick JDBC. For application users, users known to an application but unknown to the database, Oracle9*i* supports application user proxy authentication in OCI, thick JDBC, and thin JDBC.

For enterprise users or database user, OCI or thick JDBC enables a middle tier to set up, within a single database connection, a number of *lightweight* user sessions, each of which uniquely identifies a connected user. These lightweight sessions reduce the network overhead of creating separate network connections from the middle tier to the database. The application can switch between these sessions as required to process transactions on behalf of users.

The full authentication sequence from the client to the middle tier to the database occurs as follows:

1. The client authenticates to the middle tier, using whatever form of authentication the middle tier will accept. For example, the client could authenticate to the middle tier using a username/password, or an X.509 certificate by means of SSL.

2. The middle tier creating the lightweight sessions must be a database user, rather than an enterprise user. The middle tier authenticates itself to Oracle9*i*, using whatever form of authentication Oracle9*i* will accept. This could be a password, or an authentication mechanism supported by Oracle Advanced Security, such as a Kerberos ticket or an X.509 certificate (SSL).
3. The middle tier then creates one or more sessions for users using the Oracle Call Interface or thick JDBC.
   - If the user is a database user, the lightweight session must, as a minimum, include the database username. If the database requires it, the session may also include a password (which the database verifies against the password store in the database). The session may also include a list of database roles for the user.
   - If the user is an enterprise user, the lightweight session may provide different information depending on how the user is authenticated. If the user authenticated to the middle tier via SSL, the middle tier can provide the DN from the user's X.509 certificate, or the certificate itself in the session. The database uses the DN to look up the user in Oracle Internet Directory. If the user is a password-authenticated enterprise user, then the middle tier must provide, as a minimum, a globally unique name for the user. The database uses this name to look up the user in Oracle Internet Directory. If the session also provides a password for the user, the database will verify the password against Oracle Internet Directory. The user's roles are automatically retrieved from Oracle Internet Directory after the session is established.
   - The middle tier may optionally provide a list of database roles for the client, in cases where the user is a database user rather than an enterprise user. If the user is an enterprise user, then the users' roles are automatically retrieved from Oracle Internet Directory after the session is established.
4. If the user is a database user, the database verifies that the middle tier is privileged to create sessions on behalf of the user, using the roles provided.

   The `OCISessionBegin` call will fail if the application server is not allowed to proxy on behalf of the client by the administrator, or if the application server is not allowed to activate the specified roles.

Limiting the Privilege of the Middle Tier

"Least privilege" is the principle that users should have the fewest privileges necessary to perform their duties, and no more. As applied to middle tier applications, this means that the middle tier should not have more privileges than it needs. Oracle9*i* enables you to limit the middle tier such that it can connect only on behalf of certain database users, using only specific database roles. You cannot limit the ability of the middle tier to connect on behalf of enterprise users or limit the user of enterprise roles in OCI or thick JDBC lightweight connections.

For example, suppose that user Sarah wants to connect to the database through a middle tier, `appsrv` (which is also a database user). Sarah has multiple roles, but it is desirable to restrict the middle tier to exercise only the `clerk` role on her behalf.

A DBA could effectively grant permission for `appsrv` to initiate connections on behalf of Sarah using her `clerk` role only, using the following syntax:

```
ALTER USER Sarah GRANT CONNECT THROUGH appsrv WITH ROLE clerk;
```

By default, the middle tier cannot create connections for any client. The permission must be granted on a per-user basis.

To allow `appsrv` to use all of the roles granted to the client Sarah, the following statement would be used:

```
ALTER USER sarah GRANT CONNECT THROUGH appsrv;
```

Each time a middle tier initiates a lightweight (OCI) or thick JDBC session for another database user, the database verifies that the middle tier is privileged to connect for that user, using the role specified.

You can limit the privilege of the middle tier to connect on behalf of an enterprise user by granting to the middle-tier the privilege to connect as the underlying database user. For instance, if the enterprise user is mapped to the `APPUSER` schema, you must at least grant to the middle tier the ability to connect on behalf of `APPUSER`. Otherwise, attempts to create a session for the enterprise user will fail.

Re-authenticating the Real User

In the case of authentication with a database password, the password of the client is passed to the middle-tier server. The middle-tier server then passes the password as an attribute to the data server for verification. The main advantage to this is that the client machine does not have to have Oracle software actually installed on it.

As described above, it is not always beneficial to re-authenticate users to the database after they have been authenticated by the middle tier. However, if you wish to do this for an added measure of security, you can pass the database the user's password using the `OCI_ATTR_PASSWORD` attribute of the `OCIAttrSet` call.

.

See Also: For more information about security in three-tier architectures, see the *Oracle Call Interface Programmer's Guide.*

Using Proxy Authentication with Enterprise Users

If the middle tier is connecting to the database as client who is an enterprise user, either the distinguished name or the X.509 certificate containing the distinguished name is passed over instead of the database user name. If the user is a password-authenticated enterprise user, then the middle tier must provide, as a minimum, a globally unique name for the user. The database uses this name to look up the user in Oracle Internet Directory.

To pass over the distinguished name of the client, the application server would call `OCIAttrSet()` with the following pseudo-interface.

```
OCIAttrSet(OCISession *session_handle,
OCI_HTYPE_SESSION,
lxstp *distinguished_name,
(ub4) 0,
OCI_ATTR_DISTINGUISHED_NAME,
OCIError *error_handle);
```

To pass over the entire certificate, the middle tier would use the following pseudo-interfaces:

```
OCIAttrSet(OCISession *session_handle,
OCI_HTYPE_SESSION,
ub1 *certificate,
ub4 certificate_length,
OCI_ATTR_CERTIFICATE,
OCIError *error_handle);
```

Note: `OCI_ATTR_CERTIFICATE` is `DER` encoded.

If the type is not specified, then the server will use its default certificate type of X.509.

If using proxy authentication for password-authenticated enterprise users, use the same OCI attributes as for database users authenticated by password (e.g. `OCI_ATTR_USERNAME`). The database first checks the username against the database; if no user is found, then the database checks the username in the directory. This username must be globally unique.

Auditing Actions Taken on Behalf of the Real User

The proxy authentication features of Oracle9*i* enable you to audit actions that a middle tier performs on behalf of a user. For example, suppose an application server `hrappserver` creates multiple lightweight sessions for users Ajit and Jane. A DBA could enable auditing for `SELECT`s on the `bonus` table that `hrappserver` initiates for Jane as follows:

```
AUDIT SELECT TABLE BY hrappserver ON BEHALF OF Jane;
```

Alternatively, the DBA could enable auditing on behalf of multiple users (in this case, both Jane and Ajit) connecting through a middle tier as follows:

```
AUDIT SELECT TABLE BY hrappserver ON BEHALF OF ANY;
```

This auditing option only audits `SELECT` statements being initiated by `hrappserver` on behalf of other users. A DBA can enable separate auditing options to capture `SELECT`s against the `bonus` table from clients connecting directly to the database:

```
AUDIT SELECT TABLE.
```

For audit actions taken on behalf of the real user, you cannot audit `CONNECT ON BEHALF OF` *`DN`*, since the distinguished name is not known to the database. However, if the user accesses a shared schema (for example, `APPUSER`), then you can audit `CONNECT ON BEHALF OF APPUSER`.

Support for Application User Models

Many applications use session pooling to set up a number of sessions to be reused by multiple users. In this context, "application users" are users who are authenticated to the middle tier of an application, but who are not known to the database. Oracle9*i* supports application user proxy for these types of applications.

In this model, the middle tier passes a client identifier to the database upon the session establishment. (The client identifier could actually be anything that represents a client connecting to the middle tier; a cookie, for example, or an IP address.) The client identifier, representing the application user, is available in user session information and can also be accessed via an application context (via the `USERENV` naming context). In this way, applications can set up and reuse sessions, while still being able to keep track of the "application user" in the session.

Applications can reset the client identifier and thus reuse the session for a different user, enabling high performance. For OCI-based connections, alteration of the `CLIENT_IDENTIFIER` is piggybacked on the other OCI calls to further enhance performance. Application user proxy is available in thin JDBC, thick JDBC, and OCI, and provides the benefits of connection pooling without the overhead of separate user sessions (even lightweight ones).

Application user proxy can be used with global application context for additional flexibility and high performance in building applications. For example, suppose a web-based application that provides information to business partners has a notion of three types of users: gold partner, silver partner, or bronze partner representing the different levels of information available. It is not important to the application that users be known to the database, but it needs to limit data access based on *contexts* representing gold partner, silver partner, or bronze partner respectively. That is, instead of each user having his *own* session—with the individual application context set up—the application could set up global application contexts for gold partner, silver partner, or bronze partner and use the client identifier to point the session at the correct context, in order to retrieve the appropriate type of data. The application need only initialize the three global contexts, and use the client identifier to access the correct application context to limit data access. This provides performance improvements through session reuse, and through accessing global application contexts set up once, instead of having to initialize application contexts for each session individually.

See Also:

- "Initializing Application Context Globally" on page 12-35
- *Oracle9i JDBC Developer's Guide and Reference*

14 Data Encryption Using DBMS_ OBFUSCATION_TOOLKIT

This chapter provides a description of the data encryption package (`DBMS_OBFUSCATION_TOOLKIT`) that allows you to encrypt data in a database. Data encryption topics are presented in the following sections:

Securing Sensitive Information

The Internet poses new challenges in information security, especially for those organizations seeking to become e-businesses. Many of these security challenges can be addressed by the traditional arsenal of security mechanisms:

- Strong user authentication to identify users
- Granular access control to limit what users can see and do
- Auditing for accountability
- Network encryption to protect the confidentiality of sensitive data in transmission

Encryption is an important component of several of the above solutions. For example, Secure Sockets Layer (SSL), an Internet-standard network encryption and authentication protocol, uses encryption to strongly authenticate users by means of X.509 digital certificates. SSL also uses encryption to ensure data confidentiality, and cryptographic checksums to ensure data integrity. Many of these uses of encryption are relatively transparent to a user or application. For example, many browsers support SSL, and users generally do not need to do anything special to enable SSL encryption.

Oracle has provided network encryption between database clients and the Oracle database since Oracle7. Oracle Advanced Security, an option to Oracle9*i*, provides encryption and cryptographic integrity check for any protocol supported by Oracle9*i*, including Net8, Java Database Connectivity (JDBC) (both “thick” and “thin” JDBC), and the Internet Intra-Orb Protocol (IIOP). Oracle Advanced Security also supports SSL for Net8, “thick” JDBC and IIOP connections.

While encryption is not a security cure-all, it is an important tool used to address specific security threats, and will become increasingly important with the growth of e-business, especially in the area of encryption of stored data. For example, while credit card numbers are typically protected in transit to a web site using SSL, the credit card number is often stored in the clear (un-encrypted), either on the file system, where it is vulnerable to anyone who can break into the host and gain root access, or in databases. While databases can be made quite secure through proper configuration, they can also be vulnerable to host break-ins if the host is mis-configured. There have been several well-publicized break-ins, in which a hacker obtained a large list of credit card numbers by breaking into a database.

Encryption of stored data thus represents a new challenge for e-businesses, and can be an important tool in dealing with specific types of security threats.

Principles of Data Encryption

While there are many good reasons to encrypt data, there are many bad reasons to encrypt data. Encryption does not solve all security problems, and may even make some problems worse. The following section describes some of the misconceptions about encryption of stored data. It includes these topics:

- Principle 1: Encryption Does Not Solve Access Control Problems
- Principle 2: Encryption Does Not Protect Against a Malicious DBA
- Principle 3: Encrypting Everything Does Not Make Data Secure

Principle 1: Encryption Does Not Solve Access Control Problems

Most organizations need to limit access to data to those who have a "need to know." For example, a human resources system may limit employees to reviewing only their own employment records, while managers of employees may see the employment records of those employees working for them. Human resources specialists may also need to see employee records for multiple employees.

This type of security policy—limiting data access to those with a need to see it—is typically addressed by access control mechanisms. The Oracle database has provided strong, independently-evaluated access control mechanisms for many years. Recently, Oracle9*i* has added the ability to enforce access control to an extremely fine level of granularity, through its Virtual Private Database capability.

Because human resources records are considered sensitive information, it is tempting to think that this information should all be encrypted "for better security." However, encryption cannot enforce the type of granular access control described above, and may actually hinder data access. In the human resources example, an employee, his manager, and the HR clerk all need to access the employee's record. If employee data is encrypted, then each person also has to be able to access the data in un-encrypted form. Therefore, the employee, the manager and the HR clerk would have to share the same encryption key to decrypt the data. Encryption would therefore not provide any additional security in the sense of better access control, and the encryption might actually hinder the proper functioning of the application. An additional issue is that it is very difficult to securely transmit and share encryption keys among multiple users of a system.

A basic principle behind encrypting stored data is that it must not interfere with access control. For example, a user who has `SELECT` privilege on `EMP` should not be limited by the encryption mechanism from seeing all the data he is otherwise allowed to see. Similarly, there is little benefit to encrypting part of a table with one key and part of a table with another key if users need to see all encrypted data in the table; it merely adds to the overhead of decrypting the data before users can read it. Provided that access controls are implemented well, there is little additional security provided within the database itself from encryption. Any user who has privilege to access data within the database has no more nor any less privilege as a result of encryption. Therefore, encryption should never be used to solve access control problems.

Principle 2: Encryption Does Not Protect Against a Malicious DBA

Some organizations are concerned that a malicious user can gain elevated (DBA) privilege through guessing a password. These organizations would like to encrypt stored data to protect against this threat. However, the correct solution to this problem is to protect the DBA account, and to change default passwords for other privileged accounts. The easiest way to break into a database is through an unchanged password for a privileged account (for example, `SYS/CHANGE_ON_INSTALL`).

Note that there are many other destructive things a malicious user can do to a database once he gains DBA privilege (corrupting or deleting data, exporting user data to the file system and mailing the data back to himself so he can run a password cracker on it, etc.) Encryption will not protect against these threats.

Some organizations are concerned that database administrators (DBAs), because they typically have all privileges, are able to see all data in the database. These organizations feel that the DBAs should merely administer the database, but should not be able to see the data that the database contains. Some organizations are also concerned about the concentration of privilege in one person, and would prefer to partition the DBA function, or enforce two-person rules.

It is tempting to think that encrypting all data (or significant amounts of data) will solve the above problems, but there are better ways to accomplish these objectives. First of all, Oracle does support limited partitioning of DBA privilege. Oracle9*i* provides native support for SYSDBA and SYSOPER users. SYSDBA has all privileges, but SYSOPER has a limited privilege set (such as startup and shutdown of the database). Furthermore, an organization can create smaller roles encompassing a number of system privileges. A JR_DBA role might not include all system privileges, but only those appropriate to a more junior database administrator (such as CREATE TABLE, CREATE USER, and so on). Oracle does not audit the actions taken by SYS (or SYS-privileged users) but does audit startup and shutdown of the database in the operating system records.

Furthermore, the DBA function by its nature is a trusted position. Even organizations with the most sensitive data—such as intelligence agencies—do not typically partition the DBA function. Instead, they vet their DBAs strongly, because it is a position of trust.

Encryption of stored data must not interfere with the administration of the database; otherwise, larger security issues can result. For example, if by encrypting data you corrupt the data, you've created a security problem: data is not meaningful and may not be recoverable.

Encryption can be used to mitigate the ability of a DBA—or other privileged user—to see data in the database. However, it is not a substitute for vetting a DBA properly, or for limiting the use of powerful system privileges. If an untrustworthy user has significant privilege, there are multiple threats he can pose to an organization, and these may be far more significant than viewing un-encrypted credit card numbers.

Principle 3: Encrypting Everything Does Not Make Data Secure

It is a pervasive tendency to think that if storing some data encrypted strengthens security, then encrypting everything makes all data secure.

As described above, encryption does not address access control issues well. Consider the implications of encrypting an entire production database. All data must be decrypted to be read, updated, or deleted, and the encryption must not interfere with normal access controls. Encryption is inherently a performance-intensive operation; encrypting all data will significantly affect performance. Availability is a key aspect of security and if, by encrypting data, you make data unavailable, or the performance adversely affects availability, then you have created a new security problem.

Encryption keys must be changed regularly as part of good security practice, which necessitates that the database be inaccessible while the data is being decrypted and re-encrypted with a new key or keys. This also adversely affects availability.

While encrypting all or most of the data in a production database is clearly a problem, there may be advantages to encrypting data stored off-line. For example, an organization may store backups for a period of six months to a year off-line, in a remote location. Of course, the first line of protection is to secure the data in a facility to which access is controlled—a physical measure. However, there may be a benefit to encrypting this data before it is stored. Since it is not being accessed on-line, performance need not be a consideration. While Oracle9*i* does not provide this facility, there are vendors who can provide such encryption services. Before embarking on large-scale encryption of backup data, organizations considering this approach should thoroughly test the process. It is essential that any data which is encrypted before off-line storage, can be decrypted and re-imported successfully.

Solutions For Stored Data Encryption in Oracle9i

`DBMS_OBFUSCATION_TOOLKIT` provides several means for addressing the security issues that have been discussed. This section includes these topics:

- Oracle9i Data Encryption Capabilities
- Data Encryption Challenges

Oracle9*i* Data Encryption Capabilities

While there are many security threats that encryption cannot address well, it is clear that an additional measure of security can be achieved by selectively encrypting sensitive data before storage in the database. Examples of such data could include:

- Credit card numbers
- National identity numbers
- Passwords for applications whose users are not database users

To address these needs, Oracle9*i* provides a PL/SQL package to encrypt and decrypt stored data. The package, `DBMS_OBFUSCATION_TOOLKIT`, is provided in both Standard Edition and Enterprise Edition Oracle9*i*. This package currently supports bulk data encryption using the Data Encryption Standard (DES) algorithm, and includes procedures to encrypt (DESEncrypt) and decrypt (DESDecrypt) using DES. The DBMS_OBFUSCATION_TOOLKIT also includes functions to encrypt and decrypt using 2-key and 3-key DES, in outer cipher block chaining mode. They require keylengths of 128 and 192 bits, respectively.

The DBMS_OBFUSCATION_TOOLKIT includes a cryptographic checksumming capabilities (MD5), and the ability to generate a secure random number (GetKey). Secure random number generation is important part of cryptography; predictable keys are easily-guessed keys, and easily-guessed keys may lead to easy decryption of data. Most cryptanalysis is done by finding weak keys or poorly-stored keys, rather than through brute force analysis (cycling through all possible keys).

Note: Do not use DBMS_RANDOM as it is unsuitable for cryptographic key generation.

Key management is programmatic. That is, the application (or caller of the function) must supply the encryption key; and this means that the application developer must find a way of storing and retrieving keys securely. The relative strengths and weaknesses of various key management techniques are discussed below. The `DBMS_OBFUSCATION_TOOLKIT` package, which can handle both string and raw data, requires the submission of a 64-bit key. The DES algorithm itself has an effective key length of 56-bits.

The DBMS_OBFUSCATION_TOOLKIT is granted to PUBLIC by default. Oracle strongly recommends that this grant be revoked. In general, there is no reason why users should be able to encrypt stored data outside the context of an application.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for documentation of the `DBMS_OBFUSCATION_TOOLKIT` package

---

Data Encryption Challenges

Even in cases where encryption can provide additional security, it is not without its technical challenges. These challenges are described in the following section, which includes:

- Encrypting Indexed Data
- Key Management
- Key Transmission
- Key Storage
- Changing Encryption Keys
- Binary Large Objects (BLOBS)

Encrypting Indexed Data

Special difficulties arise in handling encrypted data which is indexed. For example, suppose a company uses a national identity number (such as the U.S. Social Security number (SSN)) as the employee number for its employees. The company considers employee numbers to be very sensitive data and therefore wants to encrypt data in the `EMPLOYEE_NUMBER` column of the `EMPLOYEES` table. Because `EMPLOYEE_NUMBER` contains unique values, the database designers want to have an index on it for better performance.

However, if the `DBMS_OBFUSCATION_TOOLKIT` (or another mechanism) is used to encrypt data in a column, then an index on that column will also contain encrypted values. However, although the index can be used for equality checking (for example, `‘SELECT * FROM emp WHERE employee_number = ‘123245’`), if the index on that column contains encrypted values, then the index is essentially unusable for any other purpose. Oracle therefore recommends that developers not encrypt indexed data.

One way of solving this problem would be for the company seeking to encrypt national identity numbers to create an alternate, uniquely identifying number for each of its employees. The company could subsequently create an index on these alternate employee numbers and retain them in clear text. The corresponding national identity numbers could be placed in a separate column without indexing, and the values in it could be encrypted by an application that would also handle decrypting appropriately. In this manner, the national identity number could be obtained when necessary but would not be used as a unique number to identify employees.

Given the privacy issues associated with overuse of national identity numbers (for example, identity theft), the fact that some allegedly unique national identity numbers have duplicates (as with U.S. Social Security numbers), and the ease with which a sequence can generate a unique number, there are many good reasons to avoid using national identity numbers as unique IDs.

Key Management

To address the issue of secure cryptographic key generation, Oracle9i adds support for a secure random number generation, the GetKey procedure of the DBMS_OBFUSCATION_TOOLKIT. The GetKey procedure calls the secure random number generator (RNG) that has previously been certified against the Federal Information Processing Standard (FIPS)-140 as part of the Oracle Advanced Security FIPS-140 evaluation. Developers should not, under any circumstances use the DBMS_RANDOM package. The DBMS_RANDOM package generates pseudo-random numbers; as RFC-1750 states, "The use of pseudo-random processes to generate secret quantities can result in pseudo-security."

Key Transmission

If the key is to be passed by the application to the database, then it must be encrypted. Otherwise, a snooper could grab the key as it is being transmitted. Use of network encryption, such as that provided by Oracle Advanced Security, will protect all data in transit from modification or interception, including cryptographic keys.

Key Storage

Key storage is one of the most important, yet difficult, aspects of encryption. To recover data encrypted with a symmetric key, the key must be accessible to the application or user seeking to decrypt the data. The key needs to be easy enough to retrieve that users can access encrypted data, without significant performance degradation. The key needs to be secure enough not to be easily recoverable by someone who is maliciously trying to access encrypted data which he is not supposed to see.

The three basic options available to a developer are:

- Store the key in the database
- Store the key in the operating system
- Have the user manage the key

Storing the Keys in the Database

Storing the keys in the database cannot always provide "bullet-proof" security if you are trying to protect against the DBA accessing encrypted data. This is because an all-privileged DBA could access tables containing encryption keys. However, it can often provide quite good security against the casual snooper or against someone compromising the database file on the operating system.

As a trivial example, suppose you create a table (`EMP`) that contains employee data. You want to encrypt each employee's Social Security number (one of the columns). You could encrypt each employee's SSN using a key which is stored in a separate column. However, anyone with `SELECT` access on the entire table could retrieve the encryption key and decrypt the matching SSN.

While this encryption scheme seems easily defeatable, with a little more effort you can create a solution that is much harder to break. For example, you could encrypt the SSN using a technique that performs some additional data transformation on the `employee_number` before using it to encrypt the SSN. This technique might be something as simple, for example, as XORing the `employee_number` with the birthdate of the employee.

As additional protection, a PL/SQL package body performing encryption can be "wrapped," (using the `WRAP` utility) which obfuscates the code so that the package body cannot be read. For example, putting the key into a PL/SQL package body and then wrapping it makes the package body—including the embedded key—unreadable to the `DBA` and others. A developer could wrap a package body called `KEYMANAGE` as follows:

```
wrap iname=/mydir/keymanage.sql
```

A developer can subsequently have a function in the package call the `DBMS_OBFUSCATION_TOOLKIT` with the key contained in the wrapped package.

While wrapping is not unbreakable, it makes it harder for a snooper to get the key. To make it even more difficult, the key could be split up in the package and then have the procedure to re-assemble it prior to use. Even in cases where a different key is supplied for each encrypted data value, so that the value of the key is not embedded within a package, wrapping the package that performs key management (that is, data transformation or padding) is recommended.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for additional information about the `WRAP` utility

An alternative would be to have a separate table in which to store the encryption key and to envelope the call to the keys table with a procedure. The key table can be joined to the data table using a primary key to foreign key relationship; for example, `EMPLOYEE_NUMBER` is the primary key in the `EMPLOYEES` table which stores employee information, and the encrypted SSN. `EMPLOYEE_NUMBER` is a foreign key to the `SSN_KEYS` table which stores the encryption keys for each employee's SSN. The key stored in the `SSN_KEYS` table can also be transformed before use (i.e. through XORing), so the key itself is not stored un-encrypted. The procedure itself should be wrapped, to hide the way in which keys are transformed before use.

The strengths of this approach are:

- Users who have direct table access cannot see the sensitive data un-encrypted, nor can they retrieve the keys to decrypt the data.

- Access to decrypted data can be controlled through a procedure that selects the (encrypted) data, retrieves the decryption key from the key table, and transforms it before it can be used to decrypt the data.
- The data transformation algorithm is hidden from casual snooping by wrapping the procedure, which obfuscates the procedure code.
- SELECT access to both the data table and the keys table does not guarantee that the user with this access can decrypt the data, because the key is transformed before use.

The weakness in this approach is that a user who has SELECT access to both the key table and the data table, who can derive the key transformation algorithm, can break the encryption scheme.

The above approach is not bullet-proof, but it is good enough to protect against easy retrieval of sensitive information (such as credit card numbers) stored in clear text.

Storing the Keys in the Operating System

Storing keys in the operating system (in a flat file) is another option. Oracle9*i* allows you to make callouts from PL/SQL, which you could use to retrieve encryption keys. However, if you store keys in the operating system and make callouts to it, then your data is only as secure as the protection on the operating system. If your primary security concern driving you to encrypt data stored in the database is that the database can be broken into from the operating system, then storing the keys in the operating system arguably makes it easier for a hacker to retrieve encrypted data than storing the keys in the database itself.

User Managing the Keys

Having the user supply the key assumes the user will be responsible with the key. Considering that 40% of help desk calls are from users who have forgotten their passwords, you can see the risks of having users manage encryption keys. In all likelihood, users will either forget an encryption key, or write the key down, which then creates a security weakness. If a user forgets an encryption key or leaves the company, then your data is unrecoverable.

If you do elect to have user-supplied or user-managed keys, then you need to make sure you are using network encryption so the key is not passed from client to server in the clear. You also must develop key archive mechanisms, which is also a difficult security problem. Key archives or "backdoors" create the security weaknesses that encryption is attempting to address in the first place.

Changing Encryption Keys

Prudent security practice dictates that you periodically change encryption keys. For stored data, this requires periodically un-encrypting the data, and re-encrypting it with another well-chosen key. This would likely have to be done while the data is not being accessed, which creates another challenge. This is especially true for a Web-based application encrypting credit card numbers, since you do not want to bring the entire application down while you switch encryption keys.

Binary Large Objects (BLOBS)

Certain datatypes require more work to encrypt. For example, Oracle supports storage of binary large objects (BLOBs), which lets users store very large objects (e.g. gigabytes) in the database. A BLOB can be either stored internally as a column, or stored in an external file. To use the DBMS_OBFUSCATION_TOOLKIT, the user would have to split the data into 32767 character chunks (the maximum that PL/SQL allows) and then would have to encrypt the chunk and append it to the BLOB. To decrypt, the same procedure would have to be followed in reverse.

Example of Data Encryption PL/SQL Program

Following is a sample PL/SQL program to encrypt data. It shows test string data encryption and decryption. The interface for encrypting raw data is similar.

```
DECLARE
 input_string       VARCHAR2(16) := 'tigertigertigert';
 key_string         VARCHAR2(8)  := 'scottsco';

 encrypted_string           VARCHAR2(2048);
 decrypted_string           VARCHAR2(2048);
   error_in_input_buffer_length EXCEPTION;
   PRAGMA EXCEPTION_INIT(error_in_input_buffer_length, -28232);
   INPUT_BUFFER_LENGTH_ERR_MSG VARCHAR2(100) :=
    '*** DES INPUT BUFFER NOT A MULTIPLE OF 8 BYTES ***';

BEGIN
   dbms_output.put_line('> ========= BEGIN TEST =========');
   dbms_output.put_line('> Input String                        : ' ||
 input_string);
   BEGIN
      dbms_obfuscation_toolkit. input_string => input_string,
               key_string => key_string, encrypted_string => encrypted_string );
      dbms_output.put_line('> encrypted string             : ' ||
encrypted_string);
```

```
   dbms_obfuscation_toolkit.DESDecrypt(input_string => encrypted_string,
            key => raw_key, decrypted_string => decrypted_string);
   dbms_output.put_line('> Decrypted output             : ' ||
                 decrypted_string);
   dbms_output.put_line('>  ');
   if input_string =
                 decrypted_string THEN
      dbms_output.put_line('> DES Encryption and Decryption successful');
   END if;
EXCEPTION
   WHEN error_in_input_buffer_length THEN
          dbms_output.put_line('> ' || INPUT_BUFFER_LENGTH_ERR_MSG);
END;
```

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

Part IV

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:

15

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 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 the functionality already built into Oracle. For example, do not define triggers to enforce data integrity rules that can be easily enforced using declarative integrity constraints.
- Limit the size of triggers. If the logic for your trigger requires much more than 60 lines of PL/SQL code, then 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;
/
```

If you enter a SQL statement, such as the following:

```
UPDATE Emp_tab SET sal = sal + 500.00 WHERE deptno = 10;
```

Then, the trigger fires once for each row that is updated, and it prints the new and old salaries, and the difference.

The CREATE (or CREATE OR REPLACE) statement fails if any errors exist in the PL/SQL block.

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: For more realistic examples of CREATE TRIGGER statements, see "Examples of Trigger Applications" on page 15-33.

Prerequisites for Creating Triggers

Before creating any triggers, run the `CATPROC.SQL` script while connected as `SYS`. This script automatically runs all of the scripts required for, or used within, the procedural extensions to the Oracle Server.

> **See Also:** The location of this file is operating system dependent; see your platform-specific Oracle documentation.

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 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:** *Oracle9i SQL Reference* explains the syntax for creating triggers.

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:** For a complete list of the functions you can call to get the event attributes, see Chapter 16, "Working With System Events".

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 15-53.

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).

Triggering Statement

The triggering statement specifies the following:

- The type of 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 ... ;
```

INSERT Trigger Behavior

`INSERT` triggers will fire during import and during SQL*Loader conventional loads. (For direct loads, triggers are disabled before the load.)

For example, you have three tables: A, B, and C. You also have an `INSERT` trigger on table A which looks from table B and inserts into table C. If you import table A, then table C is also updated.

Note: The `IGNORE` parameter determines whether triggers will be fired during import. If `IGNORE=N` (default), then import does not load an already existing table, so no pre-existing triggers will fire. If the table did not exist, then import creates and loads it before any triggers are defined, so again, they do not fire. If `IGNORE=Y`, then import loads rows into existing tables. Triggers will fire, and indexes will be maintained.

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.

Note: AFTER row triggers are slightly more efficient than BEFORE row triggers. With BEFORE row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement.

Alternatively, with AFTER row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

Modifying Complex Views (INSTEAD OF Triggers)

The INSTEAD OF option can also be used in triggers. INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through UPDATE, INSERT, and DELETE statements. These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger *instead* of executing the triggering statement. The trigger performs UPDATE, INSERT, or DELETE operations directly on the underlying tables.

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 15-12

Notes:

- 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:

- Set operators
- Group functions
- GROUP BY, CONNECT BY, or START WITH clauses
- The DISTINCT operator
- Joins (a subset of join views are updatable)

If a view contains pseudocolumns or expressions, 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. Now, 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: For the order of trigger firing, see *Oracle9i Database Concepts*.

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 below. 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
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's body, but a colon is not allowed when using the qualifiers in the WHEN clause or the REFERENCING option.

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.

Conditional Predicates

If more than one type of DML operation can fire a trigger (for example, `ON INSERT OR DELETE OR UPDATE OF Emp_tab`), then the trigger body can use the conditional predicates `INSERTING`, `DELETING`, and `UPDATING` to run specific blocks of code, depending on the type of statement that fires the trigger. Assume this is the triggering statement:

```
INSERT OR UPDATE ON Emp_tab
```

Within the code of the trigger body, you can include the following conditions:

```
IF INSERTING THEN ... END IF;
IF UPDATING THEN ... END IF;
```

The first condition evaluates to TRUE only if the statement that fired the trigger is an INSERT statement; the second condition evaluates to TRUE only if the statement that fired the trigger is an UPDATE statement.

In an UPDATE trigger, a column name can be specified with an UPDATING conditional predicate to determine if the named column is being updated. For example, assume a trigger is defined as 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. The following statement fires the above trigger and causes the UPDATING (sal) conditional predicate to evaluate to TRUE:

```
UPDATE Emp_tab SET Sal = Sal + 100;
```

Error Conditions and Exceptions in the Trigger Body

If a predefined or user-defined error condition or exception is raised during the execution of a trigger body, 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 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 above 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 above 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, LONG RAW, and LOB Datatypes `LONG`, `LONG RAW`, and `LOB` 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.
- `LOB` values for :`NEW` variables cannot be modified in the trigger body. For example:

```
:NEW.Column := ...
```

This is not allowed if `column` is of `LOB` datatype.

Note: Previously, `column`, in this example, would not have been allowed if it was an object, a varray, or a nested table. This restriction has been lifted in release 8.1.5.

BEFORE Triggers Fired Multiple Times If an `UPDATE` or `DELETE` statement detects a conflict with a concurrent `UPDATE`, then Oracle performs a transparent `ROLLBACK` to `SAVEPOINT` and restarts the update. This can occur many times before the statement completes successfully. Each time the statement is restarted, the `BEFORE` statement trigger is fired again. The rollback to savepoint does not undo changes to any package variables referenced in the trigger. The package should include a counter variable to detect this situation.

Row Evaluation Order for 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 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 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 chooses an arbitrary order to execute these triggers.

See Also: *Oracle9i Database Concepts* has more information on the firing order of triggers.

Each subsequent trigger sees the changes made by the previously fired triggers. Each trigger can see the old and new values. The old values are the original values, and the new values are the current values, as set by the most recently fired `UPDATE` or `INSERT` trigger.

To ensure that multiple triggered actions occur in a specific order, you must consolidate these actions into a single trigger (for example, by having the trigger call a series of procedures).

You cannot open a database that contains multiple triggers of the same type if you are using any version of Oracle before release 7.1. You also cannot open such a database if your `COMPATIBLE` initialization parameter is set to a version earlier than 7.1.0. For system triggers, compatibility must be 8.1.0.

Trigger Restrictions on Mutating Tables A *mutating* table is a table that is currently being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or it is a table that might need to be updated by the effects of a declarative `DELETE CASCADE` referential integrity constraint. The restrictions on such a table apply only to the session that issued the statement in progress.

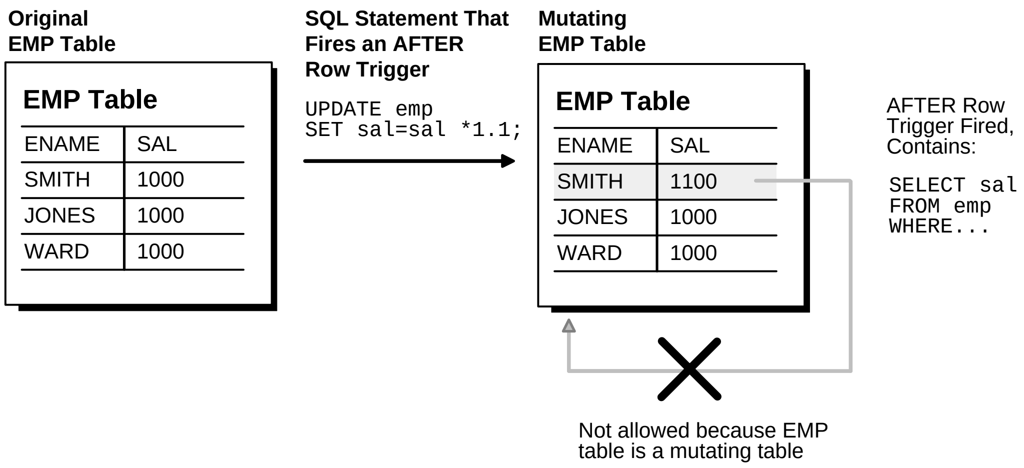
Tables are never considered mutating *for statement triggers* unless the trigger is fired as the result of a `DELETE CASCADE`. Views are not considered mutating in `INSTEAD OF` triggers.

For all row triggers, or for statement triggers that were fired as the result of a `DELETE CASCADE`, there are two important restrictions regarding mutating tables. These restrictions prevent a trigger from seeing an inconsistent set of data.

- The SQL statements of a trigger cannot read from (query) or modify a mutating table of the triggering statement.

Figure 15–1 illustrates the restriction placed on mutating tables.

Figure 15–1 Mutating Tables



Notice that the SQL statement is run for the first row of the table, and then an AFTER row trigger is fired. In turn, a statement in the AFTER row trigger body attempts to query the original table. However, because the EMP table is mutating, this query is not allowed by Oracle. If attempted, then 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;
```

Then, the following error is returned:

```
ORA-04091: table SCOTT.Emp_tab is mutating, trigger/function may not see it
```

Oracle returns this error when the trigger fires, because the table is mutating when the first row is deleted. (Only one row is deleted by the statement, because `Empno` is a primary key, but Oracle has no way of knowing that.)

If you delete the line "`FOR EACH ROW`" from the trigger above, then the trigger becomes a statement trigger, the table is not mutating when the trigger fires, and the trigger does output the correct data.

If you need to update a mutating table, then you could use a temporary table, a PL/SQL table, or a package variable to bypass these restrictions. For example, in place of a single `AFTER` row trigger that updates the original table, resulting in a mutating table error, you may be able to use two triggers—an `AFTER` row trigger that updates a temporary table, and an `AFTER` statement trigger that updates the original table with the values from the temporary table.

Declarative integrity constraints are checked at various times with respect to row triggers.

> **See Also:** *Oracle9i Database Concepts* has information about the interaction of triggers and integrity constraints.

Because declarative referential integrity constraints are currently not supported between tables on different nodes of a distributed database, the 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.

You should not use loop-back database links to circumvent the trigger restrictions. Such applications might behave unpredictably.

Restrictions on Mutating Tables Relaxed Before Oracle8*i*, there was a "constraining error" that prevented a row trigger from modifying a table when the parent statement implicitly read that table to enforce a foreign key constraint. Starting with Oracle8*i*, there is no constraining error. Also, checking of the foreign key is deferred until at least the end of the parent statement.

The mutating error still prevents the trigger from reading or modifying the table that the parent statement is modifying. However, starting in Oracle 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

Nature of the Event Depending on the event, the publication functionality imposes different restrictions. It may not be possible for the server to impose all restrictions.

The restrictions that cannot be fully enforced are clearly documented. For example, certain DDL operations may not be allowed on DDL events.

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. On the other hand, if you DROP a trigger that should be fired *before* all DROP events, then the trigger fires before the DROP.

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 does not fire a trigger that is not committed.

> **See Also:** For other restrictions, see "List of Database Events" on page 16-8.

Foreign Function Callouts All restrictions on foreign function callouts will also apply.

Who Is the Trigger User?

If you enter the following statement:

```
SELECT Username FROM USER_USERS;
```

Then, in a trigger, the name of the owner of the trigger is returned, not the name of user who is updating the table.

Privileges

Privileges to Create Triggers

To create a trigger in your schema, you must have the CREATE TRIGGER system privilege, and either:

- Own the table specified in the triggering statement, or
- Have the ALTER privilege for the table in the triggering statement, or
- Have the ALTER ANY TABLE system privilege

To create a trigger in another user's schema, you must have the CREATE ANY TRIGGER system privilege. With this privilege, the trigger can be created in any schema and can be associated with any user's table. 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.

Privileges for Referenced Schema Objects

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger's owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger's owner, not the privilege domain of the user issuing the triggering statement. This is similar to stored procedures.

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.

Migration Issues for Triggers

Non-compiled triggers cannot be fired under compiled trigger releases (such as Oracle 7.3 and Oracle8). If you are upgrading from a non-compiled trigger release to a compiled trigger release, then all existing triggers must be compiled. The upgrade script `cat73xx.sql` invalidates all triggers, so that they are automatically recompiled when first run. (The *xx* stands for a variable minor release number.)

Downgrading from Oracle 7.3 or later to a release prior to 7.3 requires that you run the `cat73xxd.sql` downgrade script. This handles portability issues between stored and non-stored trigger releases.

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 9-41

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: The *Oracle9i Database Reference* provides 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 an Oracle database. For example, triggers are commonly used to:

- Provide sophisticated auditing
- Prevent invalid transactions
- Enforce referential integrity (either those actions not supported by declarative integrity constraints or across nodes in a distributed database)
- Enforce complex business rules
- Enforce complex security authorizations
- Provide transparent event logging
- Automatically generate derived column values
- Enable building complex views that are updatable
- Track system events

This section provides an example of each of the above 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. 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 Oracle `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's auditing features provide, compared to auditing defined by triggers.

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.
Declarative Method	Auditing features enabled using the standard Oracle features are easier to declare and maintain, and less prone to errors, when compared to auditing functions defined by triggers.
Auditing Options can be Audited	Any changes to existing auditing options can also be audited to guard against malicious database activity.
Session and Execution time Auditing	Using the database auditing features, records can be generated once every time an audited statement is 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 *Oracle9i 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 above conditional evaluates to TRUE, the
   user-specified error number and message is raised,
   the trigger stops execution, and the effects of the
   triggering statement are rolled back.  Otherwise, a
   new row is inserted into the predefined auditing
   table named AUDIT_EMPLOYEE containing the existing
   and new values of the Emp_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.

Another example of using triggers to do auditing is shown below. This trigger 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 4, "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's declarative integrity constraint features, triggers should only be used to enforce complex business rules that cannot be defined using standard integrity constraints. The declarative integrity constraint features provided with Oracle offer the following advantages when compared to constraints defined by triggers:

Centralized Integrity Checks	All points of data access must adhere to the global set of rules defined by the integrity constraints corresponding to each schema object.
Declarative Method	Constraints defined using the standard integrity constraint features are much easier to write and are less prone to errors, when compared with comparable constraints defined by triggers.

While most aspects of data integrity can be defined and enforced using declarative integrity constraints, triggers can be used to enforce complex business constraints not definable using declarative integrity constraints. For example, triggers can be used to enforce:

- `UPDATE` and `DELETE SET NULL`, and `UPDATE` and `DELETE SET DEFAULT` referential actions.
- Referential integrity when the parent and child tables are on different nodes of a distributed database.
- Complex check constraints not definable using the expressions allowed in a `CHECK` constraint.

Referential Integrity Using Triggers

Many cases of referential integrity can be enforced using triggers. However, only use triggers when you want to enforce the UPDATE and DELETE SET NULL (when referenced data is updated or deleted, all associated dependent data is set to NULL), and UPDATE and DELETE SET DEFAULT (when referenced data is updated or deleted, all associated dependent data is set to a default value) referential actions, or when you want to enforce referential integrity between parent and child tables on different nodes of a distributed database.

When using triggers to maintain referential integrity, declare the PRIMARY (or UNIQUE) KEY constraint in the parent table. If referential integrity is being maintained between a parent and child table in the same database, 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 example below allows this trigger to be used with the UPDATE_SET_DEFAULT and UPDATE_CASCADE triggers. This exception can be removed if this trigger is used alone.

```
CREATE 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;  -- used for cursor fetch below
   Invalid_department EXCEPTION;
   Valid_department   EXCEPTION;
   Mutating_table     EXCEPTION;
   PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

-- Cursor used to verify parent key value exists.  If
-- present, lock parent key's row so it can't be
-- deleted by another transaction until this
-- transaction is committed or rolled back.
  CURSOR 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;      -- used for cursor fetch below
   Employees_present     EXCEPTION;
   employees_not_present EXCEPTION;

   -- Cursor used to check for dependent foreign key values.
   CURSOR Dummy_cursor (Dn NUMBER) IS
      SELECT Deptno FROM Emp_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. 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;
```

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

Now 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;
/
```

Now, 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’s system event publication lets applications subscribe to database events, just like they subscribe to messages from other applications.

> **See Also:** Chapter 16, "Working With System Events"

Oracle’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.

Publication Framework

The Oracle framework allows declarative definition of system event publication. This enables triggers to support database events, and users can specify a procedure that is to be run when the event occurs. DML events are supported on tables, and system events are supported on `DATABASE` and `SCHEMA`.

The system event publication subsystem tightly integrates with the Advanced Queueing publish / subscribe engine. The `DBMS_AQ.ENQUEUE()` procedure is used by publish / subscribe applications, and callouts are used by non-publish / subscribe applications, like cartridges.

Users or administrators can enable publication of system events by creating triggers specifying the publication attributes. By default, a trigger (and, therefore, publication of events specified in the trigger) is enabled. Users can also disable publication of these events by disabling the trigger, using the `ALTER TRIGGER` statement.

> **See Also:** *Oracle9i SQL Reference*
>
> For details on how to subscribe to published events and how to specify the delivery of these published events, see *Oracle9i Application Developer’s Guide - Advanced Queuing* and *Oracle Call Interface Programmer’s Guide*

Event Publication

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, which then enables subscribers to get notifications.

Note: Detection of an event is predefined for a given release of the server. There is no user-specified event detection mechanism.

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 has been modified but not committed within the same transaction as the firing event, it is not fired. 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; therefore, it is possible that more than one publication is made in response to the same event, and there should be no publication ordering assumptions. The publications are made in the order in which the system events transpire.

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: For event-specific attributes, see "Event Attribute Functions" on page 16-2.

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: For details on return status, see "List of Database Events" on page 16-8.

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.

16

Working With System Events

System events, like `LOGON` and `SHUTDOWN`, provide a mechanism for tracking system changes. Oracle lets you combine this tracking with database event notification, which provides a simple and elegant method of delivering asynchrononous 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:

To use the information in this chapter, you need to understand Chapter 15, "Using Triggers". You access the event information inside a trigger body.

You can set up a flexible system for handling events using the Event panels in Oracle Enterprise Manager (OEM). OEM lets you detect more conditions than the system events in this chapter, and 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.

Notes:

- 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 16–1 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 under-lying 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;`

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');`

Table 16–1 System Defined Event Attributes (Cont.)

Attribute	**Type**	**Description**	**Example**

`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;`

`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;`

Table 16–1 System Defined Event Attributes (Cont.)

Attribute	Type	Description	Example

`ora_is_creating_nested_tabl`
`e`
	`BOOLEAN`	Return 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 16–1 System Defined Event Attributes (Cont.)

Attribute	Type	Description	Example

`ora_privileges(`
`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 revokee 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_privileges(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 16–1 System Defined Event Attributes (Cont.)

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 16–1 System Defined Event Attributes (Cont.)

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 16–2 contains a list of system manager events.

Table 16–2 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_servererro r`
					`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 16–3 contains a list of client events.

Table 16–3 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 16–3 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 16–3 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 16–3 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

17 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:

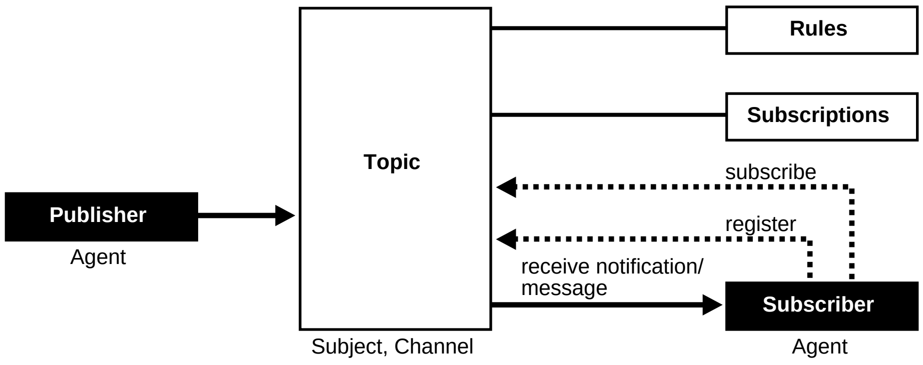
Introduction to Publish-Subscribe

Networking technologies and products now 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 17–1 illustrates publish and subscribe functionality.

Figure 17–1 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 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 16, "Working With System Events"

Advanced Queuing

Oracle Advanced Queuing 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: *Oracle9i Application Developer's Guide - Advanced Queuing*

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 functionalities.
-

```
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 code below 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));
}
```

Now, if user `SCOTT` logged on to the database, the client is notified, and the call back function `notifySnoop` is called.

Part V
Developing Specialized Applications

This part deals with application development scenarios that not every developer faces. Oracle provides a range of features to help with each kind of application.

This part contains:

18

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:

What Is a PL/SQL Web Application?

Web applications written in PL/SQL are typically sets of stored procedures that interact with web browsers through the HTTP protocol:

- Visiting a web page, following a hypertext link, or pressing a `Submit` button on an HTML form causes the database server to run a stored procedure.
- Any choices that a user makes on an HTML form are passed as parameters to the stored procedure. Parameters can also be hardcoded in the URL used to invoke the stored procedure.
- The results of the stored procedure are printed as tagged HTML text and are displayed in the browser as a web page. Web pages generated this way are **dynamic**: code runs inside the database server, producing HTML that varies depending on the database contents and the input parameters.

This kind of dynamic content is different from **dynamic HTML** (DHTML). With DHTML, the code is downloaded as Javascript or some other scripting language, and processed by the browser along with the HTML. A PL/SQL web application can print Javascript or other script code in its output, to produce complex DHTML that would be tedious to produce manually.

- The dynamic pages can contain links and HTML forms that call more stored procedures, to drill down or perform some other operation on the displayed data. The set of interlinked HTML pages forms the user interface of the web application.
- There are many techniques for coding dynamic pages, but PL/SQL is especially good for producing dynamic pages based on database processing. Its support for DML statements, dynamic SQL, cursors, and tight server integration provide both power and flexibility for web applications.
- Producing dynamic content using PL/SQL stored procedures gives you the flexibility and interactive behavior of CGI programs, without the memory overhead of forking a new CGI process each time.
- Certain other features associated with typical web applications, like the "statefulness" of Java servlets, can apply to PL/SQL stored procedures also. These features depend on the HTTP server that runs the stored procedures. To run PL/SQL stored procedures as servlets, you would use Oracle9*i* Application Server and its `mod_ose` module, as described in *Oracle Servlet Engine User's Guide*.

How Do I Generate 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 Oracle9*i* Application Server (iAS), Oracle Application Server (OAS), and WebDB:

```
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.

How Do I Pass 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.
- Hardcoded 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.
- Keeping the information in PL/SQL package variables, and running "stateful" stored procedures that act like Java servlets. This requires you to configure the `mod_ose` module of the Oracle Servlet Engine to be stateful.

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 the Oracle9*i* Application Server and its `mod_ose` module, as described in *Oracle Servlet Engine User's Guide*. This approach allows you to store state information in package variables that remain available as a user moves around a web site.

Performing Network Operations within PL/SQL Stored Procedures

While PL/SQL's built-in features are focused on traditional database operations and programming logic, Oracle supplies packages that open up Internet computing to PL/SQL programmers. You can find details and examples using most of these packages in the *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Sending Mail from PL/SQL

You can send mail from a PL/SQL program or stored procedure using the `UTL_SMTP` package:

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(conn);
    utl_smtp.quit(mail_conn);
    EXCEPTION
```

```
        WHEN OTHERS THEN
           -- Handle the error
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 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.

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.

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
-- Now 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, CGI Variables, and More from PL/SQL

Packages for all of these functions are supplied with Oracle8*i*. You use the packages in combination with any PL/SQL gateway, such as the Oracle Internet Application Server (iAS) and WebDB. 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.

For details, refer to the iAS book *Using mod_plsql*.

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 roundtrips.

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 the Oracle server at version 8.1.6 or later, together with a PL/SQL web gateway. Currently, the web gateways are Oracle Internet Application Server (iAS), the WebDB PL/SQL Gateway, and the OAS PL/SQL Cartridge. Before you start with PSP, you should have access to both the database server and the web server for one of these gateways.

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 for you, such as the page-building wizards in WebDB, 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 18-21 and "Examples of PL/SQL Server Pages" on page 18-25.

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 hardcoding 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 other 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 `<%@ page 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 casted, 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 18-25.

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 pathname such as `../include/`.

Note that the attribute names `contentType` and `errorPage` are case-sensitive.

Syntax

```
<%@ page [language="PL/SQL"] [contentType="content type string"] [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 pathname 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 pathname such as `../include/`.

For example:

```
loadpsp -replace -user scott/tiger@WEBDB banner.inc error.psp display_order.psp
```

In this example:

- The stored procedure is created in the database WEBDB. 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 and / or text 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 hardcoded 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 paramters 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 18-23 and "Running a PL/SQL Server Page Through a URL" on page 18-24 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

Now we have a dynamic page, but from the user’s 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>
```

The above 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.

- Intead 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:

```
htp.print(’preamble = "<img src=’’/images/graph_line.gif’’ height=’’8’’
width=’’"’);
```

and would make the indented Javascript code hard to read, with a separate `HTP.PRINT` call to output each line, even for very short lines.

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 URL's query string consists of name-value pairs, separated by equals signs, especially if you are passing parameters by constructing a hardcoded 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.

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 *Oracle9i Application Developer's Guide - XML*.

19 Porting Non-Oracle Applications to Oracle9i

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 features that are designed to make porting easy.

Topics include the following:

- Frequently Asked Questions About Porting

Frequently Asked Questions About Porting

How Do I Perform Natural Joins and Inner Joins?

When porting queries from other database systems to Oracle, you might have needed in the past to translate ANSI join notation into Oracle's comma notation.

You can now code Oracle 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 *Oracle9i SQL Reference* for full syntax of the `SELECT` statement and the support for join notation.

Is There an Automated Way to Migrate a Schema and Associated Data from Another Database System?

Yes. 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 Oracle8 and Oracle8*i*. Although the product itself runs on Windows, it can transfer data from databases on other operating systems, to Oracle databases running on any operating systems.

By using this product, you can avoid having to write your own applications to convert your legacy data when switching to Oracle8*i*. Related technology lets you convert certain kinds of source code, for example to migrate Visual Basic code to Java.

At the time of this publication, migration is supported from the following databases:

- MS SQL Server 6.5
- MS SQL Server 7.0
- MS Access 2.0
- MS Access 95
- MS Access 97
- Sybase Adaptive Server 11
- MySQL 3.22

For the current set of supported databases, see the web site:

`http://technet.oracle.com/tech/migration/`

How Do I Perform Large Numbers of 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.

  Oracle8i 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.

Does Oracle Support Scalar Subqueries?

Oracle9*i* does support scalar subqueries.

20 Working with Transaction Monitors with Oracle XA

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 the Oracle server. This allows inclusion of non-Oracle 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:

- For a general overview of XA, including basic architecture, see *X/Open CAE Specification - Distributed Transaction Processing: The XA Specification*. You can obtain a copy of this document by requesting X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3 from:

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

- For background and reference information about the Oracle XA library, see *Oracle Call Interface Programmer's Guide*.
- For information on library linking filenames, see the Oracle operating system-specific documentation.
- A `README.doc` file is located in a directory specified in the Oracle operating system-specific documentation and describes changes, bugs, or restrictions in the Oracle XA library for your platform since the last version.

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 20–1 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, the Oracle database server is an RM and uses its redo log and undo segments to return to a consistent state after a failure. An RM provides access to shared resources such as a database, file systems, printer servers, and so forth.

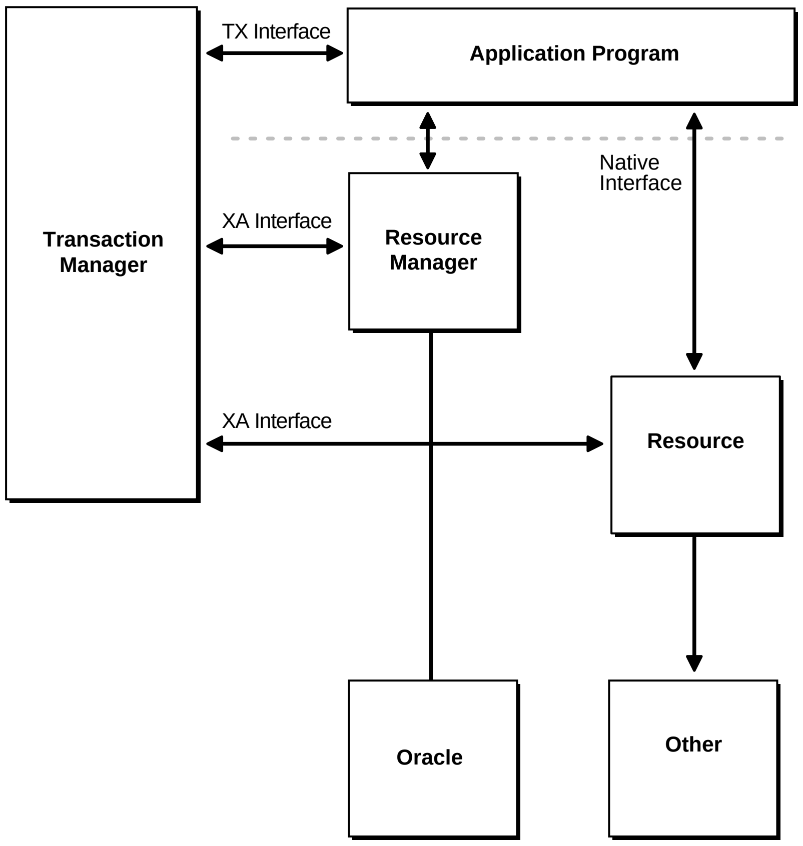
A transaction manager (TM) provides an application program interface (API) for specifying the boundaries of the transaction and manages the commit and recovery procedures.

Normally, Oracle acts as its own TM and manages its own commit and recovery. However, using a standards-based TM allows Oracle 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 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 20–1 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 is required to publish the following information.

`xa_switch_t` structures	The Oracle Server `xa_switch_t` structure name for static registration is `xaosw`. The Oracle Server `xa_switch_t` structure name for dynamic registration is `xaoswd`. These structures contain entry points and other information for the resource manager.
`xa_switch_t` resource manager	The Oracle Server resource manager name within the `xa_switch_t` structure is `Oracle_XA`.
close string	The close string used by `xa_close` () is ignored and is allowed to be null.
open string	The format of the open string used by `xa_open` () is described in detail in "Defining the xa_open String" on page 20-10.
libraries	Libraries needed to link applications using Oracle XA have operating system-specific names. It is similar to linking an ordinary precompiler or OCI program except you may have to link any TPM-specific libraries. If you are not using `sqllib`, 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 (or any other resource manager) through the Oracle XA library interface.

Support for Dynamic and Static Registration

Oracle 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 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 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

The following XA Library subroutines are available:

`xa_open`	Connects to the resource manager.
`xa_close`	Disconnects from the resource manager.
`xa_start`	Starts a new transaction and associate it with the given transaction ID (XID), or associates the process with an existing transaction.
`xa_end`	Disassociates the process from the given XID.
`xa_rollback`	Rolls back the transaction associated with the given XID.
`xa_prepare`	Prepares the transaction associated with the given XID. This is the first phase of the two-phase commit protocol.
`xa_commit`	Commits the transaction associated with the given XID. This is the second phase of the two-phase commit protocol.
`xa_recover`	Retrieves a list of prepared, heuristically committed or heuristically rolled back transaction.
`xa_forget`	Forgets the heuristic transaction associated with the given XID.

In general, the AP does not need to worry about these subroutines except to understand the role played by the `xa_open` string.

Extensions to the XA Interface

Oracle'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 error code to an XA error code. The first parameter is the service handle used to execute the work in the database. The second parameter is the error code that was returned from Oracle. Use this function to determine if the error returned from an OCI command was caused because the `xa_start` failed. The function returns `XA_OK` if the error was not generated by the XA module and a valid XA error if the error was generated by the XA module.

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 on page 20-9
- Responsibilities of the Application Developer on page 20-10
- Defining the xa_open String on page 20-10
- Interfacing XA with Precompilers and OCIs on page 20-17
- Transaction Control using XA on page 20-20
- Migrating Precompiler or OCI Applications to TPM Applications on page 20-22
- XA Library Thread Safety on page 20-24

Responsibilities of the DBA or System Administrator

The responsibilities of the DBA or system administrator are

1. Define the open string with the application developer's help.

   This is described in "Defining the xa_open String" on page 20-10.

2. Make sure the `DBA_PENDING_TRANSACTIONS` view exists on the database.

For Oracle Server Release 8.0:

Grant the select privilege to the `DBA_PENDING_TRANSACTIONS` view for all Oracle Server *user*(s) specified in the `xa_open` string.

For Oracle Server Release 7.3:

Make sure `V$XATRANS$` exists.

This view should have been created during the XA library installation. You can manually create the view, if needed, by running the SQL script `XAVIEW.SQL`. This SQL script should be executed as the Oracle user `SYS`. Grant the `SELECT` privilege to the `V$XATRANS$` view for all Oracle Server accounts which will be used by Oracle XA library applications.

See Also: Your Oracle operating system-specific documentation contains the location of the `XAVIEW.SQL` script.

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 an Oracle database server. See your TPM documentation to determine what environment exists for the process and what user ID it will have.

Be sure that correct values are set for `ORACLE_HOME` and `ORACLE_SID`.

> **See Also:** "Defining the xa_open String" on page 20-10 has information on how to specify a *sid* or a trace directory that is different from the defaults.

Also be sure to grant the user the `SELECT` privilege on `DBA_PENDING_TRANSACTIONS`.

4. Start up the relevant databases to bring Oracle XA applications on-line.

   This should be done before starting any TPM servers.

Responsibilities of the Application Developer

The application developer's responsibilities are

1. Define the open string with the DBA or system administrator's help.

   Defining the open string is described later in this section.

2. Develop the applications.

   Observe special restrictions on transaction-oriented SQL statements for precompilers.

   > **See Also:** "Interfacing XA with Precompilers and OCIs" on page 20-17

3. Link the application according to TPM vendor instructions.

Defining the xa_open String

The open string is used by the transaction monitor to open the database. The maximum number of characters in an open string is 256.

This section covers:

- Syntax of the xa_open String
- Required Fields

- Optional Fields

Syntax of the xa_open String

`Oracle_XA{+`*`required_fields`*`...} [+`*`optional_fields`*`...]`

where *required_fields* are:

`Acc=P//`

Or

`Acc=P`*/user/password*

`SesTm=`*session_time_limit*

and where *optional_fields* are:

`DB=`*db_name*

`LogDir=`*log_dir*

`MaxCur=`*maximum_#_of_open_cursors*

`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 "+" character as part of the actual information string.

Required Fields

Required fields for the open string are described in this section.

`Acc=P//`

or

Acc=P / *user/password*

`Acc`	Specifies user access information
`P`	Indicates that explicit user and password information is provided.
`P//`	Indicates that no explicit user or password information is provided, and that the operating system authentication form will be used. For more information see *Oracle9i Database Administrator's Guide.*
user	A valid Oracle Server account.
password	The corresponding current password.

For example, `Acc=P/scott/tiger` indicates that user and password information is provided. In this case, the user is `scott` and the password is `tiger`.

As previously mentioned, make sure that `scott` has the `SELECT` privilege on the `DBA_PENDING_TRANSACTIONS` table.

`Acc=P//` indicates that no user or password information is provided, thus defaulting to operating system authentication.

`SesTm=`*session_time_limit*

`SesTm`	Specifies the maximum length of time a transaction can be inactive before it is automatically aborted by the system.

session_time_limit This value should be the maximum time allowed in a transaction between one service and the next, or a service and the commit or rollback of the transaction.

For example, if the TPM uses remote procedure calls between the client and the servers, then `SesTM` applies to the time between the completion of one RPC and the initiation of the next RPC, or the `tx_commit`, or the `tx_rollback`.

The unit for this time limit is in seconds. The value of 0 indicates no limit. 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

Optional fields are described below.

`DB=`*db_name*

`DB`	Specifies the database name.

db_name	Indicates the name used by Oracle precompilers to identify the database. Application programs that use only the default database for the Oracle precompiler (that is, they do not use the AT clause in their SQL statements) should omit the `DB=`*db_name* clause in the open string. Applications that use explicitly named databases should indicate that database name in their `DB=`*db_name* field. Version 7 OCI programs need to call the `sqlld2`() function to obtain the correct `lda_def`, which is the equivalent of a service context. Version 8 OCI programs need to call the `xaoSvcCtx` function to get the `OCISvcCtx` service context. The *db_name* is not the *sid* and is not used to locate the database to be opened. Rather, it correlates the database opened by this open string with the name used in the application program to execute SQL statements. The *sid* is set from either the environment variable `ORACLE_SID` of the TPM application server or the *sid* given in the Oracle Net (formerly SQL*Net and Net8) clause in the open string. The Oracle Net clause is described later in this section. Some TPM vendors provide a way to name a group of servers that use the same open string. The DBA may find it convenient to choose the same name both for that purpose and for *db_name*.

For example, `DB=payroll` indicates that the database name is "payroll", and that the application server program will use that name in AT clauses.

`LogDir=`*log_dir*

LogDir	Specifies the directory on a local machine where the Oracle XA library error and tracing information may be logged.
log_dir	Indicates the 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 "Transaction Branches" on page 20-30 for a complete explanation.

`Objects`=*true/false*

Objects	Specifies whether the application is process 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*

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++ Precompiler Programmer's Guide*

SqlNet=*db_link*

SqlNet	Specifies the Oracle Net (formerly, SQL*Net and Net8) database link.
db_link	Indicates the string to use to log on to the system. The syntax for this string is the same as that used to set the `TWO-TASK` environment variable.

For example, `SqlNet=hqfin@NEWDB` indicates the database with *sid*=`NEWDB` accessed at host `hqfin` by TCP/IP.

The `SqlNet` parameter can be used to specify the `ORACLE_SID` in cases where you cannot control the server environment variable. It must also be used when the server needs to access more than one Oracle Server database. To use the Oracle Net string without actually accessing a remote database, use the Pipe driver.

For example:

```
SqlNet=localsid1
```

Where:

`localsid1`	An alias defined in the Oracle Net `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*

`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 waits before `XA_RETRY` is returned.

`Threads`=*true/false*

Threads	Specifies whether the application is multi-threaded. The default value is False.
true/false	If the application is multi-threaded, 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 Interfaces (OCIs).

Using Precompilers with the Oracle XA Library

When used in an Oracle XA application, cursors are valid only for the duration of the transaction. Explicit cursors should be opened after the transaction begins, and closed before the commit or rollback.

There are two options to choose from when interfacing with precompilers:

- Using precompilers with the default database
- Using precompilers with a named database

The following examples use the precompiler Pro*C/C++.

Using Precompilers with the Default Database

To interface to a precompiler with the default database, make certain that the `DB=`*db_name* field, used in the open string, is not present. The absence of this field indicates the default connection, and only one default connection is allowed 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 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: Oracle recommends against using XA applications to create connections. Any work performed would be outside the global transaction and would have to be committed separately.

Using OCI with the Oracle XA Library

OCI applications that use the Oracle XA library should not call `OCISessionBegin`() (`olon`() or `orlon`() in Version 7) to log on to the resource manager. Rather, the logon should be done through the TPM. The applications can execute the function `xaoSvcCtx`() (`sqlld2`() in Version 7) to obtain the service context (`lda` in Version 7) structure they need to access the resource manager.

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 Server 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* has 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 below.

The transaction managers typically control the transactions through the TX interface. It includes the following functions:

`tx_open`	Logs into the resource manager(s)
`tx_close`	Logs out of the resource manager(s)
`tx_begin`	Starts a new transaction
`tx_commit`	Commits a transaction
`tx_rollback`	Rolls back the transaction

Most TPM applications are written using a client-server architecture where an application client requests services and an application server provides services. The examples that follow use such a client-server model. A service is a logical unit of work, which in the case of the Oracle Server as the resource manager, comprises a set of SQL statements that perform a related unit of work.

For example, when a service named "credit" receives an account number and the amount to be credited, it 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 RELEASE WORK` (for precompilers), or `OCISessionEnd()` (for OCIs) by `tx_close()`. The V7 equivalent for `OCISessionBegin()` was `olon()` and for `OCISessionEnd()`, `ologof()`.

3. Ensure that the application replaces the regular commit / rollback statements and begins the transaction explicitly.

   For example, replace the commit / rollback statements `EXEC SQL COMMIT / ROLLBACK WORK` (for precompilers), or `ocom()/orol()` (for OCIs) by `tx_commit()/tx_rollback()` and start the transaction by calling `tx_begin()`.

4. Ensure that the application resets the fetch state prior to ending a transaction. In general, `release_cursor=no` should be used. Use `release_cursor=yes` only when you are certain that a statement will be executed only once.

Table 20–1 lists the TPM functions that replace regular Oracle commands when migrating precompiler or OCI applications to TPM applications.

Table 20–1 TPM Replacement Commands

Regular Oracle Commands	TPM Functions
`CONNECT` *user/password*	`tx_open` (possibly implicit)
implicit start of transaction	`tx_begin`
`SQL`	Service that executes the SQL
`COMMIT`	`tx_commit`
`ROLLBACK`	`tx_rollback`
disconnect	`tx_close` (possibly implicit)
`SET TRANSACTION READ ONLY`	Illegal

XA Library Thread Safety

If you use a transaction monitor that supports threads, 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 unthreaded 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 an Oracle system, once a thread has been started and establishes a connection, only that thread can use that connection. No other thread can make a call on that connection.

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-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 through XA, all other threads in the process that connect to Oracle 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 Server instance, or a logon authorization failure.

The name of the trace file is:

`xa_`*db_namedate*`.trc`

where *db_name* is the database name you specified in the open string field `DB=`*db_name,* and *date* is the date when the information is logged to the trace file.

If you do not specify `DB=`*db_name* in the open string, 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 developers.

- `0x4` Trace various other "interesting" calls made by the XA library, such as specific calls to the Oracle Call Interface. This is generally of use only to Oracle developers.

Trace File Locations

The trace file can be placed in one of the following locations:

- The trace file can be created in the `LogDir` directory as specified in the open string.
- If you do not specify `LogDir` in the open string, then the Oracle XA application attempts to create the trace file in the `$ORACLE_HOME/rdbms/log` directory, if it can determine where `$ORACLE_HOME` is located.
- If the Oracle XA application cannot determine where `$ORACLE_HOME` is located, then the trace file is created in the current working directory.

Trace File Examples

Examples of two types of trace files are discussed below:

The example, xa_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 Server information that was returned.

In-Doubt or Pending Transactions

In-doubt or pending transactions are transactions that have been prepared, but not yet committed to the database.

Generally, the transaction manager provided by the TPM system should resolve any failure and recovery of in-doubt or pending transactions. However, the DBA may have to override an in-doubt transaction in certain circumstances, such as when the in-doubt transaction is:

- Locking data that is required by other transactions
- Not resolved in a reasonable amount of time

For more information about overriding in-doubt transactions in the circumstances described above, or about how to decide whether the in-doubt transaction should be committed or rolled back, see the TPM documentation.

Oracle Server SYS Account Tables

There are four tables under the Oracle Server SYS account that contain transactions generated by regular Oracle Server applications and Oracle XA applications. They are DBA_PENDING_TRANSACTIONS, V$GLOBAL_TRANSACTIONS, DBA_2PC_PENDING and DBA_2PC_NEIGHBORS

For transactions generated by Oracle XA applications, the following column information applies specifically to the DBA_2PC_NEIGHBORS table.

- The DBID column is always xa_orcl
- The DBUSER_OWNER column is always *db_name*xa.oracle.com

Remember that the *db_name* is always specified as DB=*db_name* in the open string. If you do not specify this field in the open string, then the value of this column is NULLxa.oracle.com for transactions generated by Oracle XA applications.

For example, you could use the SQL statement below to obtain more information about in-doubt transactions generated by Oracle XA applications.

```
SELECT * FROM Dba_2pc_pending p, Dba_2pc_neighbors n
   WHERE p.Local_tran_id = n.Local_tran_id
      AND
      n.Dbid = 'xa_orcl';
```

Alternatively, if you know the format ID used by the transaction processing monitor, 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.

General XA Issues and Restrictions

Database Links

Oracle XA applications can access other Oracle Server databases through database links, with the following restrictions:

- Use the shared server (also known as Multi-Threaded Server) configuration.

  This means that the transaction processing monitors (TPMs) use shared servers to open the connection to Oracle. The operating system network connection required for the database link is opened by the dispatcher, instead of the Oracle server process. Thus, when a particular service or RPC completes, the transaction can be detached from the server so that it can be used by other services or RPCs.
- Access to the other database must use SQL*Net Version 2, Net8, or Oracle Net.
- The other database being accessed should be another Oracle Server database.

Assuming that these restrictions are satisfied, Oracle Server allows such links and propagates the transaction protocol (prepare, rollback, and commit) to the other Oracle Server databases.

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 Server 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 Option

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.

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 Server-specific statement that independently rolls back or commits a global transaction. However, you can use rollbacks and commits in a local transaction.

Do not use `EXEC SQL ROLLBACK WORK` for precompiler applications when you are in the middle of a global transaction. Similarly, an OCI application should not execute `OCITransRollback`(), or the Version 7 equivalent `orol`(). You can roll back a global transaction by calling `tx_rollback`().

Similarly, a precompiler application should not have the `EXEC SQL COMMIT WORK` statement in the middle of a global transaction. An OCI application should not execute `OCITransCommit`() or the Version 7 equivalent `ocom`(). Instead, use `tx_commit`() or `tx_rollback`() to end a global transaction.

DDL Statements

Because a DDL SQL statement, such as `CREATE TABLE`, implies an implicit commit, the Oracle XA application cannot execute any DDL SQL statements.

Session State

Oracle does not guarantee that session state will be valid between services. For example, if a service updates a session variable (such as a global package variable), 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`.

Miscellaneous XA Issues

Note the following additional information about Oracle XA:

Transaction Branches

Oracle Server transaction branches within the same global transaction can share locks in either a tightly or loosely coupled manner. However, if the branches are on different instances when running Oracle Real Application Clusters, then they will be loosely coupled.

In tightly coupled transaction branches, the locks are shared between the transaction branches. This means that updates performed in one transaction branch can be seen in other branches that belong to the same global transaction before the update is committed. The Oracle Server obtains the DX lock before executing any statement in a tightly coupled branch. Hence, the advantage of using loosely coupled transaction branches is that there 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 illustrated in Table 20–2.

Table 20–2 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

The Oracle Server does not support association migration (a means whereby a transaction manager may resume a suspended branch association in another branch).

Asynchronous Calls

The optional XA feature asynchronous XA calls is not supported.

Initialization Parameters

Set the `transactions` `init.ora` parameter to the expected number of concurrent global transactions.

The parameter `open_links_per_instance` specifies the number of migratable open database link connections. These dblink connections are used by XA transactions so that the connections are cached after a transaction is committed.

> **See Also:** "Database Links" on page 20-28

Maximum Connections for Each Thread

The maximum number of `xa_opens` for each thread is now 32. Previously, it was 8.

Installation

No scripts need be executed to use XA. It is necessary, however, to run the `xaview.sql` script to run Release 7.3 applications with the Oracle8 or later server. Grant the `SELECT` privilege on `SYS.DBA_PENDING_TRANSACTIONS` to all users that connect to Oracle through the XA interface.

Compatibility

The XA library supplied with Release 7.3 can be used with a Release 8.0 Oracle Server. You must use the Release 7.2 XA library with a Release 7.2 Oracle Server. You can use the 8.0 library with a Release 7.3 Oracle Server. There is only one case of backward compatibility: an XA application that uses Release 8.0 OCI works with a Release 7.3 Oracle Server, but only if you use `sqlld2` and obtain an `lda_def` before executing SQL statements. Client applications must remember to convert the Version 7 LDA to a service handle using `OCILdaToSvcCtx`() after completing the OCI calls.

Oracle8 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 Oracle8 XA libraries.

Link Order

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'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'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
- Loosely Coupled Transaction Branches Are Supported
- SQLLIB Is Not Needed for OCI Applications
- No Installation Script Is Needed to Run XA
- The XA Library Can Be Used with the Oracle Real Application Clusters Option on All Platforms
- Transaction Recovery for Oracle Real Application Clusters Has Been Improved
- Both Global and Local Transactions Are Possible
- The xa_open String Has Been Modified

Session Caching Is No Longer Needed

Session caching is unnecessary with the new OCI. Therefore, the old `xa_open` string parameter, `SesCacheSz`, has been eliminated. Consequently, you can also reduce the *sessions* `init.ora` parameter. Instead, set the *transactions* `init.ora` parameter to the expected number of concurrent global transactions. Because 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 the Oracle Server are Version 8.

> **See Also:** "Extensions to the XA Interface" on page 20-8

Loosely Coupled Transaction Branches Are Supported

The Oracle8 and later server supports both loosely and tightly coupled transaction branches in a single Oracle instance. The Oracle7 server supported only tightly coupled transaction branches in a single instance, and loosely coupled transaction branches in different instances.

SQLLIB Is Not Needed for OCI Applications

OCI applications used to require the use of `SQLLIB`. This 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 Version 8. It is, however, still necessary for Version 7.3 applications.

> **See Also:** "Responsibilities of the DBA or System Administrator" on page 20-9

The XA Library Can Be Used with the 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 now possible to have both global and local transactions within the same XA connection. Local transactions are transactions that are completely coordinated by the Oracle server. For example, the update below belongs to a local transaction.

```
CONNECT scott/tiger;
UPDATE Emp_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 acts as a subordinate and processes the XA commands issued by the transaction manager. The update shown below belongs to a global transaction.

```
xa_open(oracle_xa+acc=p/SCOTT/TIGER+sestm=10", 1, TMNOFLAGS);
                             /* Transaction manager opens */
                             /* connection to the Oracle server*/
tpbegin();                   /* begin global transaction, the transaction*/
                             /* manager issues XA commands to the oracle*/
                             /* server to start a global transaction */
UPDATE Emp_tab SET Sal = Sal + 1;
                             /* Update is performed in the */
                             /* global transaction*/
tpcommit();                  /* commit global transaction, */
                             /* the transaction manager issues XA commands*/
                             /* to the Oracle server to commit */
                             /* the global transaction */
```

The Oracle7 server forbids a local transaction from being started in an XA connection. The update shown below would return an `ORA-2041` error code.

```
xa_open("oracle_xa+acc=p/SCOTT/TIGER+sestm=10" , 1, TMNOFLAGS);
                             /* Transaction manager opens */
                             /*connection to the Oracle server */
UPDATE Emp_tab SET Sal = Sal + 1; /* Oracle 7 returns an error */
```

Oracle8 and later servers allow 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 an Oracle7 Server. If set to true, then global transaction branches are loosely coupled; in other words, locks are not shared between branches.

- `SesWt`

  This parameter's value indicates the time-out limit when waiting for a transaction branch that is being used by another session. If Oracle cannot switch to the transaction branch within `SesWt` seconds, then `XA_RETRY` is returned.

Two parameters have been made obsolete and should only be used when connected to an Oracle Server Release 7.3.

- `GPWD`

  The group password is not used by Oracle8 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 Oracle8 or later because session caching has been eliminated.

Index