Oracle® Database

Advanced Application Developer's Guide

11*g* Release 1 (11.1)

B28424-01

July 2007

Oracle Database Advanced Application Developer's Guide, 11*g* Release 1 (11.1)

B28424-01

Primary Author: Sheila Moore

Contributing Authors: D. Adams, L. Ashdown, M. Cowan, J. Melnick, R. Moran, E. Paapanen, J. Russell, R. Strohm, R. Ward

Contributors: D. Alpern, G. Arora, C. Barclay, D. Bronnikov, T. Chang, L. Chen, M. Davidson, R. Day, R. Decker, G. Doherty, D. Elson, A. Ganesh, M. Hartstein, Y. Hu, J. Huang, C. Iyer, N. Jain, R. Jenkins Jr., S. Kotsovolos, S. Kumar, C. Lei, B. Llewellyn, D. Lorentz, V. Moore, K. Muthukkaruppan, V. Moore, J. Muller, R. Murthy, R. Pang, B. Sinha, S. Vemuri, W. Wang, D. Wong, A. Yalamanchi, Q. Yu

Contents

Part II PL/SQL for Application Developers

7 Coding PL/SQL Subprograms and Packages

Preface

Oracle Database Advanced Application Developer's Guide explains topics that experienced application developers reference repeatedly. Information in this guide applies to features that work the same on all supported platforms, and does not include system-specific information.

Preface topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

Oracle Database Advanced Application Developer's Guide is intended for application developers who are either developing new applications or converting existing applications to run in the Oracle Database environment. This guide is also valuable to anyone who is interested in the development of database applications, such as systems analysts and project managers.

To use this document effectively, you need a working knowledge of:

- Application programming
- Structured Query Language (SQL)
- Object-oriented programming

Documentation Accessibility

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

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

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see the following documents in the Oracle Database 11*g* Release 1 (11.1) documentation set:

- *Oracle Database PL/SQL Language Reference*
- *Oracle Call Interface Programmer's Guide*
- *Oracle Database Security Guide*
- *Pro*C/C++ Programmer's Guide*
- *Oracle Database SQL Language Reference*
- *Oracle Database Administrator's Guide*
- *Oracle Database Concepts*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*
- *Oracle Database Globalization Support Guide*
- *Oracle Database Sample Schemas*

See also:

- *Oracle PL/SQL Tips and Techniques* by Joseph C. Trezzo. Oracle Press, 1999.
- *Oracle PL/SQL Programming* by Steven Feuerstein. 3rd Edition. O'Reilly & Associates, 2002.
- *Oracle PL/SQL Developer's Workbook* by Steven Feuerstein. O'Reilly & Associates, 2000.
- *Oracle PL/SQL Best Practices* by Steven Feuerstein. O'Reilly & Associates, 2001.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

`*_`*`view`* means all static data dictionary views whose names end with *`view`*. For example, `*_ERRORS` means `ALL_ERRORS`, `DBA_ERRORS`, and `USER_ERRORS`. For more information about any static data dictionary view, or about static dictionary views in general, see *Oracle Database Reference*.

What's New in Application Development?

What's New in Application Development? briefly describes new features of Oracle Database 11*g* Release 1 (11.1) and provides links to additional information.

Oracle Database 11*g* Release 1 (11.1) New Features

The new application development features for Release 11.1 are:

- WAIT Option for Data Definition Language (DDL) Statements
- Binary XML Support for Oracle XML Database
- Metadata for SQL Built-In Functions
- Enhancements to Regular Expression Built-in Functions
- Invisible Indexes
- Cross-Session PL/SQL Function Result Cache
- Sequences in PL/SQL Expressions
- PL/Scope
- PL/SQL Hierarchical Profiler
- Query Result Change Notification
- Flashback Transaction Backout
- Flashback Data Archives
- XA API Available Within PL/SQL
- Support for XA/JTA in Oracle Real Application Clusters (Oracle RAC) Environment
- Identity Code Package
- Enhanced Online Index Creation and Rebuilding
- Embedded PL/SQL Gateway
- Oracle Database Spawns Multithreaded extproc Agent Directly by Default

WAIT Option for Data Definition Language (DDL) Statements

DDL statements require exclusive locks on internal structures. If these locks are unavailable when a DDL statement is issued, the DDL statement fails, though it might have succeeded if it had been issued subseconds later. The `WAIT` option of the SQL statement `LOCK TABLE` allows a DDL statement to wait for its locks for a specified period of time before failing.

For more information, see "Choosing a Locking Strategy" on page 2-10.

Binary XML Support for Oracle XML Database

Binary XML is a third way to represent an XML document. Binary XML complements, rather than replaces, the existing object-relational storage and `CLOB` storage representations. Binary XML has two significant benefits:

- XML operations can be significantly optimized, whether or not an XML schema is available.
- The internal representation of XML is the same on disk, in memory, and on wire.

As with other storage mechanisms, the details of binary XML storage are transparent to you. You continue to use `XMLType` and its associated methods and operators.

For more information, see "Representing XML" on page 3-18.

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

Metadata for SQL Built-In Functions

Metadata for SQL built-in functions is accessible through dynamic performance (`V$`) views. Third-party tools can leverage built-in SQL functions without maintaining their metadata in the application layer.

For more information, see "Metadata for SQL Built-In Functions" on page 3-27.

Enhancements to Regular Expression Built-in Functions

The regular expression built-in functions `REGEXP_INSTR` and `REGEXP_SUBSTR` have increased functionality. A new regular expression built-in function, `REGEXP_COUNT`, returns the number of times a pattern appears in a string. These functions act the same in SQL and PL/SQL.

For more information, see "Oracle Database Implementation of Regular Expressions" on page 4-2.

See Also: *Oracle Database SQL Language Reference*

Invisible Indexes

An invisible index is maintained by Oracle Database for every Data Manipulation Language (DML) statement, but is ignored by the optimizer unless you explicitly set the parameter `OPTIMIZER_USE_INVISIBLE_INDEXES` to `TRUE` on a session or system level.

Making an index invisible is an alternative to making it unusable or dropping it. Using invisible indexes, you can do the following:

- Test the removal of an index before dropping it
- Create invisible indexes temporarily for specialized, nonstandard operations, such as online application upgrades, without affecting the behavior of existing applications

For more information, see "Drop Unused Indexes" on page 5-5.

Cross-Session PL/SQL Function Result Cache

Before Release 11.1, if you wanted your PL/SQL application to cache the results of a function, you had to design and code the cache and cache-management subprograms. If multiple sessions ran your application, each session had to have its own copy of the cache and cache-management subprograms. Sometimes each session had to perform the same expensive computations.

As of Release 11.1, PL/SQL provides a cross-session function result cache. Because the function result cache is stored in a shared global area (SGA), it is available to any session that runs your application.

For more information, see "Cross-Session PL/SQL Function Result Cache" on page 7-10.

> **See Also:** *Oracle Database PL/SQL Language Reference*

Sequences in PL/SQL Expressions

The pseudocolumns `CURRVAL` and `NEXTVAL` make writing PL/SQL source code easier for you and improve run-time performance and scalability. You can use *`sequence_name`*`.CURRVAL` and *`sequence_name`*`.NEXTVAL` wherever you can use a `NUMBER` expression.

For an example, see "Example of a PL/SQL Package Specification and Body" on page 7-10.

> **See Also:** *Oracle Database PL/SQL Language Reference*

PL/Scope

PL/Scope is a compiler-driven tool that collects and organizes data about user-defined identifiers from PL/SQL source code. Because PL/Scope is a compiler-driven tool, you use it through interactive development environments (such as SQL Developer and JDeveloper), rather than directly.

PL/Scope enables the development of powerful and effective PL/Scope source code browsers that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

For a detailed description of PL/Scope, see Chapter 8, "Using PL/Scope".

PL/SQL Hierarchical Profiler

Nonhierarchical (**flat**) profilers record the time that a program spends within each subprogram—the **function time** or **self time** of each subprogram. Function time is helpful, but often inadequate. For example, it is helpful to know that a program spends 40% of its time in the subprogram `INSERT_ORDER`, but it is more helpful to know which subprograms call `INSERT_ORDER` often and the total time the program spends under `INSERT_ORDER` (including its descendent subprograms). Hierarchical profilers provide such information.

The PL/SQL hierarchical profiler does the following:

- Reports the dynamic execution profile of your PL/SQL program, organized by subprogram calls
- Accounts for SQL and PL/SQL execution times separately
- Requires no special source or compile-time preparation
- Stores results in database tables (**hierarchical profiler tables**) for custom report generation by integrated development environment (IDE) tools (such as SQL Developer and third-party tools)

  To generate simple HTML reports from raw profiler output, you can use the `plshprof` command-line utility.

Each subprogram-level summary in the dynamic execution profile includes information such as:

- Number of calls to the subprogram
- Time spent in the subprogram itself (**function time** or **self time**)
- Time spent in the subprogram itself and in its descendent subprograms (**subtree time**)
- Detailed parent-children information, for example:
  - All callers of a given subprogram (parents)
  - All subprograms that a given subprogram called (children)
  - How much time was spent in subprogram *x* when called from *y*
  - How many calls to subprogram *x* were from *y*

You can browse the generated HTML reports in any browser. The browser's navigational capabilities, combined with well chosen links, provide a powerful way to analyze performance of large applications, improve application performance, and lower development costs.

For a detailed description of PL/SQL hierarchical profiler, see Chapter 9, "Using the PL/SQL Hierarchical Profiler".

Query Result Change Notification

Before Release 11.1, Continuous Query Notification (CQN) published only object change notifications, which result from DML or DDL changes to the objects associated with registered the queries.

As of Release 11.1, CQN can also publish query result change notifications, which result from DML or DDL changes to the result set associated with the registered queries. New static data dictionary views enable you to see which queries are registered for result-set-change notifications (see "Querying CQN Registrations" on page 12-24).

For more information, see Chapter 12, "Using Continuous Query Notification".

Flashback Transaction Backout

The `DBMS_FLASHBACK.TRANSACTION_BACKOUT` procedure rolls back a transaction and its dependent transactions while the database remains online. This recovery operation uses undo data to create and execute the compensating transactions that return the affected data to its original state.

For more information, see "Using Flashback Transaction Backout" on page 13-13.

Flashback Data Archives

A Flashback Data Archive provides the ability to store and track all transactional changes to a record over its lifetime. It is no longer necessary to build this intelligence into the application. A Flashback Data Archive is useful for compliance with record stage policies and audit reports.

For more information, see "Using Flashback Data Archives" on page 13-15.

XA API Available Within PL/SQL

The XA interface functionality that supports transactions involving multiple resource managers, such as databases and queues, is now available within PL/SQL. You can use PL/SQL to switch and share transactions across SQL*Plus sessions and across processes.

For more information, see "Using the DBMS_XA Package" on page 15-16.

Support for XA/JTA in Oracle Real Application Clusters (Oracle RAC) Environment

An XA transaction now spans Oracle RAC instances by default, allowing any application that uses XA to take full advantage of the Oracle RAC environment, enhancing the availability and scalability of the application.

For more information, see "Using Oracle XA with Oracle Real Application Clusters (Oracle RAC)" on page 15-22.

Identity Code Package

The Identity Code Package provides tools to store, retrieve, encode, decode, and translate between various product or identity codes, including Electronic Product Code (EPC), in an Oracle Database. The Identity Code Package provides new data types, metadata tables and views, and PL/SQL packages for storing EPC standard RFID tags or new types of RFID tags in a user table.

The Identity Code Package allows the Oracle Database to recognize EPC coding schemes, to support efficient storage and component-level retrieval of EPC data, and to comply with the EPCglobal Tag Data Translation 1.0 (TDT) standard that defines how to decode, encode, and translate between various EPC RFID tag representations.

The Identity Code Package also provides an extensible framework that enables you to use pre-existing coding schemes with applications that are not included in the EPC standard and adapt the Oracle Database both to these older systems and to evolving identity codes that might become part of a future EPC standard.

The Identity Code Package also lets you create your own identity codes by first registering the new encoding category, registering the new encoding type, and then registering the new components associated with each new encoding type.

For more information, see Chapter 17, "Using the Identity Code Package".

Enhanced Online Index Creation and Rebuilding

Online index creation and rebuilding no longer requires a DML-blocking lock.

Before Release 11.1, online index creation and rebuilding required a very short-term DML-blocking lock at the end of the rebuilding. The DML-blocking lock could cause a spike in the number of waiting DML operations, and therefore a short drop and spike of system usage. This system usage anomaly could trigger operating system alarm levels.

Embedded PL/SQL Gateway

The PL/SQL gateway enables a user-written PL/SQL subprogram to be invoked in response to a URL with parameters derived from an HTTP request. `mod_plsql` is a form of the gateway that exists as a plug-in to the Oracle HTTP Server. Now the PL/SQL gateway is also embedded in the database itself. The embedded PL/SQL gateway uses the internal Oracle XML Database Listener and does not depend on the Oracle HTTP Server. You configure the embedded version of the gateway with the `DBMS_EPG` package.

For more information, see "Using Embedded PL/SQL Gateway" on page 10-5.

Oracle Database Spawns Multithreaded extproc Agent Directly by Default

When an application calls an external C procedure, either Oracle Database or Oracle Listener starts the external procedure agent, `extproc`.

Before Release 11.1, Oracle Listener spawned the multithreaded `extproc` agent, and you defined environment variables for `extproc` in the file `listener.ora`.

As of Release 11.1, by default, Oracle Database spawns `extproc` directly, eliminating the risk that Oracle Listener might spawn `extproc` unexpectedly. This default configuration is recommended for maximum security. If you use it, you define environment variables for `extproc` in the file `extproc.ora`.

For more information, including situations in which you cannot use the default configuration, see "Loading External Procedures" on page 14-3.

1 Introduction to Oracle Programmatic Environments

Topics:

- Overview of Oracle Application Development
- Overview of PL/SQL
- Overview of Java Support Built into the Database
- Overview of Pro*C/C++
- Overview of Pro*COBOL
- Overview of OCI and OCCI
- Overview of Oracle Data Provider for .NET (ODP.NET)
- Overview of Oracle Objects for OLE (OO4O)
- Choosing a Programming Environment

Overview of Oracle Application Development

As an application developer, you have many choices when writing a program to interact with Oracle database:

- Client/Server Model
- Server-Side Coding
- Two-Tier and Three-Tier Models
- User Interface
- Stateful and Stateless User Interfaces

Client/Server Model

In a traditional client/server program, your application code runs on a client system; that is, a system other than the database server. Database calls are transmitted from the client system 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 system. Client/server programs are typically written by using precompilers, whereas 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 by using triggers that are executed automatically when changes occur in the database or stored subprograms (procedures and functions) that are invoked explicitly. Off-loading the work from your application lets you reuse code that performs verification and cleanup and control database operations from a variety of clients. For example, by making stored subprograms invocable through a Web server, you can construct a Web-based user interface that performs the same functions as a client/server application.

Two-Tier and 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**, a separate 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 in which the client system might need only a Web browser or other means of sending requests over the TCP/IP or HTTP protocols.

User Interface

The **user interface** is what your application displays to end users. It depends on the technology behind the application as well as the needs of the users themselves. Experienced users can enter SQL statements that are passed on to the database. Novice users can be shown a graphical user interface that uses the graphics libraries of the client system (such as Windows or X-Windows). Any of these traditional user interfaces can also be provided in a Web browser through HTML and Java.

Stateful and 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 more 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 in this way, the application is considered **stateful**.

Web or thin-client applications that are **stateless** are easier to develop. Stateless applications gather all the required information, process it using the database, and then start over 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 action to Web applications that are stateless by default. For example, an entry form on one Web page can pass information 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 system, 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

This section contains the following topics:

- What Is PL/SQL?

- Advantages of PL/SQL
- PL/SQL Web Development Tools

What Is PL/SQL?

PL/SQL is Oracle's procedural extension to SQL, the standard database access language. It is an advanced 4GL (fourth-generation programming language), which means that it is an application-specific language. PL/SQL and SQL have built-in treatment of the relational database domain.

In PL/SQL, you can manipulate data with SQL statements and control program flow with procedural constructs such as loops. You can also do the following:

- Declare constants and variables
- Define subprograms
- Use collections and object types
- Trap run-time errors

Applications written in any of the Oracle programmatic interfaces can invoke PL/SQL stored subprograms and send blocks of PL/SQL code to Oracle Database for execution. 3GL applications can access PL/SQL scalar and composite datatypes through host variables and implicit datatype conversion. A 3GL language is easier than assembler language for a human to understand and includes features such as named variables. Unlike 4GL, it is not specific to an application domain.

Example 1–1 provides an example of a simple PL/SQL subprogram. The procedure `debit_account` withdraws 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, then the procedure jumps to an error routine; otherwise, it updates the bank account.

Example 1–1 Simple PL/SQL Example

```
PROCEDURE debit_account (p_acct_id INTEGER, p_debit_amount REAL)
IS
   v_old_balance  REAL;
   v_new_balance  REAL;
   e_overdrawn    EXCEPTION;
BEGIN
   SELECT bal
     INTO v_old_balance
   FROM accts
   WHERE acct_no = p_acct_id;
   v_new_balance := v_old_balance - p_debit_amount;
   IF v_new_balance < 0 THEN
      RAISE e_overdrawn;
   ELSE
      UPDATE accts SET bal = v_new_balance
         WHERE acct_no = p_acct_id;
   END IF;
   COMMIT;
EXCEPTION
   WHEN e_overdrawn THEN
      -- handle the error
END debit_account;
```

See Also:

- *Oracle Database PL/SQL Language Reference*
- *Oracle Database SQL Language Reference*

Advantages of PL/SQL

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

- Integration with Oracle Database
- High Performance
- High Productivity
- Scalability
- Manageability
- Object-Oriented Programming Support
- Portability
- Security
- Packages

Integration with Oracle Database

PL/SQL enables you use all of the Oracle Database SQL data manipulation, cursor control, and transaction control statements. PL/SQL also supports the SQL functions, operators, and pseudocolumns. You can manipulate data in Oracle Database flexibly and safely.

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

PL/SQL supports Dynamic SQL, which is a programming technique that enables you to 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.

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

High Performance

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

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

High Productivity

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

PL/SQL is the same in all environments. When you master PL/SQL with one Oracle tool, you can transfer your knowledge to Oracle tools, multiplying your productivity gains. For example, scripts written with one tool can be used by other tools.

Scalability

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

The shared memory facilities of the shared server enable Oracle Database to support many thousands of concurrent users on a single node. For more scalability, you can use the Oracle Connection Manager to multiplex network connections.

Manageability

After being validated, you can use a PL/SQL stored subprogram in any number of applications. If its definition changes, then only the subprogram is affected, not the applications that invoke it. This simplifies maintenance and enhancement. Also, maintaining a subprogram on the Oracle Database is easier than maintaining copies on various client systems.

Object-Oriented Programming Support

PL/SQL supports object-oriented programming with:

- Object Types
- Collections

Object Types An **object type** is a user-defined composite datatype that encapsulates a data structure along with the subprograms needed to manipulate the data. The variables that form the data structure are called attributes. The subprograms that characterize the action 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 (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. You can use collections 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 on which Oracle Database runs. You can write portable program libraries and reuse them in different environments.

Security

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

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

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

Packages

A **package** is an encapsulated collection of related program objects stored together in the database. Program objects are subprograms, variables, constants, cursors, and exceptions. For information about built-in packages, see *Oracle Database PL/SQL Packages and Types Reference*.

PL/SQL Web Development Tools

Oracle Database provides built-in tools and technologies that enable you to deploy PL/SQL applications over the Web. Thus, PL/SQL serves as an alternative to Web application frameworks such as CGI.

The **PL/SQL Web Toolkit** is a set of PL/SQL packages that you can use to develop stored subprograms that can be invoked by a Web client. The PL/SQL Gateway enables an HTTP client to invoke a PL/SQL stored subprogram through `mod_plsql`, which is a plug-in to Oracle HTTP Server. This module performs the following actions:

1. Translates a URL passed by a browser client
2. Invokes an Oracle Database stored subprogram with the parameters in the URL
3. Returns output (typically HTML) to the client

See Also: Chapter 10, "Developing PL/SQL Web Applications" to learn how to use PL/SQL in Web development

Overview of Java Support Built into the Database

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

This section contains the following topics:

- Overview of Oracle JVM

- Overview of Oracle Extensions to JDBC
- Overview of Oracle SQLJ
- Overview of Oracle JPublisher
- Overview of Java Stored Subprograms
- Overview of Oracle Database Web Services
- Overview of Writing Subprograms in Java

See Also:

- *Oracle Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database JPublisher User's Guide*

Overview of Oracle JVM

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

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

This design provides the following benefits:

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

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

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

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

Overview of Oracle Extensions to JDBC

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

The JDBC standard defines four types of JDBC drivers:

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

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

You can use JDBC to do dynamic SQL. In dynamic SQL, 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 in the following sections. Oracle Database support of and extensions to various levels of the JDBC standard are described in "Oracle Database Extensions to JDBC Standards" on page 1-9.

Topics:

- JDBC Thin Driver
- JDBC OCI Driver
- JDBC Server-Side Internal Driver
- Oracle Database Extensions to JDBC Standards
- Sample JDBC 2.0 Program
- Sample Pre-2.0 JDBC Program
- JDBC in SQLJ Applications

JDBC Thin Driver

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

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

JDBC OCI Driver

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

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

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

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

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

JDBC Server-Side Internal Driver

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

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

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

Oracle Database Extensions to JDBC Standards

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

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

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

Oracle Database supports the following features from the JDBC 3.0 standard:

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

Sample JDBC 2.0 Program

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

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

Sample Pre-2.0 JDBC Program

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

The `SELECT` statement retrieves and lists the contents of the `last_name` column of the `hr.employees` table.

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

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

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

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

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

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

To use the OCI driver in this code, replace the `Connection` statement with the following, where `MyHostString` is an entry in the `tnsnames.ora` file:

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

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

JDBC in SQLJ Applications

JDBC code and SQLJ code (see "Overview of Oracle SQLJ" on page 1-11) interoperate, allowing dynamic SQL statements in JDBC to be used with both static and dynamic SQL statements in SQLJ. A SQLJ iterator class corresponds to the JDBC result set.

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

Overview of Oracle SQLJ

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

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

Note: This document uses the term **SQLJ** to refer to the Oracle SQLJ implementation, including Oracle SQLJ extensions.

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

The Oracle SQLJ translator performs the following tasks:

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

Oracle Database supports SQLJ stored subprograms and triggers that execute in the Oracle JVM. SQLJ is integrated with JDeveloper. Source-level debugging support for SQLJ is available in JDeveloper.

The following is an example of a simple SQLJ executable statement, which returns one value because `employee_id` is unique in the `employee` table:

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

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

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

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

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

Topics:

- Benefits of SQLJ
- Comparing SQLJ to JDBC
- SQLJ Stored Subprograms in the Server

Benefits of SQLJ

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

In particular, Oracle SQLJ does the following:

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

Comparing SQLJ to JDBC

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

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

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

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

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

  Following is the SQLJ equivalent:

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

The following benefits are common to SQLJ and JDBC:

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

SQLJ Stored Subprograms in the Server

SQLJ applications can be stored and executed in the server by using the following techniques:

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

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

Overview of Oracle JPublisher

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

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

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

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

Overview of Java Stored Subprograms

Java stored subprograms enable you to implement programs that run in the database server and are independent of programs that run in the middle tier. Structuring applications in this way reduces complexity and increases reuse, security, performance, and scalability.

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

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

See Also: *Oracle Database Java Developer's Guide*

Overview of Oracle Database Web Services

Web services represent a distributed computing paradigm for Java application development that is an alternative to earlier Java protocols such as JDBC. It allows application-to-application interaction through the XML and Web protocols. For example, an electronics parts vendor can provide a Web-based programmatic interface to its suppliers for inventory management. The vendor can invoke a Web service as part of a program and automatically order new stock based on the data returned.

The key technologies used in Web services are:

- Web Services Description Language (WSDL), which is a standard format for creating an XML document. WSDL describes what a web service can do, where it resides, and how to invoke it. Specifically, it describes the operations and parameters, including parameter types, provided by a Web service. In addition, a WSDL document describes the location, the transport protocol, and the invocation style for the Web service.
- Simple Object Access Protocol (SOAP) messaging, which is an XML-based message protocol used by Web services. SOAP does not prescribe a specific transport mechanism such as HTTP, FTP, SMTP, or JMS; however, most Web services accept messages that use HTTP or HTTPS.
- Universal Description, Discovery, and Integration (UDDI) business registry, which is a directory that lists Web services on the internet. The UDDI registry is often compared to a telephone directory, listing unique identifiers (white pages), business categories (yellow pages), and instructions for binding to a service protocol (green pages).

Web services can use a variety of techniques and protocols. For example:

- Dispatching can occur in a synchronous (typical) or asynchronous manner.
- You can invoke a Web service in an RPC-style operation in which arguments are sent and a response returned, or in a message style such as a one-way SOAP document exchange.
- You can use different encoding rules: literal or encoded.

You can invoke a Web service statically, when you might know everything about it beforehand, or dynamically, in which case you can discover its operations and transport endpoints while using it.

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

As a Web service provider, Oracle Database provides a disconnected and heterogeneous environment that:

- Exposes stored subprograms independently of the language in which the subprograms are written
- Exposes SQL Queries and XQuery

Overview of Writing Subprograms in Java

Subprograms (procedures and functions) are named blocks that encapsulate a sequence of statements. They are like building blocks that you can use to construct modular, maintainable applications. Write these named blocks and then define them with the `loadjava` command or SQL `CREATE FUNCTION`, `CREATE PROCEDURE`, or `CREATE PACKAGE` statements. These Java methods can accept arguments and can be called from the following:

- 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

Topics:

- Overview of Writing Database Triggers in Java
- Why Use Java for Stored Subprograms and Triggers?

Overview of Writing Database Triggers in Java

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

Why Use Java for Stored Subprograms and Triggers?

- Stored subprograms 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 subprograms run faster in the server.

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

Overview of Pro*C/C++

The Pro*C/C++ precompiler is a software tool that allows the programmer to embed SQL statements in a C or C++ source file. Pro*C/C++ reads the source file as input and outputs a C or C++ source file that replaces the embedded SQL statements with Oracle run-time 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 rerun the two steps.

Topics:

- Implementing a Pro*C/C++ Application
- Highlights of Pro*C/C++ Features

Implementing a Pro*C/C++ Application

The following is a simple code fragment from a C source file that queries the table `employees` in the schema `hr`:

```
...
#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  last_name[UNAME_LEN];
    float    salary;
    float    commission_pct;
} 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 last_name, salary, and commission_pct given the user's input
/* for employee_id. */
    EXEC SQL SELECT last_name, salary, commission_pct
        INTO :emprec INDICATOR :emprec_ind
        FROM employees
        WHERE employee_id = :emp_number;
...
```

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

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

Enter the options either in a configuration file, on the command line, or in-line inside your source code with a special statement that begins with `EXEC ORACLE`. If there are no errors found, you can 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++ gives you the freedom to design your own user interfaces and to add database access to existing applications.

Before writing your embedded SQL statements, you can test interactive versions of the SQL in SQL*Plus and then make 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 *Pro*C/C++ Precompiler Programmer's Guide*.

- You can write your application in either C or C++.
- You can write multithreaded programs if your platform supports a threads package. Concurrent connections are supported in either single-threaded or multithreaded applications.
- You can improve performance by embedding PL/SQL blocks. These blocks can invoke subprograms in Java or PL/SQL that are written by you or provided in Oracle Database packages.
- Using precompiler options, you can check the syntax and semantics of your SQL or PL/SQL statements during precompilation, as well as at run time.
- You can invoke stored PL/SQL and Java subprograms. Modules written in COBOL or in C can be invoked from Pro*C/C++. External C subprograms in shared libraries can be invoked by your program.
- You can conditionally precompile sections of your code so that they can execute in different environments.
- You can use arrays, or structures, or arrays of structures as host and indicator variables in your code to improve performance.
- You can deal with errors and warnings so that data integrity is guaranteed. As a programmer, you control how errors are handled.
- Your program can convert between internal datatypes and C language datatypes.
- The Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI), lower-level C and C++ interfaces, are available for use in your precompiler source.
- Pro*C/C++ supports dynamic SQL, a technique that allows users to input variable values and statement syntax.

- Pro*C/C++ can use special SQL statements to manipulate tables containing user-defined object types. An Object Type Translator (OTT) maps the object types and named collection types in your database to structures and headers that you include in your source.
- Two kinds of collection types, nested tables and `VARRAY`, are supported with a set of SQL statements that allow a high degree of control over data.
- Large Objects are accessed by another set of SQL statements.
- A new ANSI SQL standard for dynamic SQL is supported for new applications, so that you can execute SQL statements with a varying number of host variables. An older technique for dynamic SQL is still usable by pre-existing applications.
- Globalization support lets you use multibyte characters and UCS2 Unicode data.
- Using scrollable cursors, you can move backward and forward through a result set. For example, you can fetch the last row of the result set, or jump forward or backward to an absolute or relative position within the result set.
- A **connection pool** is a group of physical connections to a database that can be shared by several named connections. Enabling the connection pool option can help to optimize the performance of Pro*C/C++ application. The connection pool option is not enabled by default.

Overview of Pro*COBOL

The Pro*COBOL precompiler is a software tool that allows the programmer to embed SQL statements in a COBOL source code file. Pro*COBOL reads the source file as input and outputs a COBOL source file that replaces the embedded SQL statements with Oracle Database run-time 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 rerun the two steps.

Topics:

- Implementing a Pro*COBOL Application
- Highlights of Pro*COBOL Features

Implementing a Pro*COBOL Application

Here is a simple code fragment from a source file that queries the table `employees` in the schema `hr`:

```
...
 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 last_name, salary, commission_pct
         INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
         FROM employees
         WHERE employee_id = :EMP-NUMBER
     END-EXEC.
...
```

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

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

Enter the options in a configuration file, on the command line, or in-line inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can 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 gives 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 can test interactive versions of the SQL in SQL*Plus and then make minor changes to start testing your embedded SQL application.

Highlights of Pro*COBOL Features

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

- You can invoke stored PL/SQL or Java subprograms. You can improve performance by embedding PL/SQL blocks. These blocks can invoke PL/SQL subprograms written by you or provided in Oracle Database packages.
- Precompiler options enable 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 run time.
- You can conditionally precompile sections of your code so that they can execute in different environments.
- Use tables, or group items, or tables of group items as host and indicator variables in your code to improve performance.
- You can program how errors and warnings are handled, so that data integrity is guaranteed.

- Pro*COBOL supports dynamic SQL, a technique that allows users to input variable values and statement syntax.

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

Overview of OCI and OCCI

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

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

OCI lets you manipulate data and schemas in a database using a host programming language, such as C. OCCI is an object-oriented interface suitable for use with C++. These APIs provide a library of standard database access and retrieval functions in the form of a dynamic run-time library (OCILIB) that can be linked in an application at run time. This eliminates the need to embed SQL or PL/SQL within 3GL programs.

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

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

Topics:

- Advantages of OCI and OCCI
- OCI and OCCI Functions
- Procedural and Nonprocedural Elements of OCI and OCCI Applications
- Building an OCI or OCCI Application

Advantages of OCI and OCCI

OCI and OCCI provide significant advantages over other methods of accessing Oracle Database:

- More fine-grained control over all aspects of the application design.
- High degree of control over program execution.

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

OCI and OCCI Functions

Both OCI and OCCI have four kinds of functions:

Kind of Function	Purpose
Relational	To manage database access and process SQL statements
Navigational	To manipulate objects retrieved from the database
Database mapping and manipulation	To manipulate data attributes of Oracle types
External subprogram	To write C callbacks from PL/SQL

Procedural and Nonprocedural Elements of OCI and OCCI Applications

OCI and OCCI enable you to develop applications that combine the nonprocedural data access power of SQL with the procedural capabilities of most programming languages, including C and C++. procedural and nonprocedural languages have these characteristics:

- In a nonprocedural language program, the set of data to be operated on is specified, but what operations are performed and how the operations are to be carried out is not specified. The nonprocedural 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 nonprocedural and procedural language elements in an OCI or OCCI program provides easy access to Oracle Database in a structured programming environment.

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

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

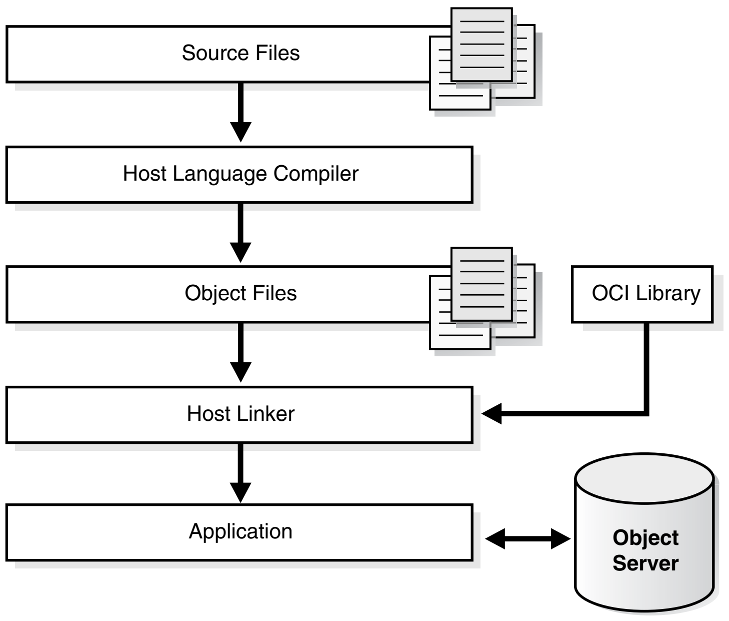
In the preceding SQL statement, `:empnumber` is a placeholder for a value to be supplied by the application.

Alternatively, you can 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. OCI and OCCI also provide facilities for accessing and manipulating objects in Oracle Database.

Building an OCI or OCCI Application

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

Figure 1–1 The OCI or OCCI Development Process



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

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

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

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

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

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

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

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

class Example
{
  OracleConnection con;

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

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

  static void Main()
  {
    Example example = new Example();
    example.Connect();
    example.Close();
  }
}
```

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

Overview of Oracle Objects for OLE (OO4O)

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

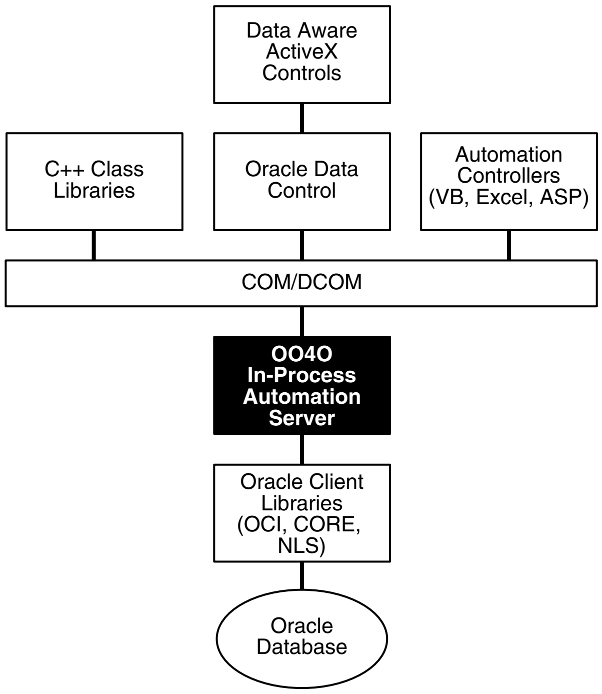
See the OO4O online help for detailed information about using OO4O.

Oracle Objects for OLE consists of the following software layers:

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

Figure 1–2 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

This illustration shows the OO4O software components (layers).

The first layer contains the Data Aware ActiveX Controls.

The second layer consists of C++ Class Libraries and Oracle Data Control Automation Controllers (VB< Excel, ASP).

The third layer contains the COM/DCOM.

The fourth layer contains the OO40 In-Process Automation Server.

The fifth layer contains the Oracle Client Libraries (OCI, CORE, NLS).

The last layer contains the Oracle Database.

****

Topics:

- OO4O Automation Server
- OO4O Object Model
- Support for Oracle LOB and Object Datatypes

- Oracle Data Control
- Oracle Objects for OLE C++ Class Library
- Additional Sources of Information

OO4O Automation Server

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

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

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

OO4O provides key features for accessing Oracle Database efficiently and easily in environments ranging from the typical two-tier client/server applications, such as those developed in Visual Basic or Excel, to application servers deployed in multitiered application server environments such as Web server applications in Microsoft Internet Information Server (IIS) or Microsoft Transaction Server (MTS).

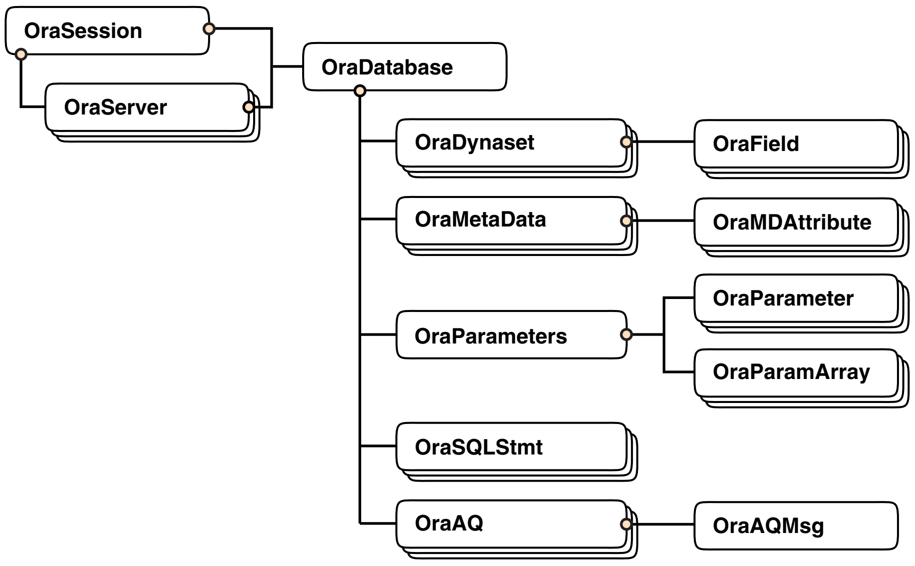
Features include:

- Support for execution of PL/SQL and Java stored subprograms, and PL/SQL anonymous blocks. This includes support for Oracle datatypes used as parameters to stored subprograms, including PL/SQL cursors. See "Support for Oracle LOB and Object Datatypes" on page 1-29.
- 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.

Figure 1–3 Objects and Their Relations



This figure shows OO4O objects and their relations. The relations are as follows:

- From OraServer to OraSession
- From OraDatabase to both OraServer and OraSession
- From each of OraDynaset, OraMetaData, OraParameters, OraSQLStmt, and OraAQ to OraDatabase
- From multiple OraField objects to OraDynaset
- From multiple OraMDAttribute objects to OraMetaData
- From multiple OraParameter and OraParamArray objects to OraParameters
- from OraAQMsg to OraAQ

****

Topics:

- OraSession
- OraServer
- OraDatabase
- OraDynaset
- OraField
- OraMetaData and OraMDAttribute
- OraParameter and OraParameters
- OraParamArray
- OraSQLStmt
- OraAQ
- OraAQMsg
- OraAQAgent

OraSession

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

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

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

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

OraServer

OraServer represents a physical network connection to Oracle Database.

The `OraServer` interface is introduced to expose the connection-multiplexing feature provided in the Oracle Call Interface. After an `OraServer` object is created, multiple user sessions (`OraDatabase`) can be attached to it by calling the `OpenDatabase` method. This feature is particularly useful for application components, such as Internet Information Server (IIS), that use Oracle Objects for OLE in n-tier distributed environments.

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

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

OraDatabase

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

In releases prior to Oracle8*i*, an `OraDatabase` object is created by calling 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, calling this method results in implicit creation of an `OraServer` object.

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

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

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

For example:

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

OraDynaset

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

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

OraField

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

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

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

OraMetaData and OraMDAttribute

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

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

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

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

OraParameter and OraParameters

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

`OraParameter` objects are created, accessed, and removed indirectly through the `OraParameters` collection of an `OraDatabase` object. Each parameter has an identifying name and an associated value. You can automatically bind a parameter to SQL and PL/SQL statements of other objects (as noted in the object descriptions), by using the parameter name as a placeholder in the SQL or PL/SQL statement. Such use of parameters can simplify dynamic queries and increase program performance.

OraParamArray

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

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

OraSQLStmt

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

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

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

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

OraAQ

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

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

OraAQMsg

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

For a detailed description of Oracle Advanced Queuing, see *Oracle Streams Advanced Queuing User's Guide.*

OraAQAgent

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

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

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

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

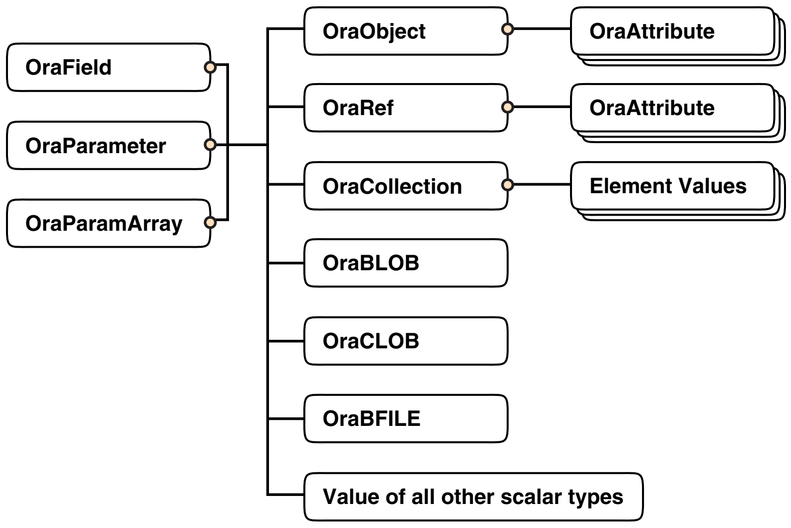
```
Set agent = qMsg.AddRecipient(name, address, protocol).
```

Support for Oracle LOB and Object Datatypes

Oracle Objects for OLE (OO4O) provides full support for accessing and manipulating instances of object datatypes and LOBs in Oracle Database. Figure 1–4 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 subprograms. All instances are mapped to COM Automation Interfaces that provide methods for dynamic attribute access and manipulation.

Figure 1–4 Supported Oracle Datatypes



Topics:

- OraBLOB and OraCLOB
- OraBFILE

OraBLOB and OraCLOB

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

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

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

OraBFILE

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

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

Oracle Data Control

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

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

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

Oracle Objects for OLE C++ Class Library

Oracle Objects for OLE (OO4O) 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 see the online help provided with the OO4O product:

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

For examples of how to use Oracle Objects for OLE, see the samples in the `ORACLE_HOME\OO4O` directory of the Oracle Database installation and in the following:

- *Oracle Database SecureFiles and Large Objects Developer's Guide*
- *Oracle Streams Advanced Queuing User's Guide*

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 subprograms can both be invoked from code written in any of the languages in this chapter. Stored subprograms include triggers and object type methods.
- External subprograms written in C can be invoked from OCI, Java, PL/SQL or SQL. The external subprogram 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.

Topics:

- Choosing a Precompiler or OCI
- Choosing PL/SQL or Java

Choosing a Precompiler or OCI

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

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

- OCI provides more detailed control over multiplexing and migrating sessions.
- OCI provides dynamic bind and define using callbacks that can be used for any arbitrary structure, including lists.
- OCI has many calls to handle metadata.
- OCI allows asynchronous event notifications to be received by a client application. It provides a means for clients to generate notifications for propagation to other clients.
- OCI allows DML statements to use arrays to complete as many iterations as possible before returning any error messages.
- OCI calls for special purposes include Advanced Queuing, globalization support, Data Cartridges, and support of the date and time datatypes.
- OCI calls can be embedded in a Pro*C/C++ application.

Choosing PL/SQL or Java

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

PL/SQL and Java interoperate in the server. You can execute a PL/SQL package from Java or wrap a PL/SQL class with a Java wrapper so that it can be invoked from distributed CORBA and EJB clients. Table 1–1 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 subprogram.
`DBMS_LOCK`	Call with SQLJ or JDBC.
`DBMS_MAIL`	Use JavaMail.
`DBMS_OUTPUT`	Use subclass `oracle.aurora.rdbms.OracleDBMSOutputStream` or Java stored subprogram `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.

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

- PL/SQL is optimized for database access

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

- PL/SQL is integrated with the database

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

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

- Both Java and PL/SQL have object-oriented features

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

- Java is used for open distributed applications

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

SQL for Application Developers

This part presents information that application developers need about Structured Query Language (SQL), which is used to manage information in an Oracle Database.

Chapters:

See Also: *Oracle Database SQL Language Reference* for a complete description of SQL

2

SQL Processing for Application Developers

This chapter explains what application developers must know about how Oracle Database processes SQL statements. Before reading this chapter, read the basic information about SQL processing in *Oracle Database Concepts*.

Topics:

Grouping Operations into Transactions

Topics:

Deciding How to Group Operations in Transactions

In general, deciding how to group operations in transactions is the concern of application designers who use the programming interfaces to Oracle Database. When deciding how to group transactions:

- Define transactions such that work is accomplished in logical units and data remains consistent.
- Ensure that data in all referenced tables is in a consistent state before the transaction begins and after it ends.

- Ensure that each transaction consists only of the SQL statements or PL/SQL blocks that comprise one consistent change to the data.

For example, suppose that you write a Web application that enables users to transfer funds between accounts. The transaction must include the debit to one account, which is executed by one SQL statement, and the credit to another account, which is executed by a second SQL statement. Both statements must fail or succeed together as a unit of work; the credit must not be committed without the debit. Other unrelated actions, such as a new deposit to one account, must not be included in the same transaction.

Improving Transaction Performance

As an application developer, you must consider whether you can improve performance. Consider the following performance enhancements when designing and writing your application:

- Use the `SET TRANSACTION` statement with the `USE ROLLBACK SEGMENT` clause to explicitly assign a transaction to a rollback segment. This technique can eliminate the need to allocate additional extents dynamically, which can reduce system performance. This clause is valid only if you use rollback segments for undo. If you use automatic undo management, then Oracle Database ignores this clause.
- Establish standards for writing SQL statements so that you can take advantage of shared SQL areas. Oracle Database recognizes identical SQL statements and allows them to share memory areas. This reduces memory usage on the database server and increases system throughput.
- Use the `ANALYZE` statement to collect statistics that can be used by Oracle Database to implement a cost-based approach to SQL statement optimization. You can supply additional "hints" to the optimizer as needed.
- Invoke 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. Specify which 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 "Invoking Stored PL/SQL Functions from SQL Statements" on page 7-32.
- Create explicit cursors when writing a PL/SQL application.
- Reduce frequency of parsing and improve performance in precompiler programs by increasing the number of cursors with `MAX_OPEN_CURSORS`.
- Use the `SET TRANSACTION` statement with the `ISOLATION LEVEL` set to `SERIALIZABLE` to get ANSI/ISO serializable transactions.

See Also:

- "How Serializable Transactions Interact" on page 2-17
- "Using Cursors" on page 2-7
- *Oracle Database Concepts* for more information about transaction management

Committing Transactions

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

```
COMMIT WORK;
COMMIT;
```

The `COMMIT` statements lets you include the `COMMENT` parameter along with a comment 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';
```

Managing Commit Redo Action

When a transaction updates the database, it generates a redo entry corresponding to this update. Oracle Database buffers this redo in memory until the completion of the transaction. When the transaction commits, the log writer process (LGWR) writes redo for the commit, along with the accumulated redo of all changes in the transaction, to disk. By default, Oracle Database writes the redo to disk before the call returns to the client. This action introduces a latency in the commit because the application must wait for the redo to be persisted on disk.

Suppose that you are writing an application that requires very high transaction throughput. If you are willing to trade commit durability for lower commit latency, then you can change the default `COMMIT` options so that the application does not need to wait for Oracle Database to write data to the online redo logs.

Oracle Database enables you to change the handling of commit redo depending on the needs of your application. You can change the commit action in the following locations:

- `COMMIT_WRITE` initialization parameter at the system or session level
- `COMMIT` statement

The options in the `COMMIT` statement override the current settings in the initialization parameter. Table 2–1 describes redo persistence options that you can set in either location.

Caution: With the `NOWAIT` option of `COMMIT` or `COMMIT_WRITE`, a failure that occurs after the commit message is received, but before the redo log record(s) are written, can falsely indicate to a transaction that its changes are persistent.

Table 2–1 Options of COMMIT Statement and COMMIT_WRITE Initialization Parameter

Option	Effect
`WAIT` (default)	Ensures that the commit returns only after the corresponding redo information is persistent in the online redo log. When the client receives a successful return from this `COMMIT` statement, the transaction has been committed to durable media. A failure that occurs after a successful write to the log might prevent the success message from returning to the client, in which case the client cannot tell whether or not the transaction committed.
`NOWAIT`	The commit returns to the client whether or not the write to the redo log has completed. This behavior can increase transaction throughput.

Table 2–1 (Cont.) Options of COMMIT Statement and COMMIT_WRITE Initialization

Option	Effect
`BATCH`	The redo information is buffered to the redo log, along with other concurrently executing transactions. When sufficient redo information is collected, a disk write to the redo log is initiated. This behavior is called **group commit**, as redo information for multiple transactions is written to the log in a single I/O operation.
`IMMEDIATE` (default)	LGWR writes the transaction's redo information to the log. Because this operation option forces a disk I/O, it can reduce transaction throughput.

The following example shows how to set the commit action to `BATCH` and `NOWAIT` in the initialization parameter file:

```
COMMIT_WRITE = BATCH, NOWAIT
```

You can change the commit action at the system level by executing `ALTER SYSTEM` as in the following example:

```
ALTER SYSTEM SET COMMIT_WRITE = BATCH, NOWAIT
```

After the initialization parameter is set, a `COMMIT` statement with no options conforms to the options specified in the parameter. Alternatively, you can override the current initialization parameter setting by specifying options directly on the `COMMIT` statement as in the following example:

```
COMMIT WRITE BATCH NOWAIT
```

In either case, your application specifies that log writer does not have to write the redo for the commit immediately to the online redo logs and need not wait for confirmation that the redo was written to disk.

Note: You cannot change the default `IMMEDIATE` and `WAIT` action for distributed transactions.

If your application uses OCI, then you can modify redo action by setting the following flags in the `OCITransCommit` function within your application:

Caution: There is a potential for silent transaction loss when you use `OCI_TRANS_WRITENOWAIT`. Transaction loss occurs silently with shutdown abort, startup force, and any instance or node failure. On a RAC system asynchronously committed changes might not be immediately available to read on other instances.

- `OCI_TRANS_WRITEBATCH`
- `OCI_TRANS_WRITENOWAIT`
- `OCI_TRANS_WRITEIMMED`
- `OCI_TRANS_WRITEWAIT`

The specification of the `NOWAIT` and `BATCH` options allows a small window of vulnerability in which Oracle Database can roll back a transaction that your application view as committed. Your application must be able to tolerate the following scenarios:

- The database host fails, which causes the database to lose redo that was buffered but not yet written to the online redo logs.
- A file I/O problem prevents log writer from writing buffered redo to disk. If the redo logs are not multiplexed, then the commit is lost.

See Also:

- *Oracle Database SQL Language Reference* for information on the `COMMIT` statement
- *Oracle Call Interface Programmer's Guide* for information about the `OCITransCommit` function

Rolling Back Transactions

To roll back an entire transaction, or to roll back part of a transaction to a savepoint, use the `ROLLBACK` statement. For example, either of the following statements rolls back the entire current transaction:

```
ROLLBACK WORK;
ROLLBACK;
```

The `WORK` option of the `ROLLBACK` statement has no function.

To roll back to a savepoint defined in the current transaction, use the `TO` option of the `ROLLBACK` statement. For example, either of the following statements rolls back the current transaction to the savepoint named `POINT1`:

```
SAVEPOINT Point1;
...
ROLLBACK TO SAVEPOINT Point1;
ROLLBACK TO Point1;
```

Defining Transaction Savepoints

To define a savepoint in a transaction, use the `SAVEPOINT` statement. 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 was specified since the last commit or rollback.

Table 2–2 shows a series of SQL statements that illustrates the use of `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements within a transaction.

Table 2–2 Use of COMMIT, SAVEPOINT, and ROLLBACK

SQL Statement	Results
`SAVEPOINT a;`	First savepoint of this transaction
`DELETE...;`	First DML statement of this transaction
`SAVEPOINT b;`	Second savepoint of this transaction
`INSERT INTO...;`	Second DML statement of this transaction
`SAVEPOINT c;`	Third savepoint of this transaction

Table 2–2 (Cont.) Use of COMMIT, SAVEPOINT, and ROLLBACK

SQL Statement	Results
`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.

Ensuring Repeatable Reads with Read-Only Transactions

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

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

Long-running queries sometimes fail because undo information required for consistent read (CR) operations is no longer available. This happens when committed undo blocks are overwritten by active transactions. Automatic undo management provides a way to explicitly control when undo space can be reused; that is, how long undo information is retained. Your database administrator can specify a retention period by using the parameter `UNDO_RETENTION`.

See Also: *Oracle Database Administrator's Guide* for information on long-running queries and resumable space allocation

For example, if `UNDO_RETENTION` is set to 30 minutes, then all committed undo information in the system is retained for at least 30 minutes. This ensures that all queries running for 30 minutes or less, under usual circumstances, do not encounter the OER error "snapshot too old."

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

```
SET TRANSACTION READ ONLY;
```

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

Using Cursors

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 subprogram (procedure or function). Cursor variables enable you to pass cursors as parameters in your 3GL application. Cursor variables are described in *Oracle Database PL/SQL Language Reference.*

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

Topics:

- How Many Cursors Can a Session Have?
- Using a Cursor to Re-Execute a Statement
- Closing a Cursor
- Canceling a Cursor

How Many Cursors Can a Session Have?

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

- Each cursor requires virtual memory, so a session's total number of cursors is limited by the memory available to that process.
- A systemwide limit of cursors for each session is set by the initialization parameter named OPEN_CURSORS found in the parameter file (such as INIT.ORA).

See Also: *Oracle Database Reference* for more information about OPEN_CURSORS

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

Using a Cursor to Re-Execute a Statement

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 was associated with that cursor. The statement can be reexecuted without including the parse stage.

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

To understand the performance characteristics of a cursor, a DBA can retrieve the text of the query represented by the cursor using the `V$SQL` dynamic performance view. Because the results of `EXPLAIN PLAN` on the original query might differ from the way the query is actually processed, a DBA can get more precise information by examining the following dynamic performance views:

View	Description
`V$SQL_PLAN`	Execution plan information for each child cursor loaded in the library cache.
`V$SQL_STATISTICS`	Execution statistics at the row source level for each child cursor.
`V$SQL_STATISTICS_ALL`	Memory usage statistics for row sources that use SQL memory (sort or hash-join). This view concatenates information in `V$SQL_PLAN` with execution statistics from `V$SQL_PLAN_STATISTICS` and `V$SQL_WORKAREA`.

See Also: *Oracle Database Reference* for details of the preceding dynamic performance views

Closing a Cursor

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.)

Canceling a Cursor

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/C++ or PL/SQL.

See Also: *Oracle Call Interface Programmer's Guide* for information about cancelling a cursor with the `OCIStmtFetch2` statement

Locking Tables Explicitly

Oracle Database always performs necessary locking to ensure data concurrency, integrity, and statement-level read consistency. You can override these default locking mechanisms. For example, you might want to override the default locking of Oracle Database if:

- You want transaction-level read consistency or "repeatable reads"—where transactions query a consistent set of data for the duration of the transaction, knowing that the data was not changed by any other transactions. This level of consistency can be achieved by using explicit locking, read-only transactions, serializable transactions, or overriding default locking for the system.
- A transaction requires exclusive access to a resource. To proceed with its statements, the transaction with exclusive access to a resource does not have to wait for other transactions to complete.

The automatic locking mechanisms can be overridden at the transaction level. Transactions including the following SQL statements override Oracle Database's default locking:

- `LOCK TABLE`
- `SELECT`, including the `FOR UPDATE` clause
- `SET TRANSACTION` with the `READ ONLY` or `ISOLATION LEVEL SERIALIZABLE` options

Locks acquired by these statements are released after the transaction is committed or rolled back.

The following sections describe each option available for overriding the default locking of Oracle Database. The initialization parameter `DML_LOCKS` determines the maximum number of DML locks allowed.

See Also: *Oracle Database Reference* for more information about `DML_LOCKS`

Although the default value is usually enough, you might need to increase it if you use additional manual locks.

Caution: If you override the default locking of Oracle Database at any level, be sure that the overriding locking subprograms operate correctly: Ensure that data integrity is guaranteed, data concurrency is acceptable, and deadlocks are either impossible or appropriately handled.

Topics:

- Privileges Required
- Choosing a Locking Strategy
- Letting Oracle Database Control Table Locking
- Explicitly Acquiring Row Locks

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.

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.

In the LOCK TABLE statement, you can also indicate how long you want to wait for the table lock:

- If you do not want to wait, specify either NOWAIT or WAIT 0.

  You acquire the table lock only if it is immediately available; otherwise, an error notifies you that the lock is not available at this time.

- If you want to wait up to *n* seconds to acquire the table lock, specify WAIT *n*, where *n* is greater than 0 and less than or equal to 100000.

  If the table lock is still unavailable after *n* seconds, an error notifies you that the lock is not available at this time.

- If you want to wait indefinitely to acquire the lock, specify neither NOWAIT nor WAIT.

  The database waits indefinitely until the table is available, locks it, and returns control to you. When the database is executing DDL statements concurrently with DML statements, a timeout or deadlock can sometimes result. The database detects such timeouts and deadlocks and returns an error.

For the syntax of the LOCK TABLE statement, see *Oracle Database SQL Language Reference*.

Topics:

- When to Lock with ROW SHARE MODE and ROW EXCLUSIVE MODE
- When to Lock with SHARE MODE
- When to Lock with SHARE ROW EXCLUSIVE MODE
- When to Lock with EXCLUSIVE MODE

When to Lock with ROW SHARE MODE and ROW EXCLUSIVE MODE

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

ROW SHARE and ROW EXCLUSIVE table locks offer the highest degree of concurrency. You might use these locks if:

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

- Your transaction needs to prevent a table from being altered or dropped before the table can be modified later in your transaction.

When to Lock with SHARE MODE

```
LOCK TABLE Emp_tab IN SHARE MODE;
```

`SHARE` table locks are rather restrictive data locks. You might use these locks if:

- Your transaction only queries the table, and requires a consistent set of the table data for the duration of the transaction.
- You can hold up other transactions that try to update the locked table, until all transactions that hold `SHARE` locks on the table either commit or roll back.
- Other transactions might acquire concurrent `SHARE` table locks on the same table, also allowing them the option of transaction-level read consistency.

Caution: Your transaction might or might 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 might need to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE dept_tab(
   deptno NUMBER(2) NOT NULL,
   dname VARCHAR2(14),
   loc VARCHAR2(13));

CREATE TABLE emp_tab (
   empno NUMBER(4) NOT NULL,
   ename VARCHAR2(10),
   job VARCHAR2(9),
   mgr NUMBER(4),
   hiredate DATE,
   sal NUMBER(7,2),
   comm NUMBER(7,2),
   deptno NUMBER(2));

CREATE TABLE Budget_tab (
   totsal NUMBER(7,2),
   deptno NUMBER(2) NOT NULL);
```

```
LOCK TABLE Dept_tab IN SHARE MODE;
UPDATE Emp_tab
    SET sal = sal * 1.1
    WHERE deptno IN
      (SELECT deptno FROM Dept_tab WHERE loc = 'DALLAS');
UPDATE Budget_tab
    SET Totsal = Totsal * 1.1
    WHERE Deptno IN
      (SELECT Deptno FROM Dept_tab WHERE Loc = 'DALLAS');

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

When to Lock with SHARE ROW EXCLUSIVE MODE

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

You might use a `SHARE ROW EXCLUSIVE` table lock if:

- Your transaction requires both transaction-level read consistency for the specified table and the ability to update the locked table.
- You do not care if other transactions acquire explicit row locks (using `SELECT FOR UPDATE`), which might make `UPDATE` and `INSERT` statements in the locking transaction wait and might cause deadlocks.
- You only want a single transaction to have this action.

When to Lock with 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.

Letting Oracle Database Control Table Locking

Letting Oracle Database control table locking means your application needs less programming logic, but also has less control, than if you manage the table locks yourself.

Issuing the statement `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` or `ALTER SESSION ISOLATION LEVEL SERIALIZABLE` preserves ANSI serializability without changing the underlying locking protocol. This technique allows concurrent access to the table while providing ANSI serializability. Getting table locks greatly reduces concurrency.

See Also:

- *Oracle Database SQL Language Reference* for information on the `SET TRANSACTION` statement
- *Oracle Database SQL Language Reference* for information on the `ALTER SESSION` statements

Change the settings for these parameters only when an instance is shut down. If multiple instances are accessing a single database, then all instances must use the same setting for these parameters.

Explicitly Acquiring Row Locks

You can override default locking with a `SELECT` statement that includes the `FOR UPDATE` clause. This statement acquires exclusive row locks for selected rows (as an `UPDATE` statement does), in anticipation of updating the selected rows in a subsequent statement.

You can use a `SELECT FOR UPDATE` statement to lock a row without actually changing it. For example, several triggers in *Oracle Database PL/SQL Language Reference* show how to implement referential integrity. In the `EMP_DEPT_CHECK` trigger, 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 is violated.

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

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

Each row in the return set of a `SELECT FOR UPDATE` statement is locked individually; the `SELECT FOR UPDATE` statement waits until the other transaction releases the conflicting row lock. If a `SELECT FOR UPDATE` statement locks many rows in a table, and if the table experiences a lot of update activity, it might be faster to acquire an EXCLUSIVE table lock instead.

Note: The return set for a SELECT FOR UPDATE might change while the query is running; for example, if columns selected by the query are updated or rows are deleted after the query started. When this happens, SELECT FOR UPDATE acquires locks on the rows that did not change, gets a new read-consistent snapshot of the table using these locks, and then restarts the query to acquire the remaining locks.

This can cause a deadlock between sessions querying the table concurrently with DML operations when rows are locked in a nonsequential order. To prevent such deadlocks, design your application so that any concurrent DML on the table does not affect the return set of the query. If this is not feasible, you might want to serialize queries in your application.

By default, the transaction waits until the requested row lock is acquired. If you are not willing to wait to acquire the row lock, use either the NOWAIT clause of the LOCK TABLE statement (see "Choosing a Locking Strategy" on page 2-10) or the SKIP LOCKED clause of the SELECT FOR UPDATE statement.

If you can lock some of the requested rows, but not all of them, the SKIP LOCKED option skips the rows that you cannot lock and locks the rows that you can lock.

See Also: *Oracle Database SQL Language Reference* for information on the SELECT FOR UPDATE statement and an example of the SKIP LOCKED clause

Using Oracle Lock Management Services

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

See Also: *Oracle Database PL/SQL Packages and Types Reference* for detailed information on the DBMS_LOCK package

Topics:

- When to Use User Locks
- Example of a User Lock
- Viewing and Monitoring Locks

When to Use User Locks

User locks can help to:

- Provide exclusive access to a device, such as a terminal
- Provide application-level enforcement of read locks

- Detect when a lock is released and cleanup after the application
- Synchronize applications and enforce sequential processing

Example of a User Lock

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

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

Viewing and Monitoring Locks

Table 2–5 describes the Oracle Database facilities that display locking information for ongoing transactions within an instance.

Table 2–3 Ways to Display Locking Information

Tool	Description
Oracle Enterprise Manager 10g Database Control	From the Additional Monitoring Links section of the Database Performance page, click Database Locks to display user blocks, blocking locks, or the complete list of all database locks. See *Oracle Database 2 Day DBA* for more information.
`UTLLOCKT.SQL`	The `UTLLOCKT.SQL` script displays a simple character lock wait-for graph in tree structured fashion. Using any *ad hoc* SQL tool (such as SQL*Plus) to execute the script, it prints the sessions in the system that are waiting for locks and the corresponding blocking locks. The location of this script file is operating system dependent. (You must have run the `CATBLOCK.SQL` script before using `UTLLOCKT.SQL`.)

Using Serializable Transactions for Concurrency Control

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

If a transaction A attempts to update or delete a row that has been locked by another transaction B (by way of a DML or `SELECT FOR UPDATE` statement), then A's DML statement 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 make only the database changes that they could make if the transactions ran one after the other.

Figure 2–1 shows a serializable transaction (B) interacting with another transaction (A).

The ANSI/ISO SQL standard SQL92 defines three possible kinds of transaction interaction, and four levels of isolation that provide increasing protection against these interactions. These interactions and isolation levels are summarized in Table 2–4.

Table 2–4 Summary of ANSI Isolation Levels

Isolation Level	Dirty Read[1]	Unrepeatable Read[2]	Phantom Read[3]
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	Not possible	Possible	Possible
`REPEATABLE READ`	Not possible	Not possible	Possible
`SERIALIZABLE`	Not possible	Not possible	Not possible

[1] A transaction can read uncommitted data changed by another transaction.

[2] A transaction rereads data committed by another transaction and sees the new data.

[3] A transaction can execute a query again, and discover new rows inserted by another committed transaction.

The action of Oracle Database with respect to these isolation levels is summarized in Table 2–5.

Table 2–5 ANSI Isolation Levels and Oracle Database

Isolation Level	Description
`READ UNCOMMITTED`	Oracle Database never permits "dirty reads." Although some other database products use this undesirable technique to improve thoughput, it is not required for high throughput with Oracle Database.
`READ COMMITTED`	Oracle Database meets the `READ COMMITTED` isolation standard. This is the default mode for all Oracle Database applications. Because an Oracle Database query only sees data that was committed at the beginning of the query (the snapshot time), Oracle Database actually offers more consistency than is required by the ANSI/ISO SQL92 standards for `READ COMMITTED` isolation.
`REPEATABLE READ`	Oracle Database does not normally support this isolation level, except as provided by `SERIALIZABLE`.

Table 2–5 (Cont.) ANSI Isolation Levels and Oracle Database

Isolation Level	**Description**
`SERIALIZABLE`	Oracle Database does not normally support this isolation level, except as provided by `SERIALIZABLE`.

Topics:

- How Serializable Transactions Interact
- Setting the Isolation Level of a Serializable Transaction
- Referential Integrity and Serializable Transactions
- READ COMMITTED and SERIALIZABLE Isolation
- Application Tips for Transactions

How Serializable Transactions Interact

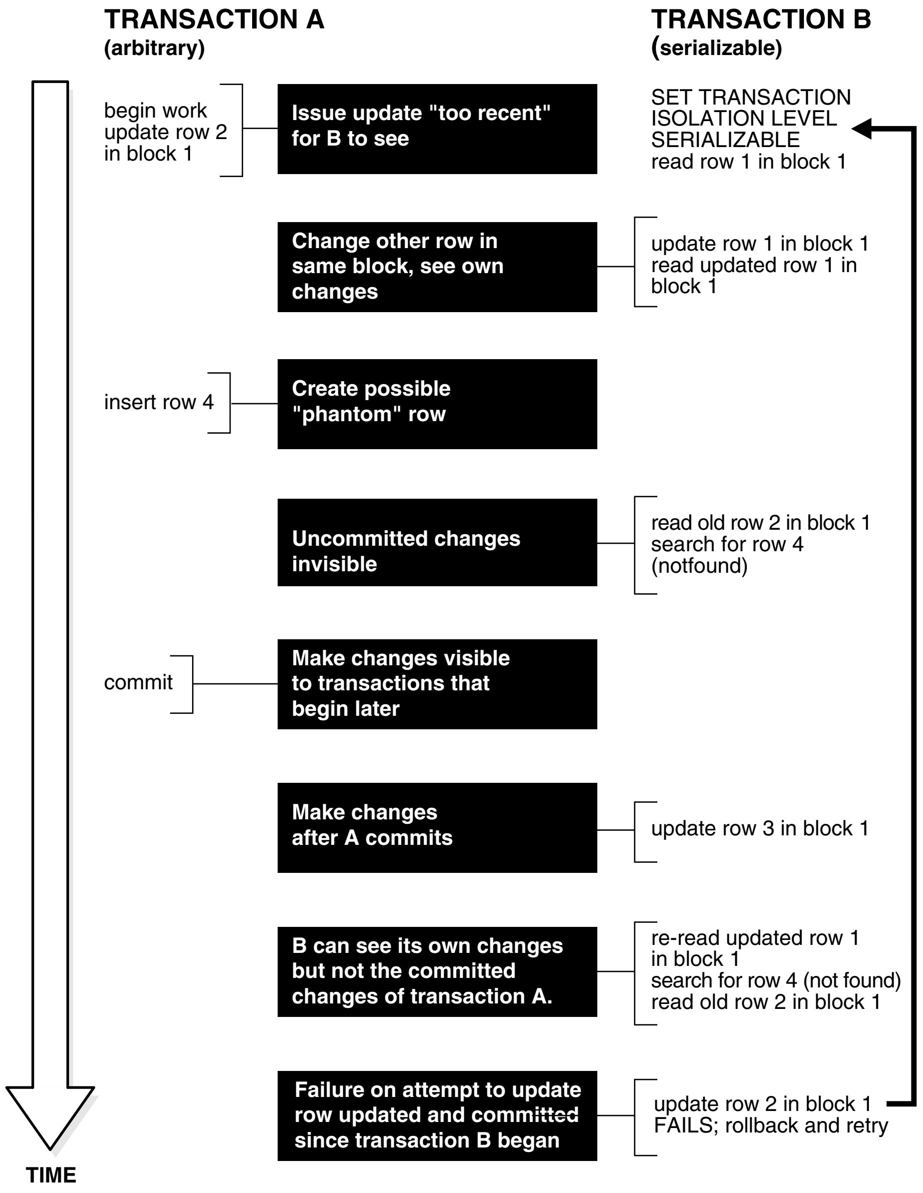
Figure 2–1 on page 2-18 shows how a serializable transaction (Transaction B) interacts with another transaction (A, which can be either `SERIALIZABLE` or `READ COMMITTED`).

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

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

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

Figure 2–1 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 and committed since transaction B began
update row 2 in block 1
FAILS; rollback and retry
TIME

Setting the Isolation Level of a Serializable Transaction

You can change the isolation level of a transaction using the `ISOLATION LEVEL` clause of the `SET TRANSACTION` statement, which must be the first statement issued in a transaction.

Use the `ALTER SESSION` statement to set the transaction isolation level on a session-wide basis.

See Also:

- *Oracle Database SQL Language Reference* for the syntax of the `ALTER SESSION` statement
- *Oracle Database SQL Language Reference* for the syntax of the `SET TRANSACTION` statement

Oracle Database stores control information in each data block to manage access by concurrent transactions. Therefore, if you set the transaction isolation level to `SERIALIZABLE`, then you must use the `ALTER TABLE` statement to set `INITRANS` to at least 3. This parameter causes Oracle Database to allocate sufficient storage in each block to record the history of recent transactions that accessed the block. Use higher values for tables that will undergo many transactions updating the same blocks.

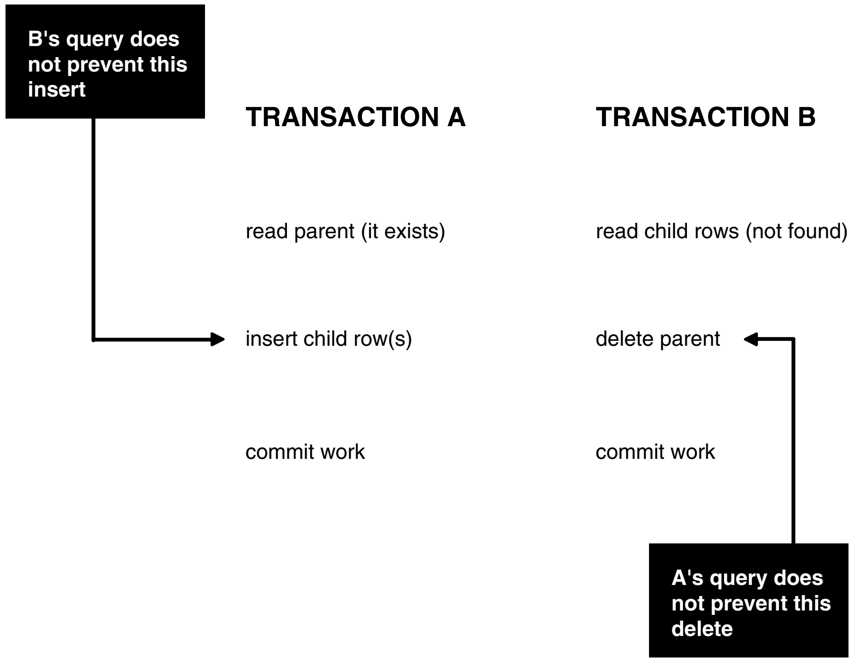
Referential Integrity and Serializable Transactions

Because Oracle Database does not use read locks, even in `SERIALIZABLE` transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level must 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: Examples in this section apply to both `READ COMMITTED` and `SERIALIZABLE` transactions.

Figure 2–2 on page 2-20 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 2–2 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.

Fortunately, it is straightforward in Oracle Database to prevent the anomaly described:

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

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

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

READ COMMITTED and SERIALIZABLE Isolation

Oracle Database gives you a choice of two transaction isolation levels with different characteristics. Both the `READ COMMITTED` and `SERIALIZABLE` isolation levels provide a high degree of consistency and concurrency. Both levels reduce contention, and are designed for deploying real-world applications. The rest of this section compares the two isolation modes and provides information helpful in choosing between them.

Topics:

- Transaction Set Consistency
- Comparison of READ COMMITTED and SERIALIZABLE Transactions
- Choosing an Isolation Level for Transactions

Transaction Set Consistency

A useful way to describe the `READ COMMITTED` and `SERIALIZABLE` isolation levels in Oracle Database is to consider:

- A collection of database tables (or any set of data)
- A sequence of reads of rows in those tables

- The set of transactions committed at any moment

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

Oracle Database transactions executing in `READ COMMITTED` mode are transaction-set consistent on an individual-statement basis, because all rows read by a query must be committed before the query begins.

Oracle Database transactions executing in `SERIALIZABLE` mode are transaction set consistent on an individual-transaction basis, because all statements in a `SERIALIZABLE` transaction execute on an image of the database as of the beginning of the transaction.

In other database systems, a single query run in `READ COMMITTED` mode provides results that are not transaction set consistent. The query is not transaction set consistent, because it might see only a subset of the changes made by another transaction. For example, a join of a master table with a detail table can see a master record inserted by another transaction, but not the corresponding details inserted by that transaction, or vice versa. The `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 2–6 summarizes key similarities and differences between `READ COMMITTED` and `SERIALIZABLE` transactions.

Table 2–6 Read Committed and Serializable Transactions

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

Table 2–6 (Cont.) Read Committed and Serializable Transactions

Operation	Read Committed	Serializable
Waits for blocking transaction	Yes	Yes
Subject to "can't serialize access" error	No	Yes
Error after blocking transaction aborts	No	No
Error after blocking transaction commits	No	Yes

Choosing an Isolation Level for Transactions

Choose an isolation level that is appropriate to the specific application and workload. You might choose different isolation levels for different transactions. The choice depends on performance and consistency needs, and consideration of application coding requirements.

For environments with many concurrent users rapidly submitting transactions, you must assess transaction performance against the expected transaction arrival rate and response time demands, and choose an isolation level that provides the required degree of consistency while performing well. Frequently, for high performance environments, you must trade-off between consistency and concurrency (transaction throughput).

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

`READ COMMITTED` isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results (due to phantoms and unrepeatable reads) for some transactions. The `SERIALIZABLE` isolation level provides somewhat more consistency by protecting against phantoms and unrepeatable reads, and might 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:** *Oracle Database PL/SQL Language Reference* for detailed information on autonomous transactions.

At times, you might 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 might want to commit a customer's information regardless of whether the overall stock purchase actually goes through. Or, while running that same transaction, you might want to log error messages to a debug table even if the overall transaction rolls back. Autonomous transactions enable you to do such tasks.

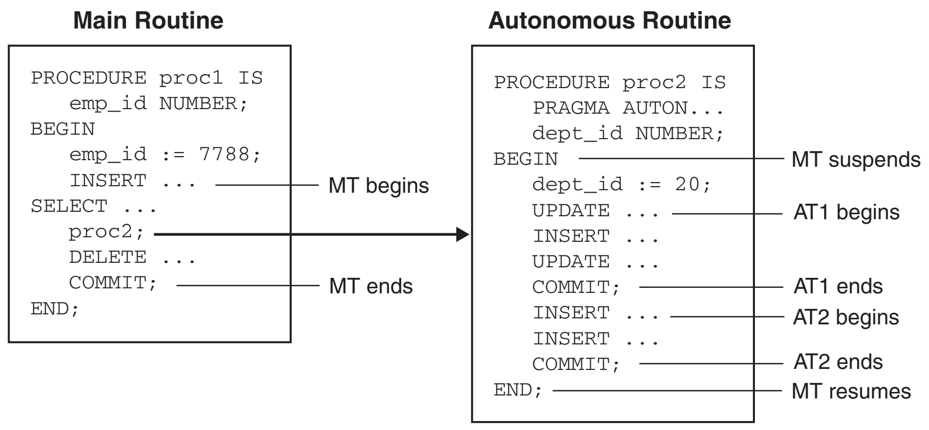
An **autonomous transaction** (AT) is an independent transaction started by another transaction, the **main transaction** (MT). It lets you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction.

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 subprograms
- Methods of a SQL object type
- PL/SQL triggers

Figure 2–3 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 2–3 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;
```

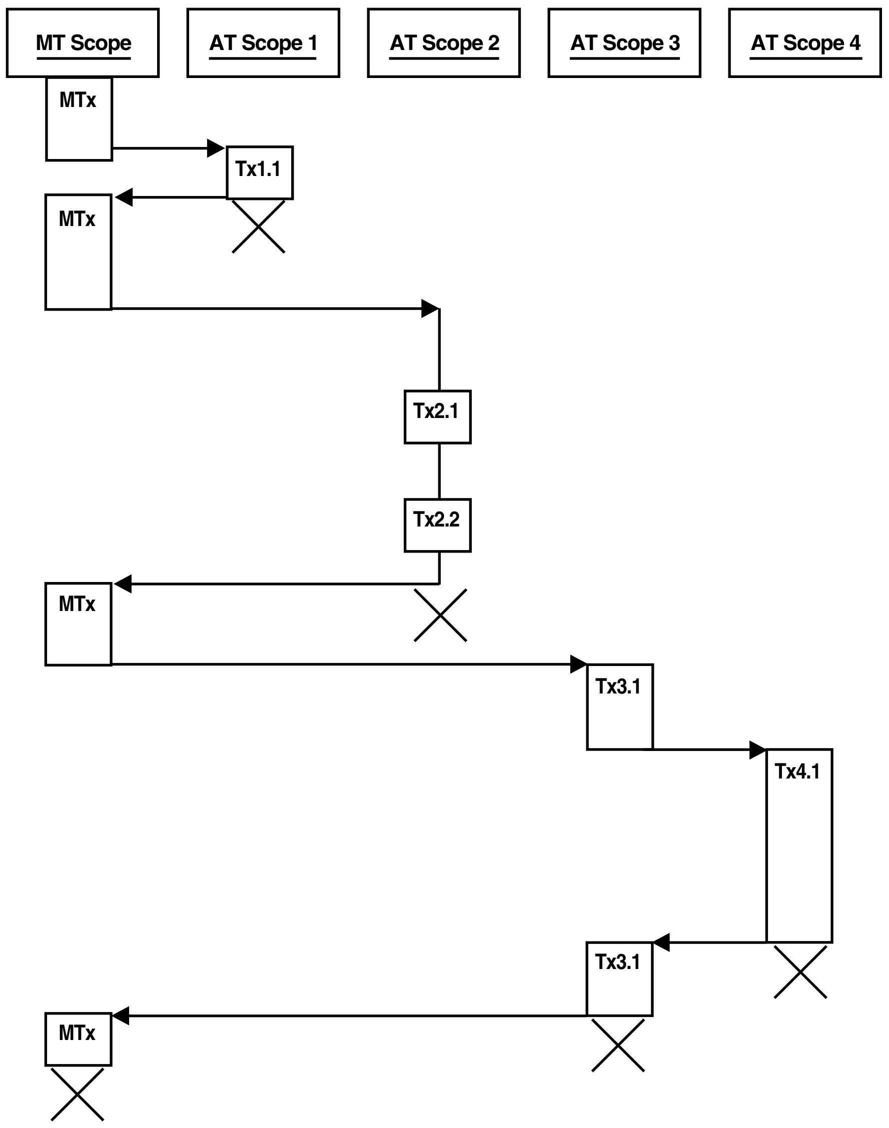
When you enter the executable section of an autonomous routine, the main routine suspends. When you exit the routine, the main routine resumes. `COMMIT` and `ROLLBACK` end the active autonomous transaction but do not exit the autonomous routine. As Figure 2–3 shows, when one transaction ends, the next 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 2–4 illustrates some of the possible sequences autonomous transactions can follow.

Figure 2–4 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

- Ordering a Product
- Withdrawing Money from a Bank Account

As these examples illustrate, there are four possible outcomes when you use autonomous and main transactions (see Table 2–7). There is no dependency between the outcome of an autonomous transaction and that of a main transaction.

Table 2–7 Possible Transaction Outcomes

Autonomous Transaction	Main Transaction
Commits	Commits
Commits	Rolls back
Rolls back	Commits

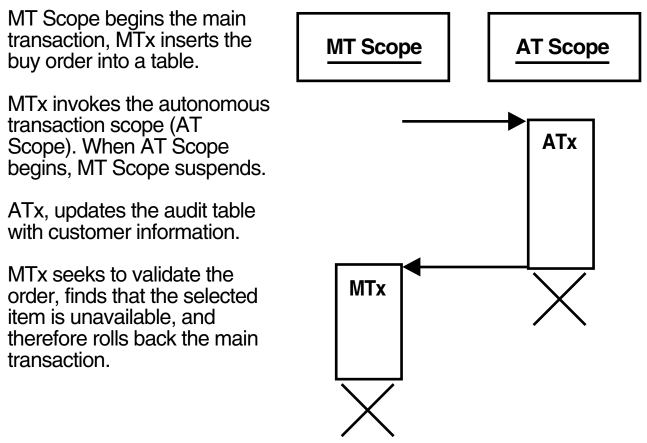
Table 2–7 (Cont.) Possible Transaction Outcomes

Autonomous Transaction	Main Transaction
Rolls back	Rolls back

Ordering a Product

In the example illustrated by Figure 2–5, a customer orders a product. The customer's information (such as name, address, phone) is committed to a customer information table—even though the sale does not go through.

Figure 2–5 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
ATx
MTx

Withdrawing Money from a Bank Account

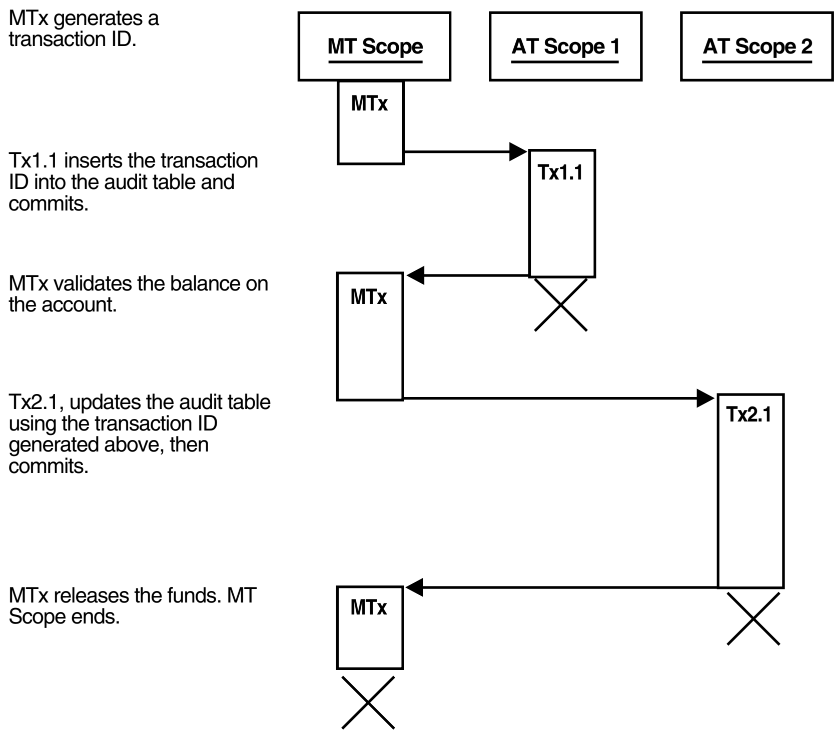
In this example, a customer tries to withdraw money from a bank account. In the process, a main transaction invokes one of two autonomous transaction scopes (AT Scope 1 or AT Scope 2).

The possible scenarios for this transaction are:

- Scenario 1: Sufficient Funds
- Scenario 2: Insufficient Funds with Overdraft Protection
- Scenario 3: Insufficient Funds Without Overdraft Protection

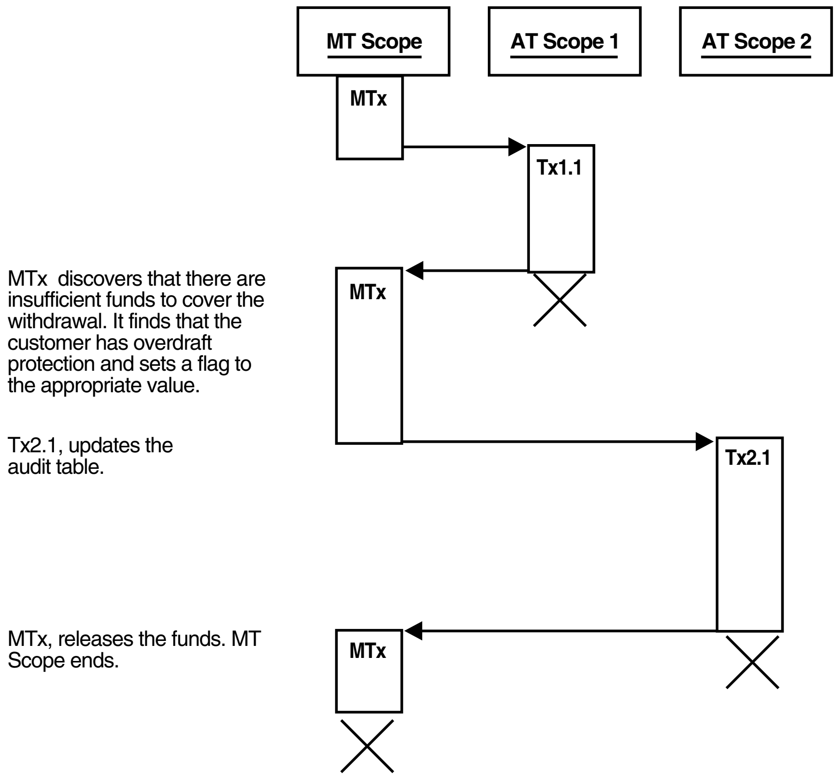
Scenario 1: Sufficient Funds There are sufficient funds to cover the withdrawal, so the bank releases the funds (see Figure 2–6).

Figure 2–6 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.
MT Scope
AT Scope 1
AT Scope 2
MTx
Tx1.1
MTx
Tx2.1
MTx

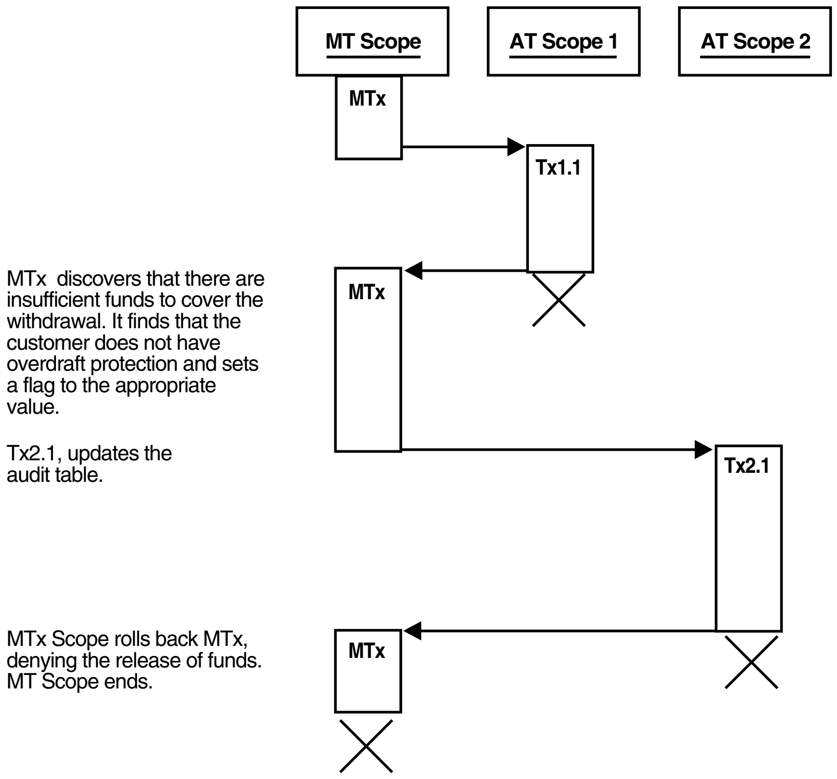
Scenario 2: Insufficient Funds with Overdraft Protection There are insufficient funds to cover the withdrawal, but the customer has overdraft protection, so the bank releases the funds (see Figure 2–7).

Figure 2–7 Bank Withdrawal—Insufficient Funds with 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 has overdraft protection and sets a flag to the appropriate value.
MTx
Tx2.1, updates the audit table.
Tx2.1
MTx, releases the funds. MT Scope ends.
MTx

Scenario 3: Insufficient Funds Without Overdraft Protection There are insufficient funds to cover the withdrawal and the customer does not have overdraft protection, so the bank withholds the requested funds (see Figure 2–8).

Figure 2–8 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.
Tx2.1, updates the audit table.
MTx
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 *Oracle Database PL/SQL Language Reference*.

To define autonomous transactions, you use the pragma (compiler directive) `AUTONOMOUS_TRANSACTION`. The pragma instructs the PL/SQL compiler to mark the subprogram or PL/SQL block as autonomous (independent).

You can code the pragma anywhere in the declarative section of a subprogram or PL/SQL block. But, for readability, code the pragma at the top of the section. The syntax follows:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

In the following example, you mark a packaged function as autonomous:

```
CREATE OR REPLACE PACKAGE Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
    -- add additional functions and packages
END Banking;

CREATE OR REPLACE PACKAGE BODY Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        My_bal REAL;
    BEGIN
       --add appropriate code
    END;
    -- add additional functions and packages...
END Banking;
```

Restrictions on Autonomous Transactions

Autonomous transactions have the following restrictions:

- You cannot use the pragma to mark all subprograms in a package (or all methods in an object type) as autonomous. Only individual routines can be marked autonomous. For example, the following pragma is illegal:

```
CREATE OR REPLACE PACKAGE Banking AS
    PRAGMA AUTONOMOUS_TRANSACTION; -- illegal
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
    END Banking;
```

- You cannot execute a `PIPE ROW` statement in your autonomous routine while your autonomous transaction is open. You must close the autonomous transaction before executing the `PIPE ROW` statement. This is normally accomplished by committing or rolling back the autonomous transaction before executing the `PIPE ROW` statement.

See Also: *Oracle Database PL/SQL Language Reference*

Resuming Execution After Storage Allocation Error

When a long-running transaction is interrupted by an out-of-space error condition, your application can suspend the statement that encountered the problem and resume it after the space problem is corrected. This capability is known as **resumable storage allocation**. It lets you avoid time-consuming rollbacks, without the need to split the operation into smaller pieces and write your own code to track its progress.

See Also:

- *Oracle Database Concepts* for more information about resumable storage allocation
- *Oracle Database Administrator's Guide* for more information about resumable storage allocation

Topics:

- What Operations Can Be Resumed After an Error Condition?
- Handling Suspended Storage Allocation

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 subprogram, 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.

Certain storage errors cannot be handled using this technique. In dictionary-managed tablespaces, you cannot resume an operation if you run into the limit for rollback segments, or the maximum number of extents while creating an index or a table. Use locally managed tablespaces and automatic undo management in combination with this feature.

Handling 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 invoke 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 *Oracle Database 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 static data dictionary views `DBA_RESUMABLE` and `USER_RESUMABLE` (described in *Oracle Database Reference*) and the dynamic performance view `V$_SESSION_WAIT` (described in *Oracle Database Reference*).

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.

In Example 2–1, a 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 waits for eight hours before resuming, with the expectation that the storage problem will be fixed by then.

Example 2–1 Resumable Storage Allocation

```
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;
```

3

Using SQL Datatypes in Database Applications

This chapter explains how to use SQL datatypes in database applications.

Topics:

See Also:

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

Overview of SQL 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 subprogram. These properties cause Oracle Database to treat values of one datatype differently from values of another datatype. For example, Oracle Database can add values of `NUMBER` datatype, but not values of `RAW` datatype.

Oracle Database provides a number of built-in datatypes as well as several categories for user-defined types that can be used as datatypes. The datatypes supported by Oracle Database can be divided into the following categories:

- Oracle built-in datatypes, which include datatypes for characters, numbers, dates and times (known as **datetime datatypes**), raw data, large objects (LOBs), and row addresses (ROWIDs).
- ANSI datatypes and datatypes from the IBM products SQL/DS and DB2, which are usable in SQL statements that create tables and clusters
- User-defined types, which use Oracle built-in datatypes and other user-defined datatypes as the building blocks of object types that model the structure and action of data in applications
- Oracle supplied types, which are SQL-based interfaces for defining new types

The Oracle precompilers recognize other datatypes in embedded SQL programs. These datatypes are called **external datatypes** and are associated with host variables. Do not confuse Oracle Database built-in datatypes and user-defined types with external datatypes.

See Also:

- *Oracle Database SQL Language Reference* for complete reference information on the SQL datatypes
- *Pro*COBOL Programmer's Guide* and *Pro*C/C++ Programmer's Guide* for information on external datatypes, including how Oracle converts between them and built-in or user-defined types
- *Oracle Database Concepts* to learn about Oracle built-in datatypes

Representing Character Data

This section contains the following topics:

- Overview of Character Datatypes
- Specifying Column Lengths as Bytes or Characters
- Choosing Between CHAR and VARCHAR2 Datatypes
- Using Character Literals in SQL Statements

Overview of Character Datatypes

You can use the following SQL datatypes to store alphanumeric data:

- `CHAR` and `NCHAR` datatypes store fixed-length character literals.
- `VARCHAR2` and `NVARCHAR2` datatypes store variable-length character literals.
- `NCHAR` and `NVARCHAR2` datatypes store Unicode character data only.
- `CLOB` and `NCLOB` datatypes store single-byte and multibyte character strings of up to (4 gigabytes - 1) * (the value obtained from `DBMS_LOB.GETCHUNKSIZE`).
- The `LONG` datatype stores variable-length character strings containing up to two gigabytes, but with many restrictions. This datatype is provided only for backward compatibility with existing applications. In general in new applications, use `CLOB` and `NCLOB` datatypes to store large amounts of character data, and `BLOB` and `BFILE` to store large amounts of binary data.
- The `LONG RAW` datatype is similar to the `RAW` datatype, except that it stores raw data with a length up to two gigabytes (2^31-1 bytes). The `LONG RAW` datatype is provided only for backward compatibility with existing applications.

See Also:

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information on `LOB` datatypes (including `CLOB` and `NCLOB` datatypes) and migration from `LONG` to `LOB` datatypes
- *Oracle Database SQL Language Reference* for restrictions on `LONG` datatypes

Specifying Column Lengths as Bytes or Characters

You can specify the lengths of `CHAR` and `VARCHAR2` columns 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.

Consider the following list of column length specifications:

- `id VARCHAR2(32 BYTE)`

  The `id` column contains only single-byte data, up to 32 bytes.

- `name VARCHAR2(32 CHAR)`

  The `name` column contains data in the database character set. If the database character set allows multibyte characters, then the 32 characters can be stored as more than 32 bytes.

- `biography NVARCHAR2(2000)`

  The `biography` column can represent 2000 characters in any Unicode-representable language. The encoding depends on the national character set, but the column can contain multibyte values even if the database character set is single-byte.

- `comment VARCHAR2(2000)`

  The representation of `comment` as 2000 bytes or characters depends on the initialization parameter `NLS_LENGTH_SEMANTICS`.

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, however, then there generally is no such correspondence. A character might consist of one or more bytes, depending upon the specific multibyte encoding scheme and whether shift-in/shift-out control codes are present. To avoid overflowing buffers, specify data as `NCHAR` or `NVARCHAR2` if it might use a Unicode encoding that is different from the database character set.

See Also:

- *Oracle Database Globalization Support Guide* for more information about SQL datatypes `NCHAR` and `NVARCHAR2`
- *Oracle Database SQL Language Reference* for more information about SQL datatypes `NCHAR` and `NVARCHAR2`

Choosing Between CHAR and VARCHAR2 Datatypes

When deciding which datatype to use for a column that stores 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, whereas the VARCHAR2 datatype does not add 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 fully supported. At this time, the VARCHAR datatype automatically corresponds to the VARCHAR2 datatype and is reserved for future use.

When an application interfaces with Oracle Database, there is a character set on the client and server side. Oracle Database uses the NLS_LANGUAGE parameter to automatically convert CHAR, VARCHAR2, and LONG data from the database character set to the character set defined for the user session, if these are different.

Oracle Database SQL Language Reference explains the comparison semantics that Oracle Database uses to compare character data. Because Oracle Database blank-pads values stored in CHAR columns but not in VARCHAR2 columns, a value stored in a VARCHAR2 column can take up less space than the same value 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 may be able to improve performance by storing data in VARCHAR2 rather than in CHAR columns.

Performance is not the only factor to consider when deciding which datatype to use. Oracle Database uses different semantics to compare values of each datatype. You might choose one datatype over the other if your application is sensitive to the differences between these semantics. For example, if you want Oracle Database to ignore trailing blanks when comparing character values, then you must store these values in CHAR columns.

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

Using Character Literals in SQL Statements

Many SQL statements, functions, expressions, and conditions require you to specify character literal values. You can specify character literals with the following notations:

- Character literals with the `'text'` notation, as in the literals `'users01.dbf'` and `'Muthu''s computer'`.

- National character literals with the `N'text'` or `n'text'` notation, where `N` or `n` specifies the literal using the national character set. For example, `N'résumé'` is a National character literal.

  Oracle Database translates N-quoted text into the national character set by way of the database character set. If client-side characters do not have corresponding encoding in the database character set, then Oracle Database converts them into question marks. To avoid the potential loss of data during the text literal conversion, set the environment variable $ORA_NCHAR_LITERAL_REPLACE to TRUE. This setting transparently replaces the `N'`*`text`*`'` internally and preserves the text literal for SQL processing.

The `UNISTR` function provides support for Unicode character literals by enabling you to specify the Unicode encoding value of characters in the string, as in `UNISTR('\1234')`. This technique is useful, for example, when inserting data into `NCHAR` columns. Because every character has a corresponding Unicode encoding, the client application can safely send character data to the server without data loss.

By default you must quote character literals in single-quotes, as in `'Hello'`. This technique can sometimes be inconvenient if the text itself contains single quotes. In such cases, you can also use the Q-quote mechanism, which enables you to specify `q` or `Q` followed by a single quote and then another character to be used as the quote delimiter. For example, the literal `q'#it's the "final" deadline#'` uses the pound sign (#) as a quote delimiter for the string `it's the "final" deadline`.

The Q-quote delimiter can be any single- or multibyte character except space, tab, and return. If the opening quote delimiter is a `[`, `{`, `<`, or `(` character, then the closing quote delimiter must be the corresponding `]`, `}`, `>`, or `)` character. In all other cases, the opening and closing delimiter must be the identical character.

The following character literals use the alternative quoting mechanism:

```
q'(name LIKE '%DBMS_%%')'
q'<'Data,' he said, 'Make it so.'>'
q'"name like '['"'
nq'ïŸ1234ï'
```

See Also:

- *Oracle Database Globalization Support Guide* for information about national character sets
- *Oracle Database SQL Language Reference* for information about character literals

Representing Numeric Data

This section contains the following topics:

- Overview of Numeric Datatypes
- Floating-Point Number Formats
- Comparison Operators for Native Floating-Point Datatypes
- Arithmetic Operations with Native Floating-Point Datatypes
- Conversion Functions for Native Floating-Point Datatypes
- Client Interfaces for Native Floating-Point Datatypes

Overview of Numeric Datatypes

The SQL datatypes `NUMBER`, `BINARY_FLOAT`, and `BINARY_DOUBLE` store 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 Database platforms, and offer up to 38 decimal digits of precision. You can store positive and negative numbers of magnitude 1×10^{-130} through 9.99×10^{125}, as well as 0, in a `NUMBER` column.

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

In client interfaces supported by Oracle Database, the native instruction set supplied by the hardware vendor performs arithmetic operations on BINARY_FLOAT and BINARY_DOUBLE datatypes. The term **native floating-point datatypes** refers to datatypes including BINARY_FLOAT and BINARY_DOUBLE and to all implementations of these types in supported client interfaces.

The floating-point number system is a common way of representing and manipulating numeric values in computer systems. A floating-point number is characterized by the following components:

- Binary-valued sign
- Signed exponent
- Significand
- Base

A floating-point value is the signed product of its significand and the base raised to the power of its exponent, as in the following formula:

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

For example, the number 4.31 is represented as follows:

$(-1)^{0} \cdot 431 \cdot 10^{-2}$

The components of the preceding representation are as follows:

Component Name	Component Value
Sign	0
Significand	431
Base	10
Exponent	-2

See Also:

- *Oracle Database Concepts* for information about the internal format for the NUMBER datatype
- *Oracle Database SQL Language Reference* for more information about the NUMBER datatype
- *Oracle Database SQL Language Reference* for more information about the BINARY_FLOAT and BINARY_DOUBLE datatypes

Floating-Point Number Formats

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

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

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

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

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

Topics:

- Using a Floating-Point Binary Format
- Representing Special Values with Native Floating-Point Formats

Using a Floating-Point Binary Format

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

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

Table 3–1 describes the components of the formula.

Table 3–1 Components of the Binary Format for Floating-Point Numbers

Component	Specifies . . .
s	0 or 1
E	Any integer between E_{min} and E_{max}, inclusive (see Table 3–2)
b_i	0 or 1, where the sequence of bits represents a number in base 2 (see Table 3–2)

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

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

The parameters for these formats are described in Table 3–2.

Table 3–2 Summary of Binary Format Parameters

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

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

Table 3–3 Summary of Binary Format Storage Parameters

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

A significand is **normalized** when the leading bit of the significand is set. IEEE 754 defines **denormal** or **subnormal** values as numbers that are too small to be represented with an implied leading set bit in the significand. The number is too small because its exponent would be too large if its significand were normalized to have an implied leading bit set. IEEE 754 formats support subnormal values. Subnormal values preserve the following property:

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

then: x == y

Table 3–4 shows the range and precision of the required formats in the IEEE 754 standard and those of Oracle `NUMBER`. Range limits are expressed here in terms of positive numbers; they also apply to the absolute value of a negative number. (The notation "*number* e *exponent*" used here stands for *number* multiplied by 10 raised to the *exponent* power: $number \cdot 10^{exponent}$.)

Table 3–4 Range and Precision of IEEE 754 formats

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

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

See Also:

- *Oracle Database SQL Language Reference*, section "Numeric Literals", for information about literal representation of numeric values
- *Oracle Database SQL Language Reference* for more information about floating-point formats

Representing Special Values with Native Floating-Point Formats

Table 3–5 shows the special values that IEEE 754 allows to be represented.

Table 3–5 Special Values for Negative Floating-Point Formats

Value	Meaning
`+INF`	Positive infinity
`-INF`	Negative infinity
`NaN`	Not a number
`+0`	Positive zero
`-0`	Negative zero

`NaN` represent results of operations that are undefined. Many bit patterns in IEEE 754 represent `NaN`. Bit patterns can represent `NaN` with and without the sign bit set. IEEE 754 distinguishes between signalling `NaN`s and quiet `NaN`s.

IEEE 754 specifies action for when exceptions are enabled and disabled. Oracle Database does not allow exceptions to be enabled; the database action is that specified by IEEE 754 for when exceptions are disabled. In particular, Oracle Database makes no distinction between signalling and quiet `NaN`s. Programmers who use OCI can retrieve `NaN` values from Oracle Database; whether a retrieved `NaN` value is signalling or quiet depends on the client platform and beyond the control of Oracle Database.

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

Ignoring signs, there are the following classes of values, with each of the classes except for `NaN` greater than the one preceding it in the list:

- Zero
- Subnormal
- Normal
- Infinity
- `NaN`

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

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

- All `NaN`s are quiet.
- IEEE 754 exceptions are not raised.
- `NaN` is ordered as follows:

  All non-`NaN` < `NaN`

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

See Also: "Comparison Operators for Native Floating-Point Datatypes" on page 3-10 for more information on `NaN` compared to other values

Comparison Operators for Native Floating-Point Datatypes

Oracle Database defines the following comparison operators for operations involving floating-point datatypes:

- Equal to
- Not equal to
- Greater than
- Greater than or equal to
- Less than
- Less than or equal to
- Unordered

Note the following special cases:

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

> **See Also:** "Representing Special Values with Native Floating-Point Formats" on page 3-8 for more information on comparison results, ordering, and other actions of special values.

Arithmetic Operations with Native Floating-Point Datatypes

Oracle Database defines operators for the following arithmetic operations:

- Multiplication
- Division
- Addition
- Subtraction
- Remainder
- Square root

You can define the mode used to round the result of the operation. Exceptions can be raised when operations are performed. Exceptions can also be disabled.

Formerly, Java required floating-point arithmetic to be exactly reproducible. IEEE 754 does not require such action. The standard allows for the result of operations, including arithmetic, to be delivered to a destination that uses a range greater than that used by the operands to the operation.

You can compute the result of a double-precision multiplication at an extended double-precision destination. When this is done, the result must be rounded as if the destination were single-precision or double-precision. The range of the result, that is, the number of bits used for the exponent, can use the range supported by the wider (extended double-precision) destination. This occurrence may result in a double-rounding error in which the least significant bit of the result is incorrect.

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

Conversion Functions for Native Floating-Point Datatypes

Oracle Database defines functions that convert between floating-point and other formats, including string formats that use decimal precision (precision may be lost during the conversion). For example, you can use the following functions:

- `TO_BINARY_DOUBLE`, which converts float to double, decimal (string) to double, and float or double to integer-valued double
- `TO_BINARY_FLOAT`, which converts double to float, decimal (string) to float, and float or double to integer-valued float
- `TO_CHAR`, which converts float or double to decimal (string)
- `TO_NUMBER`, which converts a float, double, or string to a number

Oracle Database can raise exceptions during conversion. The IEEE 754 specification defines the following exceptions:

- Invalid
- Inexact
- Divide by zero
- Underflow
- Overflow

Oracle Database does not raise these exceptions for native floating-point datatypes. Generally, situations that raise exceptions produce the values described in Table 3–6.

Table 3–6 Values Resulting from Exceptions

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

Client Interfaces for Native Floating-Point Datatypes

Oracle Database has implemented support for native floating-point datatypes in the following client interfaces:

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

Topics:

- OCI Native Floating-Point Datatypes SQLT_BFLOAT and SQLT_BDOUBLE
- Native Floating-Point Datatypes Supported in Oracle OBJECT Types
- Pro*C/C++ Support for Native Floating-Point Datatypes

OCI Native Floating-Point Datatypes SQLT_BFLOAT and SQLT_BDOUBLE

The OCI API implements the IEEE 754 single precision and double precision native floating-point datatypes with the datatypes `SQLT_BFLOAT` and `SQLT_BDOUBLE` respectively. Conversions between these types and the SQL types `BINARY_FLOAT` and `BINARY_DOUBLE` are exact on platforms that implement the IEEE 754 standard for the C datatypes `FLOAT` and `DOUBLE`.

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

Native Floating-Point Datatypes Supported in Oracle OBJECT Types

Oracle Database supports the SQL datatypes `BINARY_FLOAT` and `BINARY_DOUBLE` as attributes of Oracle `OBJECT` types.

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

Pro*C/C++ supports the native `FLOAT` and `DOUBLE` datatypes using the column datatypes `BINARY_FLOAT` and `BINARY_DOUBLE`. You can use these datatypes in the same way that Oracle `NUMBER` datatype is used. You can bind the native C/C++ datatypes `FLOAT` and `DOUBLE` to `BINARY_FLOAT` and `BINARY_DOUBLE` types respectively by setting the Pro*C/C++ precompiler command line option `NATIVE_TYPES` to `Y` (yes) when you compile your application.

Representing Date and Time Data

This section contains the following topics:

- Overview of Date and Time Datatypes
- Changing the Default Date Format
- Changing the Default Time Format
- Arithmetic Operations with Date and Time Datatypes
- Converting Between Date and Time Types
- Importing and Exporting Date and Time Types

Overview of Date and Time Datatypes

Oracle Database supports the following date and time datatypes:

- `DATE`

  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.

- `TIMESTAMP`

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

- `TIMESTAMP WITH TIME ZONE`

  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.

- TIMESTAMP WITH LOCAL TIME ZONE

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

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

- INTERVAL DAY TO SECOND

  Use the INTERVAL DAY TO SECOND datatype 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 with high precision, you can use a large value for the days portion.

- INTERVAL YEAR TO MONTH

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

Oracle Database stores dates in its own internal format. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second.

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

Displaying Current Date and Time

Use the SQL function SYSDATE to return the system date and time. You can use the FIXED_DATE initialization parameter to set SYSDATE to a constant, which can be useful for testing.

By default, SYSDATE is printed without any BC or AD qualifier. You can add BC to the format string to print the date with BC or AD as appropriate:

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY BC')
FROM DUAL;

TO_CHAR(SYSDAT
--------------
24-JAN-2004 AD
```

For input and output of dates, the standard Oracle Database default date format is DD-MON-RR. The RR datetime format element enables you store 20th century dates in the 21st century by specifying only the last two digits of the year.

As explained in *Oracle Database SQL Language Reference*, the century of the return value varies according to the specified two-digit year and the last two digits of the current year. For example, the following format refers to the year 1954 in a query issued between 1950 and 2049, but to the year 2054 in a query issued between 2050 and 2099:

```
'13-NOV-54'
```

Changing the Default Date Format

Use the following techniques to change the default date format:

- To change on an instance-wide basis, use the NLS_DATE_FORMAT parameter.
- To change 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:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR') ,'YYYY') "Year"
FROM DUAL;
```

Be careful when using a date format such as 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.

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

Changing the Default Time Format

Time is stored in the following 24-hour format:

```
HH24:MI:SS
```

By default, the time in a DATE column is 12:00:00 A.M. (midnight) if no time portion is entered or if the DATE is truncated.

In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO_DATE function with a format mask indicating the time portion, as shown in Example 3–1.

Example 3–1 Indicating Time with the TO_DATE Function

```
-- create test table
CREATE TABLE birthdays
( Bname VARCHAR2(20),
  Bday  DATE
);

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

Arithmetic Operations with Date and Time Datatypes

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

- ADD_MONTHS function, which returns the date plus the specified number of months.

- SYSDATE function, which returns the current date and time set for the operating system on which the database resides.
- SYSTIMESTAMP function, which returns the system date, including fractional seconds and time zone, of the system on which the database resides.
- TRUNC function, which when applied to a DATE value, trims off the time portion so that it represents the very beginning of the day (the stroke of midnight). By truncating two DATE values and comparing them, you can determine whether they refer to the same day. You can also use TRUNC along with a GROUP BY clause to produce daily totals.
- Arithmetic operators such as + and -. For example, SYSDATE-7 refers to 7 days before the current system date.
- INTERVAL datatypes, which enable you to represent constants when performing date arithmetic rather than performing your own calculations. For example, you can add or subtract INTERVAL constants from DATE values or subtract two DATE values and compare the result to an INTERVAL.
- Comparison operators such as >, <, =, and BETWEEN.

Converting Between Date and Time Types

Oracle Database provides several useful functions that enable you to convert to a from datetime datatypes. Some useful functions include:

- EXTRACT, which extracts and returns the value of a specified datetime field from a datetime or interval value expression
- NUMTODSINTERVAL, which converts a NUMBER or expression that can be implicitly converted to a NUMBER value to an INTERVAL DAY TO SECOND literal
- NUMTOYMINTERVAL, which converts a NUMBER or expression that can be implicitly converted to a NUMBER value to an INTERVAL YEAR TO MONTH literal
- TO_DATE, which converts character data to a DATE datatype
- TO_CHAR, which converts DATE data to character data
- TO_DSINTERVAL, which converts a character string to an INTERVAL DAY TO SECOND value
- TO_TIMESTAMP, which converts character data to a value of TIMESTAMP datatype
- TO_TIMESTAMP_TZ, which converts character data to a value of TIMESTAMP WITH TIME ZONE datatype
- TO_YMINTERVAL, which converts a character string to an INTERVAL YEAR TO MONTH type

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

Importing and Exporting Date and Time Types

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

Representing Specialized Data

This section contains the following topics:

- Representing Geographic Data
- Representing Multimedia Data
- Representing Large Amounts of Data
- Representing Searchable Text
- Representing XML
- Representing Dynamically Typed Data
- Representing Data with ANSI/ISO, DB2, and SQL/DS Datatypes

Representing Geographic Data

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

> **See Also:** *Oracle Spatial Developer's Guide* to learn how to use `MDSYS.SDO_GEOMETRY`

Representing Multimedia Data

Oracle Multimedia enables Oracle Database to store, manage, and retrieve images, audio, video, or other heterogeneous media data in an integrated fashion with other enterprise information. Oracle Multimedia extends Oracle Database reliability, availability, and data management to multimedia content in traditional, Internet, electronic commerce, and media-rich applications.

Whether you store such multimedia data inside the database as `BLOB` or `BFILE` values, or store it externally on a Web server or other kind of server, you can use Oracle Multimedia 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.

Oracle Multimedia provides the `ORDAudio`, `ORDDoc`, `ORDImage`, `ORDImageSignature`, `ORDVideo`, and `SI_StillImage` object types and methods for the following purposes:

- Extracting metadata and attributes from multimedia data
- Retrieving and managing multimedia data from Oracle Multimedia, Web servers, file systems, and other servers
- Performing manipulation operations on image data

> **See Also:** *Oracle Multimedia Reference* for information about Oracle Multimedia types

Representing Large Amounts of Data

Oracle Database provides several datatypes for representing large amounts of data. These datatypes are grouped under the general category of Large Objects (LOBs). Table 3–7 describes the different LOBs.

Table 3–7 Large Object Datatypes

Datatype	Name	Description
`BLOB`	Binary large object	Represents large amounts of binary data such as images, video, or other multimedia data.
`CLOB`	Character large object	Represents large amounts of character data. `CLOB` types are stored by using the database character set. Oracle database stores a `CLOB` up to 4,000 bytes inline as a `VARCHAR2`. If the `CLOB` exceeds this length, then Oracle database moves the `CLOB` out of line.
`NCLOB`	National character large objects	Represents large amounts of character data in National Character Set format.
`BFILE`	External large object	Stores objects in the operating system's file system, outside of the database files or tablespace. The `BFILE` type is read-only; other LOB types are read/write. `BFILE` objects are also sometimes referred to as **external LOBs**.

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

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

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

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

Oracle Net and the Export and Import utilities do not perform character conversion when transmitting `RAW` or `LONG RAW` data. When Oracle Database automatically converts `RAW` or `LONG RAW` data to and from `CHAR` data, as is the case when entering `RAW` data as a literal in an `INSERT` statement, the database represents the data 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`.

You cannot index `LONG RAW` data, but you can index `RAW` data. In earlier releases, the `LONG` and `LONG RAW` datatypes were typically used to store large amounts of data. Use of these types is no longer recommended for new development. If your existing application still uses these types, migrate your application to use LOB types. Oracle recommends that you convert `LONG  RAW` columns to binary LOB (`BLOB`) columns and convert `LONG` columns to character LOB (`CLOB` or `NCLOB`) columns. LOB columns are subject to far fewer restrictions than `LONG` and `LONG RAW` columns.

See Also:

- See *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about LOBs
- See *Oracle Database SQL Language Reference* for restrictions on `LONG` and `LONG RAW` datatypes

Representing Searchable Text

Rather than writing low-level code to do full-text searches, you can use Oracle Text. It stores the search data in a special kind of index, and lets you query the data with operators and PL/SQL packages. This technology enables you 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 with the XPath notation.

See Also: *Oracle Text Application Developer's Guide* for more information

Representing XML

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

`XMLType` columns store their data as either `CLOB` or binary XML. The `XMLType` constructor can turn an existing object of any datatype into an XML object.

When 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 following packages to transfer XML data into and out of relational tables:

- `DBMS_XMLQUERY`, which provides database-to-`XMLType` functionality
- `DBMS_XMLGEN`, which converts the results of a SQL query to a canonical XML format
- `DBMS_XMLSAVE`, which provides XML to database-type functionality

You can use the following SQL functions to process XML:

- `EXTRACT`, which applies a `VARCHAR2` XPath string and returns an `XMLType` instance containing an XML fragment
- `SYS_XMLAGG`, which aggregates all of the XML documents or fragments represented by an expression and produces a single XML document
- `SYS_XMLGEN`, which takes an expression that evaluates to a particular row and column of the database, and returns an instance of type `XMLType` containing an XML document
- `UPDATEXML`, which takes as arguments an `XMLType` instance and an XPath-value pair and returns an `XMLType` instance with the updated value
- `XMLAGG`, which takes a collection of XML fragments and returns an aggregated XML document
- `XMLCOLATTVAL`, which creates an XML fragment and then expands the resulting XML so that each XML fragment has the name column with the attribute name
- `XMLCONCAT`, which takes as input a series of `XMLType` instances, concatenates the series of elements for each row, and returns the concatenated series
- `XMLELEMENT`, which takes an element name for identifier, an optional collection of attributes for the element, and arguments that make up the content of the element
- `XMLFOREST`, which converts each of its argument parameters to XML, and then returns an XML fragment that is the concatenation of these converted arguments
- `XMLSEQUENCE`, which either takes as input an `XMLType` instance and returns a varray of the top-level nodes in the `XMLType`, or takes as input a `REFCURSOR`

instance, with an optional instance of the `XMLFormat` object, and returns as an `XMLSequence` type an XML document for each row of the cursor

`XMLTRANSFORM`, which takes as arguments an `XMLType` instance and an XSL style sheet, applies the style sheet to the instance, and returns an `XMLType`

See Also:

- *Oracle XML DB Developer's Guide* for details about the `XMLType` datatype
- *Oracle XML Developer's Kit Programmer's Guide* for information about client-side programming with XML
- *Oracle Database SQL Language Reference* for information about XML functions

Representing Dynamically Typed Data

Some languages allow datatypes to change at run time or let a program check the type of a variable. For example, C has the `union` keyword and the `void *` pointer, while Java has the `typeof` operator and wrapper types such as `Number`. Oracle Database includes features that enable you to create variables and columns that can hold data of any type and test such data values to determine their underlying representation. For example, you can use these features to have a single table column 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 `SYS.ANYTYPE` in combination with the `DBMS_TYPES` package. The program in Example 3–2 represents data of different underlying types in a table, then interprets the underlying type of each row and processes each value appropriately.

Example 3–2 Accessing Information in a SYS.ANYDATA Column

```
-- This example defines and executes a PL/SQL procedure that
-- uses methods built into SYS.ANYDATA to access information about
-- data stored in a SYS.ANYDATA table column.
DROP TYPE Employee_type FORCE;
DROP TABLE mytab;
CREATE OR REPLACE TYPE Employee_type AS OBJECT ( empno NUMBER,
  ename VARCHAR2(10) );
/
CREATE TABLE mytab ( id NUMBER, data SYS.ANYDATA );
INSERT INTO mytab VALUES (1, SYS.ANYDATA.ConvertNumber(5));
INSERT INTO mytab VALUES (2,
                          SYS.ANYDATA.ConvertObject(Employee_type(5555, 'john')));
COMMIT;
CREATE OR REPLACE PROCEDURE p
IS
  CURSOR cur IS SELECT id, data FROM mytab;
  v_id          mytab.id%TYPE;
  v_data        mytab.data%TYPE;
  v_type        SYS.ANYTYPE;
  v_typecode    PLS_INTEGER;
  v_typename    VARCHAR2(60);
  v_dummy       PLS_INTEGER;
  v_n           NUMBER;
  v_employee    Employee_type;
  non_null_anytype_for_NUMBER exception;
  unknown_typename            exception;
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO v_id, v_data;
    EXIT WHEN cur%NOTFOUND;

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

    v_typecode := v_data.GetType ( v_type /* OUT */ );

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

    CASE v_typecode
      WHEN DBMS_TYPES.TYPECODE_NUMBER THEN
        IF v_type IS NOT NULL
-- This condition should never happen, but check just in case.
          THEN RAISE non_null_anytype_for_NUMBER; END IF;
-- For each type, there is a Get method.
        v_dummy := v_data.GetNUMBER ( v_n /* OUT */ );
        DBMS_OUTPUT.PUT_LINE (
          TO_CHAR(v_id) || ': NUMBER = ' || To_Char(v_n) );
      WHEN DBMS_TYPES.TYPECODE_OBJECT THEN
        v_typename := v_data.GetTypeName();
-- An object type's name is qualified with the schema name.
        IF v_typename NOT IN ( 'HR.EMPLOYEE_TYPE' )
-- If we encounter any object type besides EMPLOYEE_TYPE, raise an exception.
          THEN RAISE unknown_typename; END IF;
        v_dummy := v_data.GetObject ( v_employee /* OUT */  );
        DBMS_OUTPUT.PUT_LINE (
          To_Char(v_id) || ': user-defined type = ' || v_typename ||
            ' ( ' || v_employee.empno || ', ' || v_employee.ename || ' )' );
    END CASE;
  END LOOP;
  CLOSE cur;
EXCEPTION
  WHEN non_null_anytype_for_NUMBER THEN
    RAISE_Application_Error ( -20000,
      'Paradox: the return AnyType instance FROM GetType ' ||
      'should be NULL for all but user-defined types' );
  WHEN unknown_typename THEN
    RAISE_Application_Error ( -20000, 'Unknown user-defined type ' ||
      v_typename || ' - program written to handle only HR.EMPLOYEE_TYPE' );
END;
/
```

The query and procedure in Example 3–2 produce output like that shown in Example 3–3.

Example 3–3   Sample Output for Example 3–2

```
SQL> SELECT t.data.gettypename() AS "Type Name" FROM mytab t;

Type Name
```

```
--------------------------------------------------------------------------------
SYS.NUMBER
HR.EMPLOYEE_TYPE

SQL> EXEC p;
1: NUMBER = 5
2: user-defined type = HR.EMPLOYEE_TYPE ( 5555, john )
```

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

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_TYPES` package
- *Oracle Database Object-Relational Developer's Guide* for information and examples using the `ANYDATA`, `ANYDATASET`, and `ANYTYPE` types
- *Oracle Call Interface Programmer's Guide* for details about the OCI interfaces

Representing Data with ANSI/ISO, DB2, and SQL/DS Datatypes

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

The ANSI datatype conversions are shown in Table 3–8. The ANSI/ISO datatypes `NUMERIC`, `DECIMAL`, and `DEC` can specify only fixed-point numbers. For these datatypes, `s` defaults to 0.

Table 3–8 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`

Table 3–9 shows the SQL/DS and DB2 conversions.

Table 3–9 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes

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

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

Representing Conditional Expressions as Data

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

For example, suppose you create a `traders` table in which row holds data for a stock trading account holder. You want to define a column that stores information about stocks each trader is interested in as a conditional expression. You follow these steps:

1. Create a table `traders` holds data for a stock trading account holder:

   ```
   CREATE TABLE traders
   ( name     VARCHAR2(50),
     email    VARCHAR2(50),
     interest VARCHAR2(50)
   );
   ```

2. Create the user-defined datatype `ticker` with attributes for the trading symbol, limit price, and amount of change in the stock price:

   ```
   CREATE OR REPLACE TYPE ticker
   AS OBJECT
   ( symbol VARCHAR2(20),
     price  NUMBER,
     change NUMBER
   );
   ```

3. Use the following PL/SQL block to create an attribute set `ticker` based on the `ticker` datatype:

   ```
   BEGIN
     DBMS_EXPFIL.CREATE_ATTRIBUTE_SET( attr_set  => 'ticker',
                                       from_type => 'YES' );
   END;
   ```

4. Associate the attribute set with the expression set stored in the database column `trader.interest` as follows:

   ```
   BEGIN
     DBMS_EXPFIL.ASSIGN_ATTRIBUTE_SET (attr_set => 'ticker',
                                       expr_tab => 'traders',
                                       expr_col => 'interest');
   END;
   ```

   The preceding code places a constraint on the `interest` column that ensures the column stores valid conditional expressions.

5. Populate the table with trader names, email addresses and conditional expressions that represents a stock the trader is interested in at a particular price:

   ```
   INSERT INTO traders (name, email, interest)
     VALUES ('Vishu', 'vishu@abc.com', 'symbol = ''ABC'' AND price > 25');
   ```

6. Use the `EVALUATE` operator to identify the conditional expressions that evaluate to `TRUE` for a given data item. For example, the following query returns traders who are interested in a given stock quote `(symbol='ABC', price=31, change=5.2)`:

   ```
   SELECT Name, Email
   FROM Traders
   WHERE EVALUATE ( interest,
                    'symbol=>''ABC'',
                    price=>31,
                    change=>5.2'
                  ) = 1;
   ```

To speed up this type of query, you can optionally create an Oracle Expression Filter index on the `interest` column.

> **See Also:** *Oracle Database Rules Manager and Expression Filter Developer's Guide* for details on Oracle Expression Filter

Identifying Rows by Address

Each row in a database table has an address called a **rowid**. You can examine a row address by querying the pseudocolumn `ROWID`, whose values are strings representing the address of each row. These strings have the datatype `ROWID` or `UROWID`. You can also create tables and clusters that contain actual columns having the `ROWID` datatype. Oracle Database does not guarantee that the values of such columns are valid rowids.

Rowid values are important for application development for the following reasons:

- They are the fastest way to access a single row.
- They can show you how the rows in a table are stored.
- They are unique identifiers for rows in a table.

> **See Also:**
>
> - *Oracle Database Concepts* for general information about the `ROWID` pseudocolumn and the `ROWID` datatype
> - *Oracle Database SQL Language Reference* to learn about the `ROWID` pseudocolumn

Topics:

- Querying the ROWID Pseudocolumn
- Accessing the ROWID Datatype
- Accessing the UROWID Datatype

Querying the ROWID Pseudocolumn

Each table in Oracle Database has a pseudocolumn named ROWID. If the row is too large to fit within a single data block, then 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.

The following SQL statements return the ROWID pseudocolumn of the row of the hr.employees table that satisfies the query, and inserts it into the t_tab table:

```
CREATE TABLE t_tab (col1 ROWID);
INSERT INTO t_tab
  SELECT ROWID
  FROM hr.employees
  WHERE employee_id = 7499;
```

Note: Although you can use the ROWID pseudocolumn in the SELECT and WHERE clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the ROWID pseudocolumn.

Accessing the ROWID Datatype

In tables that are not index-organized and foreign tables, the values of the ROWID pseudocolumn have the datatype ROWID. The format of this datatype is either **extended** or **restricted**.

Topics:

- Restricted ROWID
- Extended ROWID
- External Binary ROWID

Restricted ROWID

Internally, the ROWID is a structure that holds information that the database server needs to access a row. The restricted internal ROWID is 6 bytes on most platforms. Each restricted rowid includes the following data:

- Datafile identifier
- Block identifier
- Row identifier

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

Extended ROWID

The extended ROWID datatype includes the data in the restricted rowid plus a data object number. The data object number is an identification number assigned to every database segment. The extended internal ROWID is 10 bytes on most platforms.

Data in an extended ROWID pseudocolumn is returned to the client application 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. Extended rowids are not available directly. You can use a supplied package, DBMS_ROWID, to interpret extended rowid contents. The package functions extract and provide information that is available directly from a restricted rowid as well as information specific to extended rowids.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_ROWID package

External Binary ROWID

Some client applications use a binary form of the ROWID. For example, OCI and some precompiler applications can map the ROWID datatype 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 as follows:

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

Accessing the UROWID Datatype

The rows of some tables have addresses that are not physical or permanent or were not generated by Oracle Database. For example, the row addresses of index-organized tables are stored in index leaves, which can move. Oracle provides these tables with logical row identifiers, called **logical rowids**. Rowids of foreign tables, such as DB2 tables accessed through a gateway, are not standard Oracle Database rowids. Oracle provides foreign tables with identifiers called **foreign rowids**.

Oracle Database uses universal rowids (urowids) to store the addresses of index-organized and foreign tables. Both types of urowid are stored in the ROWID pseudocolumn, as are the physical rowids of heap-organized tables.

Oracle creates logical rowids based on the primary key of the table. The logical rowids do not change as long as the primary key does not change. The ROWID pseudocolumn of an index-organized table has a datatype of UROWID. You can access this pseudocolumn as you would access the ROWID pseudocolumn of a heap-organized table (that is, using a SELECT ROWID statement). To store the rowids of an index-organized table, define a column of type UROWID for the table and retrieve the value of the ROWID pseudocolumn into that column.

How Oracle Database Converts Datatypes

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

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

Topics:

- Datatype Conversion During Assignments
- Datatype Conversion During Expression Evaluation

Datatype Conversion During Assignments

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

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

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

- `variable := expression`

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

  ```
  VAR1 := 0;
  ```

- `INSERT INTO Table1_tab VALUES (expression1, expression2, ...)`

  The datatypes of `expression1`, `expression2`, and so on, must be either the same as, or convertible to, the datatypes of the corresponding columns in `Table1_tab`. For example, Oracle Database automatically converts the data provided in the following `INSERT` statement for `Table1_tab`:

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

- `UPDATE Table1_tab SET column = expression`

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

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

- `SELECT` *`column`* `INTO` *`variable`* `FROM Table1_tab`

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

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

Datatype Conversion During Expression Evaluation

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

Oracle Database can automatically convert the following:

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

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

Some common types of expressions follow:

- Simple expressions, such as:

  ```
  commission + '500'
  ```

- Boolean expressions, such as:

  ```
  bonus > salary / '10'
  ```

- Subprogram calls, such as:

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

- `WHERE` clause conditions, such as:

  ```
  WHERE hiredate = TO_DATE('1997-01-01','yyyy-mm-dd')
  ```

- `WHERE` clause conditions, such as:

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

In general, Oracle Database uses the rule for expression evaluation when a datatype conversion is needed in places not covered by the rule for assignment conversions.

In assignments of the form:

```
variable := expression
```

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

Metadata for SQL Built-In Functions

You can see metadata for SQL built-in functions with the dynamic performance views `V$SQLFN_METADATA` (which has general metadata) and `V$SQLFN_ARG_METADATA` (which has metadata about arguments). You can join these views on the column `FUNCID`. For functions with unlimited arguments, such as `LEAST` and `GREATEST`, `V$SQLFN_ARG_METADATA` has only one row for each repeating argument.

These views allow third-party tools to leverage SQL built-in functions without maintaining their metadata in the application layer.

See Also: *Oracle Database Reference* for detailed information about the dynamic performance views `V$SQLFN_METADATA` and `V$SQLFN_ARG_METADATA`

Often, an argument for a SQL built-in function can have any datatype in a datatype family. Table 3–10 shows which datatypes belong to which families.

Table 3–10 Datatype Families

Family	Datatypes
STRING	CHARACTER
	VARCHAR2
	CLOB
	NCHAR
	NVARCHAR2
	NCLOB
NUMERIC	NUMBER
	BINARY_FLOAT
	BINARY_DOUBLE
DATETYPE	DATE
	TIMESTAMP
	TIMESTAMP WITH TIME ZONE
	TIMESTAMP WITH LOCAL TIME ZONE
	INTERVAL YEAR TO MONTH
	INTERVAL DAY TO SECOND
BINARY	BLOB
	RAW
	LONGRAW

ARG*n* Datatype

In the view `V$SQLFN_METADATA`, `ARG`*n* is the datatype of a function whose return value has the same datatype as its *n*th argument. For example:

- The `MAX` function returns a value that has the datatype of its first argument, so the `MAX` function has datatype `ARG1`.
- The `DECODE` function returns a value that has the datatype of its third argument, so the `DECODE` function has datatype `ARG3`.

EXPR Datatype

In the view `V$SQLFN_ARG_METADATA`, `EXPR` is the datatype of an argument that can be any expression. An expression is either a single value or a combination of values and SQL functions that has a single value.

Table 3–11 Display Types of SQL Built-In Functions

Display Type	Description	Example
NORMAL	FUNC(A,B,...)	LEAST(A,B,C)

Table 3–11 (Cont.) Display Types of SQL Built-In Functions

Display Type	Description	Example
`ARITHMETIC`	`A FUNC B)`	`A+B`
`PARENTHESIS`	`FUNC()`	`SYS_GUID()`
`RELOP`	`A FUNC B)`	`A IN B`
`CASE_LIKE`	`CASE` statement or `DECODE` decode	
`NOPAREN`	`FUNC`	`SYSDATE`

4

Using Regular Expressions in Database Applications

This chapter explains how to use regular expressions in database applications.

Topics:

- Overview of Regular Expressions
- Metacharacters in Regular Expressions
- Using Regular Expressions in SQL Statements: Scenarios

See Also:

- *Oracle Database SQL Language Reference* for information about Oracle Database SQL functions for regular expressions
- *Oracle Database Globalization Support Guide* for details on using SQL regular expression functions in a multilingual environment
- *Oracle Regular Expressions Pocket Reference* by Jonathan Gennick, O'Reilly & Associates
- *Mastering Regular Expressions* by Jeffrey E. F. Friedl, O'Reilly & Associates

Overview of Regular Expressions

Topics:

- What Are Regular Expressions?
- How Are Regular Expressions Useful?
- Oracle Database Implementation of Regular Expressions
- Oracle Database Support for the POSIX Regular Expression Standard

What Are Regular Expressions?

Regular expressions enable you to search for patterns in string data by using standardized syntax conventions. You specify a regular expression through the following types of characters:

- Metacharacters, which are operators that specify search algorithms
- Literals, which are the characters for which you are searching

A regular expression can specify complex patterns of character sequences. For example, the following regular expression searches for the literals `f` or `ht`, the `t` literal, the `p` literal optionally followed by the `s` literal, and finally the colon (`:`) literal:

```
(f|ht)tps?:
```

The parentheses are metacharacters that group a series of pattern elements to a single element; the pipe symbol (`|`) matches one of the alternatives in the group. The question mark (`?`) is a metacharacter indicating that the preceding pattern, in this case the `s` character, is optional. Thus, the preceding regular expression matches the `http:`, `https:`, `ftp:`, and `ftps:` strings.

How Are Regular Expressions Useful?

Regular expressions are a powerful text processing component of programming languages such as Perl and Java. For example, a Perl script can process each HTML file in a directory, read its contents into a scalar variable as a single string, and then use regular expressions to search for URLs in the string. One reason that many developers write in Perl is for its robust pattern matching functionality.

Oracle's support of regular expressions enables developers to implement complex match logic in the database. This technique is useful for the following reasons:

- By centralizing match logic in Oracle Database, you avoid intensive string processing of SQL results sets by middle-tier applications. For example, life science customers often rely on Perl to do pattern analysis on bioinformatics data stored in huge databases of DNAs and proteins. Previously, finding a match for a protein sequence such as `[AG].{4}GK[ST]` was handled in the middle tier. The SQL regular expression functions move the processing logic closer to the data, thereby providing a more efficient solution.
- Prior to Oracle Database 10*g*, developers often coded data validation logic on the client, requiring the same validation logic to be duplicated for multiple clients. Using server-side regular expressions to enforce constraints solves this problem.
- The built-in SQL and PL/SQL regular expression functions and conditions make string manipulations more powerful and less cumbersome than in previous releases of Oracle Database.

Oracle Database Implementation of Regular Expressions

Oracle Database implements regular expression support with a set of Oracle Database SQL functions and conditions that enable you to search and manipulate string data. You can use these functions in any environment that supports Oracle Database SQL. You can use these functions on a text literal, bind variable, or any column that holds character data such as `CHAR`, `NCHAR`, `CLOB`, `NCLOB`, `NVARCHAR2`, and `VARCHAR2` (but not `LONG`).

Table 4–1 describes the regular expression functions and conditions.

Table 4–1 SQL Regular Expression Functions and Conditions

SQL Element	Category	Description
`REGEXP_LIKE`	`Condition`	Searches a character column for a pattern. Use this function in the `WHERE` clause of a query to return rows matching a regular expression. The condition is also valid in a constraint or as a PL/SQL function returning a boolean. The following `WHERE` clause filters employees with a first name of Steven or Stephen: `WHERE REGEXP_LIKE(first_name, '^Ste(v|ph)en$')`
`REGEXP_REPLACE`	`Function`	Searches for a pattern in a character column and replaces each occurrence of that pattern with the specified string. The following function call puts a space after each character in the `country_name` column: `REGEXP_REPLACE(country_name, '(.)', '\1 ')`
`REGEXP_INSTR`	`Function`	Searches a string or substring for a given occurrence of a regular expression pattern (a substring) and returns an integer indicating the position in the string or substring where the match is found. You specify which occurrence you want to find and the start position. The following function call performs a boolean test for a valid email address in the `email` column: `REGEXP_INSTR(email, '\w+@\w+(\.\w+)+') > 0`
`REGEXP_SUBSTR`	`Function`	Searches a string or substring for a given occurrence of a regular expression pattern (a substring) and returns the substring itself. You specify which occurrence you want to find and the start position. The following function call uses the `x` flag to match the first string by ignoring spaces in the regular expression: `REGEXP_SUBSTR('oracle', 'o r a c l e', 1, 1, 'x')`
`REGEXP_COUNT`	`Function`	Returns the number of times a pattern appears in a string. You specify the string and the pattern. You can also specify the start position and matching options (for example, `c` for case sensitivity). The following function call returns the number of times that `e` (but not `E`) appears in the string `'Albert Einstein'`, starting at character position 7 (that is, one): `REGEXP_COUNT('Albert Einstein', 'e', 7, 'c')`

A string literal in a `REGEXP` function or condition conforms to the rules of SQL text literals. By default, regular expressions must be enclosed in single quotes. If your regular expression includes the single quote character, then enter two single quotation marks to represent one single quotation mark within the expression. This technique ensures that the entire expression is interpreted by the SQL function and improves the readability of your code. You can also use the q-quote syntax to define your own character to terminate a text literal. For example, you can delimit your regular expression with the pound sign (`#`) and then use a single quote within the expression.

Note: If your expression comes from a column or a bind variable, then the same rules for quoting do not apply.

See Also:

- *Oracle Database SQL Language Reference* for syntax, descriptions, and examples of the `REGEXP` functions and conditions
- *Oracle Database SQL Language Reference* for information about character literals

Oracle Database Support for the POSIX Regular Expression Standard

Oracle's implementation of regular expressions conforms to the following standards:

- IEEE Portable Operating System Interface (POSIX) standard draft 1003.2/D11.2
- Unicode Regular Expression Guidelines of the Unicode Consortium

Oracle Database follows the exact syntax and matching semantics for these operators as defined in the POSIX standard for matching ASCII (English language) data. You can find the POSIX standard draft at the following URL:

`http://www.opengroup.org/onlinepubs/007908799/xbd/re.html`

Oracle Database enhances regular expression support in the following ways:

- Extends the matching capabilities for multilingual data beyond what is specified in the POSIX standard.
- Adds support for the common Perl regular expression extensions that are not included in the POSIX standard but do not conflict with it. Oracle Database provides built-in support for some of the most heavily used Perl regular expression operators, for example, character class shortcuts, the non-greedy modifier, and so on.

Oracle Database supports a set of common metacharacters used in regular expressions. The action of supported metacharacters and related features is described in "Metacharacters in Regular Expressions" on page 4-4.

Note: The interpretation of metacharacters differs between tools that support regular expressions. If you are porting regular expressions from another environment to Oracle Database, ensure that the regular expression syntax is supported and the action is what you expect.

Metacharacters in Regular Expressions

This section contains the following topics:

- POSIX Metacharacters in Oracle Database Regular Expressions
- Multilingual Extensions to POSIX Regular Expression Standard
- Perl-Influenced Extensions to POSIX Regular Expression Standard

POSIX Metacharacters in Oracle Database Regular Expressions

Table 4–2 lists the list of metacharacters supported for use in regular expressions passed to SQL regular expression functions and conditions. These metacharacters conform to the POSIX standard; any differences in action from the standard are noted in the "Description" column.

Table 4–2 POSIX Metacharacters in Oracle Database Regular Expressions

Syntax	Operator Name	Description	Example
`.`	Any Character — Dot	Matches any character in the database character set. If the `n` flag is set, it matches the newline character. The newline is recognized as the linefeed character (`\x0a`) on Linux, UNIX, and Windows or the carriage return character (`\x0d`) on Macintosh platforms. **Note:** In the POSIX standard, this operator matches any English character except NULL and the newline character.	The expression `a.b` matches the strings `abb`, `acb`, and `adb`, but does not match `acc`.
`+`	One or More — Plus Quantifier	Matches one or more occurrences of the preceding subexpression.	The expression `a+` matches the strings `a`, `aa`, and `aaa`, but does not match `bbb`.
`?`	Zero or One — Question Mark Quantifier	Matches zero or one occurrence of the preceding subexpression.	The expression `ab?c` matches the strings `abc` and `ac`, but does not match `abbc`.
`*`	Zero or More — Star Quantifier	Matches zero or more occurrences of the preceding subexpression. By default, a quantifier match is greedy because it matches as many times as possible while still allowing the rest of the match to succeed.	The expression `ab*c` matches the strings `ac`, `abc`, and `abbc`, but does not match `abb`.
`{`*`m`*`}`	Interval—Exact Count	Matches exactly *m* occurrences of the preceding subexpression.	The expression `a{3}` matches the strings `aaa`, but does not match `aa`.
`{`*`m`*`,}`	Interval—At Least Count	Matches at least *m* occurrences of the preceding subexpression.	The expression `a{3,}` matches the strings `aaa` and `aaaa`, but does not match `aa`.
`{`*`m`*`,`*`n`*`}`	Interval—Betwee n Count	Matches at least *m*, but not more than *n* occurrences of the preceding subexpression.	The expression `a{3,5}` matches the strings `aaa`, `aaaa`, and `aaaaa`, but does not match `aa`.
`[ ... ]`	Matching Character List	Matches any single character in the list within the brackets. The following operators are allowed within the list, but other metacharacters included are treated as literals: ■ Range operator: `-` ■ POSIX character class: `[: :]` ■ POSIX collation element: `[. .]` ■ POSIX character equivalence class: `[= =]` A dash (`-`) is a literal when it occurs first or last in the list, or as an ending range point in a range expression, as in `[#--]`. A right bracket (`]`) is treated as a literal if it occurs first in the list. **Note:** In the POSIX standard, a range includes all collation elements between the start and end of the range in the linguistic definition of the current locale. Thus, ranges are linguistic rather than byte values ranges; the semantics of the range expression are independent of character set. In Oracle Database, the linguistic range is determined by the `NLS_SORT` initialization parameter.	The expression `[abc]` matches the first character in the strings `all`, `bill`, and `cold`, but does not match any characters in `doll`.
`[^ ... ]`	Nonmatching Character List	Matches any single character not in the list within the brackets. Characters not in the nonmatching character list are returned as a match. See the description of the Matching Character List operator for an account of metacharacters allowed in the character list.	The expression `[^abc]` matches the character `d` in the string `abcdef`, but not the character `a`, `b`, or `c`. The expression `[^abc]+` matches the sequence `def` in the string `abcdef`, but not `a`, `b`, or `c`. The expression `[^a-i]` excludes any character between `a` and `i` from the search result. This expression matches the character `j` in the string `hij`, but does not match any characters in the string `abcdefghi`.

Table 4–2 (Cont.) POSIX Metacharacters in Oracle Database Regular Expressions

Syntax	Operator Name	Description	Example
|	Or	Matches one of the alternatives.	The expression a|b matches character a or character b.
(...)	Subexpression or Grouping	Treats the expression within parentheses as a unit. The subexpression can be a string of literals or a complex expression containing operators.	The expression (abc)?def matches the optional string abc, followed by def. Thus, the expression matches abcdefghi and def, but does not match ghi.
\n	Backreference	Matches the *n*th preceding subexpression, that is, whatever is grouped within parentheses, where *n* is an integer from 1 to 9. The parentheses cause an expression to be remembered; a backreference refers to it. A backreference counts subexpressions from left to right, starting with the opening parenthesis of each preceding subexpression. The expression is invalid if the source string contains fewer than *n* subexpressions preceding the \n. Oracle supports the backreference expression in the regular expression pattern and the replacement string of the REGEXP_REPLACE function.	The expression (abc|def)xy\1 matches the strings abcxyabc and defxydef, but does not match abcxydef or abcxy. A backreference enables you to search for a repeated string without knowing the actual string ahead of time. For example, the expression ^(.*)\1$ matches a line consisting of two adjacent instances of the same string.
\	Escape Character	Treats the subsequent metacharacter in the expression as a literal. Use a backslash (\) to search for a character that is normally treated as a metacharacter. Use consecutive backslashes (\\\\) to match the backslash literal itself.	The expression \+ searches for the plus character (+). It matches the plus character in the string abc+def, but does not match abcdef.
^	Beginning of Line Anchor	Matches the beginning of a string (default). In multiline mode, it matches the beginning of any line within the source string.	The expression ^def matches def in the string defghi but does not match def in abcdef.
$	End of Line Anchor	Matches the end of a string (default). In multiline mode, it matches the beginning of any line within the source string.	The expression def$ matches def in the string abcdef but does not match def in the string defghi.

Table 4–2 (Cont.) POSIX Metacharacters in Oracle Database Regular Expressions

Syntax	Operator Name	Description	Example
`[:`*`class`*`:]`	POSIX Character Class	Matches any character belonging to the specified POSIX character *`class`*. You can use this operator to search for characters with specific formatting such as uppercase characters, or you can search for special characters such as digits or punctuation characters. The full set of POSIX character classes is supported. **Note:** In English regular expressions, range expressions often indicate a character class. For example, `[a-z]` indicates any lowercase character. This convention is not useful in multilingual environments, where the first and last character of a given character class might not be the same in all languages. Oracle supports the character classes in Table 4–3 based on character class definitions in Globalization classification data.	The expression `[[:upper:]]+` searches for one or more consecutive uppercase characters. This expression matches `DEF` in the string `abcDEFghi` but does not match the string `abcdefghi`.
`[.`*`element`*`.]`	POSIX Collating Element Operator	Specifies a collating element to use in the regular expression. The *`element`* must be a defined collating element in the current locale. Use any collating element defined in the locale, including single-character and multicharacter elements. The `NLS_SORT` initialization parameter determines supported collation elements. This operator lets you use a multicharacter collating element in cases where only one character is otherwise allowed. For example, you can ensure that the collating element `ch`, when defined in a locale such as Traditional Spanish, is treated as one character in operations that depend on the ordering of characters.	The expression `[[.ch.]]` searches for the collating element `ch` and matches `ch` in string `chabc`, but does not match `cdefg`. The expression `[a-[.ch.]]` specifies the range `a` to `ch`.
`[=`*`character`*`=]`	POSIX Character Equivalence Class	Matches all characters that are members of the same character equivalence class in the current locale as the specified *`character`*. The character equivalence class must occur within a character list, so the character equivalence class is always nested within the brackets for the character list in the regular expression. Usage of character equivalents depends on how canonical rules are defined for your database locale. See *Oracle Database Globalization Support Guide* for more information on linguistic sorting and string searching.	The expression `[[=n=]]` searches for characters equivalent to `n` in a Spanish locale. It matches both `N` and `ñ` in the string `El Niño`.

See Also: *Oracle Database SQL Language Reference* for syntax, descriptions, and examples of the `REGEXP` functions and conditions

Multilingual Extensions to POSIX Regular Expression Standard

When applied to multilingual data, Oracle's implementation of the POSIX operators extends beyond the matching capabilities specified in the POSIX standard. Table 4–3 shows the relationship of the operators in the context of the POSIX standard.

- The first column lists the supported operators.
- The second column indicates whether the POSIX standard for Basic Regular Expression (BRE) defines the operator.
- The third column indicates whether the POSIX standard for Extended Regular Expression (ERE) defines the operator.
- The fourth column indicates whether the Oracle Database implementation extends the operator's semantics for handling multilingual data.

Oracle Database lets you enter multibyte characters directly, if you have a direct input method, or use functions to compose the multibyte characters. You cannot use the Unicode hexadecimal encoding value of the form *xxxx*. Oracle evaluates the characters based on the byte values used to encode the character, not the graphical representation of the character.

Table 4–3 POSIX and Multilingual Operator Relationships

Operator	POSIX BRE syntax	POSIX ERE Syntax	Multilingual Enhancement
\	Yes	Yes	--
*	Yes	Yes	--
+	--	Yes	--
?	--	Yes	--
|	--	Yes	--
^	Yes	Yes	Yes
$	Yes	Yes	Yes
.	Yes	Yes	Yes
[ ]	Yes	Yes	Yes
( )	Yes	Yes	--
{m}	Yes	Yes	--
{m,}	Yes	Yes	--
{m,n}	Yes	Yes	--
\n	Yes	Yes	Yes
[..]	Yes	Yes	Yes
[::]	Yes	Yes	Yes
[==]	Yes	Yes	Yes

Perl-Influenced Extensions to POSIX Regular Expression Standard

Table 4–4 describes Perl-influenced metacharacters supported in Oracle Database regular expression functions and conditions. These metacharacters are not in the POSIX standard, but are common at least partly due to the popularity of Perl. Perl character class matching is based on the locale model of the operating system, whereas Oracle Database regular expressions are based on the language-specific data of the database. In general, a regular expression involving locale data cannot be expected to produce the same results between Perl and Oracle Database.

Table 4–4 Perl-Influenced Extensions in Oracle Regular Expressions

Reg. Exp.	Matches . . .	Example
`\d`	A digit character. It is equivalent to the POSIX class `[[:digit:]]`.	The expression `^\(\d{3}\) \d{3}-\d{4}$` matches `(650) 555-1212` but does not match `650-555-1212`.
`\D`	A nondigit character. It is equivalent to the POSIX class `[^[:digit:]]`.	The expression `\w\d\D` matches `b2b` and `b2_` but does not match `b22`.
`\w`	A word character, which is defined as an alphanumeric or underscore (_) character. It is equivalent to the POSIX class `[[:alnum:]_]`. If you do not want to include the underscore character, you can use the POSIX class `[[:alnum:]]`.	The expression `\w+@\w+(\.\w+)+` matches the string `jdoe@company.co.uk` but not the string `jdoe@company`.
`\W`	A nonword character. It is equivalent to the POSIX class `[^[:alnum:]_]`.	The expression `\w+\W\s\w+` matches the string `to: bill` but not the string `to bill`.
`\s`	A whitespace character. It is equivalent to the POSIX class `[[:space:]]`.	The expression `\(\w\s\w\s\)` matches the string `(a b )` but not the string `(ab)`.
`\S`	A nonwhitespace character. It is equivalent to the POSIX class `[^[:space:]]`.	The expression `\(\w\S\w\S\)` matches the string `(abde)` but not the string `(a b d e)`.
`\A`	Only at the beginning of a string. In multi-line mode, that is, when embedded newline characters in a string are considered the termination of a line, `\A` does not match the beginning of each line.	The expression `\AL` matches only the first `L` character in the string `Line1\nLine2\n`, regardless of whether the search is in single-line or multi-line mode.
`\Z`	Only at the end of string or before a newline ending a string. In multi-line mode, that is, when embedded newline characters in a string are considered the termination of a line, `\Z` does not match the end of each line.	In the expression `\s\Z,` the `\s` matches the last space in the string `L i n e \n`, regardless of whether the search is in single-line or multi-line mode.
`\z`	Only at the end of a string.	In the expression `\s\z`, the `\s` matches the newline in the string `L i n e \n`, regardless of whether the search is in single-line or multi-line mode.
`*?`	The preceding pattern element 0 or more times (non-greedy). This quantifier matches the empty string whenever possible.	The expression `\w*?x\w` is "non-greedy" and so matches `abxc` in the string `abxcxd`. The expression `\w*x\w` is "greedy" and so matches `abxcxd in the string abxcxd`. The expression `\w*?x\w` also matches the string `xa`.
`+?`	The preceding pattern element 1 or more times (non-greedy).	The expression `\w+?x\w` is "non-greedy" and so matches `abxc` in the string `abxcxd`. The expression `\w+x\w` is "greedy" and so matches `abxcxd in the string abxcxd`. The expression `\w+?x\w` does not match the string `xa`, but does match the string `axa`.
`??`	The preceding pattern element 0 or 1 time (non-greedy). This quantifier matches the empty string whenever possible.	The expression `a??aa` is "non-greedy" and matches `aa` in the string `aaaa`. The expression `a?aa` is "greedy" and so matches `aaa` in the string `aaaa`.
`{n}?`	The preceding pattern element exactly `n` times (non-greedy). In this case `{n}?` is equivalent to `{n}`.	The expression `(a|aa){2}?` matches `aa` in the string `aaaa`.
`{n,}?`	The preceding pattern element at least `n` times (non-greedy).	The expression `a{2,}?` is "non-greedy" and matches `aa` in the string `aaaaa`. The expression `a{2,}` is "greedy" and so matches `aaaaa`.
`{n,m}?`	At least `n` but not more than `m` times (non-greedy). `{0,m}?` matches the empty string whenever possible.	The expression `a{2,4}?` is "non-greedy" and matches `aa` in the string `aaaaa`. The expression `a{2,4}` is "greedy" and so matches `aaaa`.

The Oracle Database regular expression functions and conditions support the pattern matching modifiers described in Table 4–5.

Table 4–5 Pattern Matching Modifiers

Mod.	Description	Example
`i`	Specifies case-insensitive matching.	The following regular expression returns `AbCd`: `REGEXP_SUBSTR('AbCd', 'abcd', 1, 1, 'i')`
`c`	Specifies case-sensitive matching.	The following regular expression fails to match: `REGEXP_SUBSTR('AbCd', 'abcd', 1, 1, 'c')`
n	Allows the period (.), which by default does not match newlines, to match the newline character.	The following regular expression matches the string only because the `n` flag is specified: `REGEXP_SUBSTR('a'||CHR(10)||'d', 'a.d', 1, 1, 'n')`
m	Performs the search in multi-line mode. The metacharacter `^` and `$` signify the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string.	The following regular expression returns `ac`: `REGEXP_SUBSTR('ab'||CHR(10)||'ac', '^a.', 1, 2, 'm')`
`x`	Ignores whitespace characters in the regular expression. By default, whitespace characters match themselves.	The following regular expression returns `abcd`: `REGEXP_SUBSTR('abcd', 'a b c d', 1, 1, 'x')`

Using Regular Expressions in SQL Statements: Scenarios

This section contains the following scenarios:

- Using a Constraint to Enforce a Phone Number Format
- Using Back References to Reposition Characters

Using a Constraint to Enforce a Phone Number Format

Regular expressions are a useful way to enforce constraints. For example, suppose that you want to ensure that phone numbers are entered into the database in a standard format. Example 4–1 creates a `contacts` table and adds a `CHECK` constraint to the `p_number` column to enforce the following format mask:

```
(XXX) XXX-XXXX
```

Example 4–1 Enforcing a Phone Number Format with Regular Expressions

```
CREATE TABLE contacts (
  l_name    VARCHAR2(30),
  p_number  VARCHAR2(30)
    CONSTRAINT c_contacts_pnf
      CHECK (REGEXP_LIKE (p_number, '^\(\d{3}\) \d{3}-\d{4}$'))
);
```

Table 4–6 explains the elements of the regular expression.

Table 4–6 Explanation of the Regular Expression Elements in Example 4–1

Regular Expression Element	Matches . . .
`^`	The beginning of the string.

Table 4–6 (Cont.) Explanation of the Regular Expression Elements in Example 4–1

Regular Expression Element	Matches . . .
`\(`	A left parenthesis. The backward slash (\) is an escape character that indicates that the left parenthesis following it is a literal rather than a grouping expression.
`\d{3}`	Exactly three digits.
`\)`	A right parenthesis. The backward slash (\) is an escape character that indicates that the right parenthesis following it is a literal rather than a grouping expression.
(space character)	A space character.
`\d{3}`	Exactly three digits.
`-`	A hyphen.
`\d{4}`	Exactly four digits.
`$`	The end of the string.

Example 4–2 shows a SQL script that attempts to insert seven phone numbers into the `contacts` table. Only the first two `INSERT` statements use a format that conforms to the `c_contacts_pnf` constraint; the remaining statements generate `CHECK` constraint errors.

Example 4–2 insert_contacts.sql

```
-- first two statements use valid phone number format
INSERT INTO contacts (p_number)
  VALUES(  '(650) 555-5555'   );
INSERT INTO contacts (p_number)
  VALUES(  '(215) 555-3427'   );
-- remaining statements generate check contraint errors
INSERT INTO contacts (p_number)
  VALUES(  '650 555-5555'     );
INSERT INTO contacts (p_number)
  VALUES(  '650 555 5555'     );
INSERT INTO contacts (p_number)
  VALUES(  '650-555-5555'     );
INSERT INTO contacts (p_number)
  VALUES(  '(650)555-5555'    );
INSERT INTO contacts (p_number)
  VALUES(  ' (650) 555-5555'  );
/
```

Using Back References to Reposition Characters

As explained in Table 4–2, back references store matched subexpressions in a temporary buffer, thereby enabling you to reposition characters. You access buffers with the `\n` notation, where *n* is a number between 1 and 9. Each subexpression is contained in parentheses and is numbered from left to right.

Example 4–3 creates a `famous_people` table and populates the `famous_people.names` column with names in different formats.

Example 4–3 Using Back References to Reposition Characters

```
CREATE TABLE famous_people
  ( names VARCHAR2(30) );
```

```
-- populate table with data
INSERT INTO famous_people
  VALUES ('John Quincy Adams');
INSERT INTO famous_people
  VALUES ('Harry S. Truman');
INSERT INTO famous_people
  VALUES ('John Adams');
INSERT INTO famous_people
  VALUES (' John Quincy Adams');
INSERT INTO famous_people
  VALUES ('John_Quincy_Adams');
COMMIT;
```

Example 4–4 shows a query that repositions names in the format "first middle last" to the format "last, first middle". It ignores names not in the format "first middle last".

Example 4–4 Using Back References to Reposition Characters

```
SELECT names "names",
  REGEXP_REPLACE(names,
                 '^(\S+)\s(\S+)\s(\S+)$',
                 '\3, \1 \2')
  AS "names after regexp"
FROM famous_people;
```

Table 4–7 explains the elements of the regular expression.

Table 4–7 Explanation of the Regular Expression Elements in Example 4–4

Regular Expression Element	Description
`^`	Matches the beginning of the string.
`$`	Matches the end of the string.
`(\S+)`	Matches one or more nonspace characters. The parentheses are not escaped so they function as a grouping expression.
`\s`	Matches a whitespace character.
`\1`	Substitutes the first subexpression, that is, the first group of parentheses in the matching pattern.
`\2`	Substitutes the second subexpression, that is, the second group of parentheses in the matching pattern.
`\3`	Substitutes the third subexpression, that is, the third group of parentheses in the matching pattern.
`,`	Inserts a comma character.

Example 4–5 shows the result set of the query in Example 4–4. The regular expression matched only the first two rows.

Example 4–5 Result Set of Regular Expression Query

```
names
------------------------------
names after regexp
------------------------------
John Quincy Adams
Adams, John Quincy
```

```
Harry S. Truman
Truman, Harry S.

John Adams
John Adams

 John Quincy Adams
 John Quincy Adams

John_Quincy_Adams
John_Quincy_Adams
```

5 Using Indexes in Database Applications

This chapter explains how to use indexes in database applications.

Topics:

- Privileges Needed to Create Indexes
- Guidelines for Application-Specific Indexes
- Examples of Creating Basic Indexes
- When to Use Domain Indexes
- When to Use Function-Based Indexes

See Also:

- *Oracle Database Administrator's Guide* for information about creating and managing indexes
- *Oracle Database Performance Tuning Guide* for detailed information about using indexes
- *Oracle Database SQL Language Reference* for the syntax of statements to work with indexes
- *Oracle Database Administrator's Guide* for information on creating hash clusters to improve performance, as an alternative to indexing

Privileges Needed to Create Indexes

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.

Guidelines for Application-Specific Indexes

You can create indexes on columns to speed up queries. Indexes provide faster access to data for operations that return a small portion of a table's rows.

In general, create an index on a column in any of the following situations:

- The column is queried frequently.

- A referential constraint exists on the column.
- A UNIQUE key constraint exists on the column.

You can create an index on any column; however, if the column is not used in any of these situations, creating an index on the column does not increase performance and the index takes up resources unnecessarily.

Although the database creates an index for you on a column with a constraint, explicitly creating an index on such a column is recommended.

You can use the following techniques to determine which columns are best candidates for indexing:

- Use the EXPLAIN PLAN feature to show a theoretical execution plan of a given query statement.
- Use the dynamic performance view V$SQL_PLAN to determine the actual execution plan used for a given query statement.

Sometimes, if an index is not being used by default and it would be more efficient to use that index, 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 statements, and give guidelines for managing indexes.

See Also:

- *Oracle Database Performance Tuning Guide* for information on using the V$SQL_PLAN view, the EXPLAIN PLAN statement, query hints, and measuring the performance benefits of indexes
- *Oracle Database Reference* for general information about the V$SQL_PLAN view

Topics:

Which Come First, Data or Indexes?

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.

Create a New Temporary Table Space Before Creating Indexes

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

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

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

See Also: *Oracle Database Utilities* for information on direct path load

Index the Correct Tables and Columns

Use the following guidelines for determining when to create an index:

- Create an index if you frequently want to retrieve less than about 15% of the rows in a large table. This threshold percentage varies greatly, however, according to the relative speed of a table scan and how clustered the row data is about the index key. The faster the table scan, the lower the percentage; the more clustered the row data, the higher the percentage.
- Index columns that are used for joins to improve join performance.
- Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key; see Chapter 6, "Maintaining Data Integrity in Database Applications" for more information.
- Small tables do not require indexes; if a query is taking too long, then the table might have grown from small to large.

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

- Values are unique in the column, or there are few duplicates.
- There is a wide range of values (good for regular indexes).
- There is a small range of values (good for bitmap indexes).
- The column contains many nulls, but queries often select all rows having a value. In this case, a comparison that matches all the non-null values, such as:

  ```
  WHERE COL_X >= -9.99 *power(10,125)
  ```

  is preferable to

  ```
  WHERE COL_X IS NOT NULL
  ```

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

Columns with the following characteristics are less suitable for indexing:

- There are many nulls in the column and you do not search on the non-null values.

`LONG` and `LONG RAW` columns cannot be indexed.

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

Limit the Number of Indexes for Each Table

The more indexes, the more overhead is incurred as the table is altered. When rows are inserted or deleted, all indexes on the table must be updated. When a column is updated, all indexes on the column must be updated.

You must weigh the performance benefit of indexes for queries against the performance overhead of updates. For example, if a table is primarily read-only, you might use more indexes; but, if a table is heavily updated, you might use fewer indexes.

Choose Column Order in Composite Indexes

Although you can specify columns in any order in the `CREATE INDEX` statement, the order of columns in the `CREATE INDEX` statement can affect query performance. In general, 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 Example 5–1.

Example 5–1 VENDOR_PARTS Table

```
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         8-300       1.33
1012         8-300       1.19
1292           457       5.28
```

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

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

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

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

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

Composite indexes speed up queries that use the leading portion of the index. So in this example, the performance of queries with `WHERE` clauses using only the `PART_NO` column improve also. Because there are only five distinct values, placing a separate index on `VENDOR_ID` serves no purpose.

Gather Index Statistics

The database can use indexes more effectively when it has statistical information about the tables involved in the queries. You or the DBA can periodically gather statistics by invoking procedures such as `DBMS_STATS.GATHER_TABLE_STATISTICS` and `DBMS_STATS.GATHER_SCHEMA_STATISTICS`. For information about these procedures, see *Oracle Database PL/SQL Packages and Types Reference*.

Drop Unused Indexes

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.

To find out if an index is being used, you can monitor it. If you see that the index is never used, rarely used, or used in a way that seems to provide no benefit, you can either drop it immediately or you can make it invisible until you are sure that you do not need it, and then drop it. If you discover that you do need the invisible index, you can make it visible again.

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.

To drop an index, use the SQL statement `DROP INDEX`. For example, the following statement drops the index named `Emp_name`:

```
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.

See Also:

- *Oracle Database Administrator's Guide* for information about monitoring index usage
- *Oracle Database Administrator's Guide* for information about making indexes invisible
- *Oracle Database SQL Language Reference* for information about the `DROP INDEX` statement

Examples of Creating Basic Indexes

You can create an index for a table to improve the performance of queries issued against the corresponding table. You can also create an index for a cluster. You can create a *composite* index on multiple columns up to a maximum of 32 columns. A composite index key cannot exceed roughly one-half (minus some overhead) of the available space in the data block.

Oracle Database automatically creates an index to enforce a `UNIQUE` or `PRIMARY KEY` 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 statement `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;
```

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);
```

In this example, the query is going to sort on the function `UPPER(ENAME)`. An index on the `ENAME` column itself does not speed up this operation, and it might be slow to invoke 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));
```

When to Use Domain Indexes

Domain indexes are appropriate for special-purpose applications implemented using data cartridges. The domain index helps to manipulate complex data, such as spatial, audio, or video data. If you need to develop such an application, see *Oracle Database Data Cartridge Developer's Guide*.

Oracle Database supplies a number of specialized data cartridges to help manage these kinds of complex data. So, if you need to create a search engine, or a geographic information system, you can do much of the work simply by creating the right kind of index.

When to Use Function-Based Indexes

A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.

Note:

- The index is more effective if you gather statistics for the table or schema, using the procedures in the `DBMS_STATS` package.
- The index cannot contain any null values. Either ensure that 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 5-7.

Function-based indexes have all of the same properties as indexes on columns. Unlike indexes on columns that can be used by both cost-based and rule-based optimization, however, function-based indexes can be used by only by cost-based optimization. Other restrictions on function-based indexes are described in "Restrictions on Function-Based Indexes" on page 5-8.

See Also:

- *Oracle Database Concepts* for general information about function-based indexes
- *Oracle Database Administrator's Guide* for information about creating function-based indexes

Topics:

- Advantages of Function-Based Indexes
- Restrictions on Function-Based Indexes
- Examples of Function-Based Indexes

Advantages of Function-Based Indexes

Function-based indexes:

- Increase the number of situations where the optimizer can perform a range scan instead of a full table scan.

  For example, consider the expression in this `WHERE` clause:

  ```
  CREATE INDEX Idx ON Example_tab(Column_a + Column_b);
  SELECT * FROM Example_tab WHERE Column_a + Column_b < 10;
  ```

  The optimizer can use a range scan for this query because the index is built on (`column_a` + `column_b`). Range scans typically produce fast response times if the predicate selects less than 15% of the rows of a large table. The optimizer can estimate how many rows are selected by expressions more accurately if the expressions are materialized in a function-based index. (Expressions of function-based indexes are represented as virtual columns and `ANALYZE` can build histograms on such columns.)

- Precompute the value of a computationally intensive function and store it in the index.

  An index can store computationally intensive expression that you access often. When you need to access a value, it is already computed, greatly improving query execution performance.

- Create indexes on object columns and `REF` columns.

Methods that describe objects can be used as functions on which to build indexes. For example, you can use the `MAP` method to build indexes on an object type column.

- Create more powerful sorts.

  You can perform case-insensitive sorts with the `UPPER` and `LOWER` functions, descending order sorts with the `DESC` keyword, and linguistic-based sorts with the `NLSSORT` function.

> **Note:** Oracle Database sorts columns with the `DESC` keyword in descending order. Such indexes are treated as function-based indexes. Descending indexes cannot be bitmapped or reverse, and cannot be used in bitmapped optimizations. To get the `DESC` functionality prior to Oracle Database version 8, remove the `DESC` keyword from the `CREATE INDEX` statement.

Another function-based index calls the object method `distance_from_equator` for each city in the table. The method is applied to the object column `Reg_Obj`. A query can 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 can 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');
```

Restrictions on Function-Based Indexes

Function-based indexes have the following restrictions:

- Only cost-based optimization can use function-based indexes. Remember to invoke `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, for example, the `UPPER` function. You must ensure that the subprogram really is deterministic, because Oracle Database does not check that the assertion is true.

  The following semantic rules demonstrate how to use the keyword `DETERMINISTIC`:

  - You can declare a top level subprogram as `DETERMINISTIC`.

  - You can declare a PACKAGE level subprogram as DETERMINISTIC in the PACKAGE specification but not in the PACKAGE BODY. Errors are raised if DETERMINISTIC is used inside a PACKAGE BODY.
  - You can declare a private subprogram (declared inside another subprogram or a PACKAGE BODY) as DETERMINISTIC.
  - A DETERMINISTIC subprogram can invoke another subprogram whether the invoked subprogram is declared as DETERMINISTIC or not.
- If you change the semantics of a DETERMINISTIC function and recompile it, then existing function-based indexes and materialized views report results for the prior version of the function. Thus, if you change the semantics of a function, you must manually rebuild any dependent function-based indexes and materialized views.
- Expressions in a function-based index cannot contain any aggregate functions. The expressions must reference only columns in a row in the table.
- You must analyze the table or index before the index is used.
- Bitmap optimizations cannot use descending indexes.
- Function-based indexes are not used when OR-expansion is done.
- The index function cannot be marked NOT NULL. To avoid a full table scan, you must ensure that the query cannot fetch null values.
- Function-based indexes cannot use expressions that return VARCHAR2 or RAW data types of unknown length from PL/SQL functions. A workaround is to limit the size of the function's output by indexing a substring of known length:

```
-- INITIALS() might return 1 letter, 2 letters, 3 letters, and so on.
-- We limit the return value to 10 characters for purposes of the index.
CREATE INDEX func_substr_index ON
  emp_tab(substr(initials(ename),1,10);

-- Invoke SUBSTR both when creating the index and when referencing
-- the function in queries.
SELECT SUBSTR(initials(ename),1,10) FROM emp_tab;
```

See Also: *Oracle Database SQL Language Reference* for an account of CREATE FUNCTION restrictions.

Examples of Function-Based Indexes

- Function-Based Index for Case-Insensitive Searches
- Precomputing Arithmetic Expressions with a Function-Based Index
- Function-Based Index for Language-Dependent Sorting

Function-Based Index for Case-Insensitive Searches

The following statement allows faster case-insensitive searches in table EMP_TAB.

```
CREATE INDEX Idx ON Emp_tab (UPPER(Ename));
```

The SELECT statement 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;
```

Precomputing Arithmetic Expressions with a Function-Based Index

The following statement 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 (because the expression is a prefix of index IDX) or index fast full scan (which might be preferable if the index has specified a high parallel degree).

```
SELECT a FROM Fbi_tab WHERE A + B * (C - 1) < 100;
```

Function-Based Index for Language-Dependent Sorting

This example demonstrates how a function-based index can be used to sort based on the collation order for a national language. The NLSSORT function returns a sort key for each name, using the collation sequence GERMAN.

```
CREATE INDEX Nls_index
    ON Nls_tab (NLSSORT(Name, 'NLS_SORT = German'));
```

The SELECT statement selects all of the contents of the table and orders it by NAME. The rows are ordered using the German collation sequence. The Globalization Support parameters are not needed in the SELECT statement, because in a German session, NLS_SORT is set to German and NLS_COMP is set to ANSI.

```
SELECT * FROM Nls_tab WHERE Name IS NOT NULL
    ORDER BY Name;
```

6 Maintaining Data Integrity in Database Applications

This chapter explains how to use constraints to enforce the business rules associated with your database and prevent the entry of invalid information into tables.

Topics:

Overview of Constraints

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

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

Enforcing Business Rules with Constraints

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

For example, to ensure that each employee works for a valid department:

1. Create a rule that all values in the department table are unique:

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

2. Create a rule that every department listed in the employee table must match one of the values in the department table:

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

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

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

Enforcing Business Rules with Application Logic

You might enforce business rules through application logic as well as through 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. Create these indexes by hand, rather than letting the database create them. 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 only 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 static data dictionary view or dynamic performance view.
- Oracle Database does not automatically index foreign keys.

When to Use NOT NULL 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 must not have `NOT`

NULL 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 constraints to further restrict the values that can exist in specific columns of a table. Use the combination of NOT NULL and UNIQUE key constraints to force the input of values in the UNIQUE key; this combination of data integrity rules eliminates the possibility that a new row's data conflicts with an existing row's data.

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

> **See Also:** "Defining Relationships Between Parent and Child Tables" on page 6-8

A NOT NULL constraint is specified like this:

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

Example 6–1 shows an example of a table with NOT NULL constraints. The JOB column has a NOT NULL constraint, so no row can have the value NULL in the JOB column. The COMM column does not have a NOT NULL constraint, so any row can have the value NULL in the COMM column.

Example 6–1 EMPLOYEES Table

```
ID    LNAME     JOB       MGR   HIREDATE    SAL     COMM   DEPTNO
---   -------   -------   ---   --------    -----   ----   ------
100   King      AD_PRES         17-JUN-87   24000              90
101   Kochhar   AD_VP     100   21-SEP-89   17000              90
102   De Hann   AD_VP     100   13-JAN-93   17000              90
103   Hunold    IT_PROG   102   03-JAN-90    9000              60
```

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 invokes the USER function:

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

Setting Default Column Values

Default values can be defined using any literal, or almost any expression, including calls to the following:

- SYSDATE
- SYS_CONTEXT
- USER
- USERENV
- 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 invoke any other functions.

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

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

Choosing a Primary Key for a Table

Each table can have one primary key, which uniquely identifies each row in a table and ensures that no duplicate rows exist. When selecting a primary key, use these guidelines:

- Whenever practical, use a column containing a sequence number. This satisfies all the other guidelines.
- Choose a column whose data values are unique, because the purpose of a primary key is to uniquely identify each row of the table.
- Choose a column whose data values never change. A primary key value is only used to identify a row in the table, and its data must never be used for any other purpose.
- Choose a column that does not contain any nulls. A PRIMARY KEY constraint, by definition, does not allow any row to contain a null in any column that is part of the primary key.
- Choose a column that is short and numeric. Short primary keys are easy to type. You can use sequence numbers to easily generate numeric primary keys.
- Minimize your use of composite primary keys. Although composite primary keys are allowed, they do not satisfy all of the other recommendations. For example, composite primary key values are long and cannot be assigned by sequence numbers.

When to Use UNIQUE Constraints

Choose columns for unique keys carefully. The purpose of these constraints is different from that of primary keys. Unique key constraints are appropriate for any column where duplicate values are not allowed. Primary keys identify each row of the table uniquely, and typically contain values that have no significance other than being unique. Figure 6–1 shows an example of a table with a unique key constraint.

Figure 6–1 Table with a UNIQUE Constraint

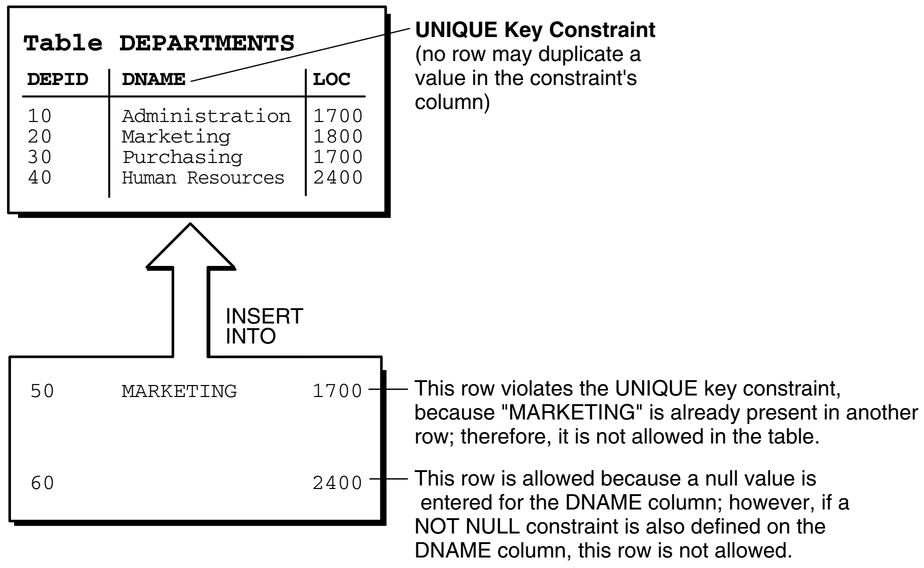


Table DEPARTMENTS
DEPID
DNAME
LOC
10 Administration 1700
20 Marketing 1800
30 Purchasing 1700
40 Human Resources 2400
UNIQUE Key Constraint
(no row may duplicate a value in the constraint's column)
INSERT INTO
50 MARKETING 1700
60 2400
This row violates the UNIQUE key constraint, because "MARKETING" is already present in another row; therefore, it is not allowed in the table.
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.

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

Some examples of good unique keys include:

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

When to Use Constraints On Views

The constraints in this chapter apply to tables, not views.

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

See Also: *Oracle Database Data Warehousing Guide* for information about using constraints in data warehousing

Enforcing Referential Integrity with Constraints

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

See Also: "Defining Relationships Between Parent and Child Tables" on page 6-8 for information on defining additional constraints, including the foreign key

Figure 6–2 shows a foreign key defined on the department number. It guarantees that every value in this column must match a value in the primary key of the department table. This constraint prevents erroneous department numbers from getting into the employee table.

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

Figure 6–2 Tables with FOREIGN KEY Constraints

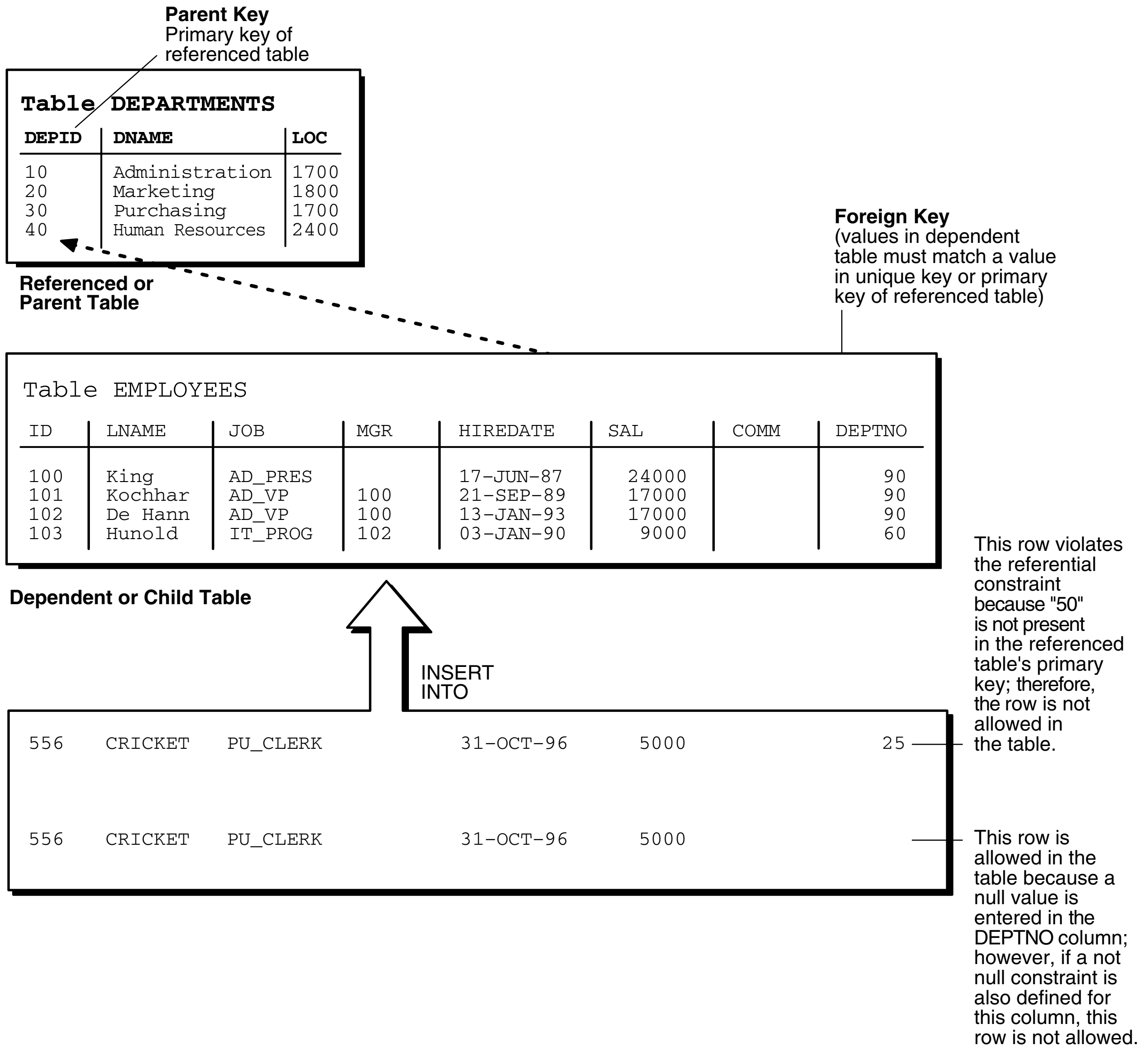

Parent Key
Primary key of referenced table
Referenced or Parent Table
Foreign Key
(values in dependent table must match a value in unique key or primary key of referenced table)
Dependent or Child Table
INSERT INTO
This row violates the referential constraint because "50" is not present in the referenced 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 DEPARTMENTS

DEPID	DNAME	LOC
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
40	Human Resources	2400

Table EMPLOYEES

ID	LNAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
100	King	AD_PRES		17-JUN-87	24000		90
101	Kochhar	AD_VP	100	21-SEP-89	17000		90
102	De Hann	AD_VP	100	13-JAN-93	17000		90
103	Hunold	IT_PROG	102	03-JAN-90	9000		60

556	CRICKET	PU_CLERK		31-OCT-96	5000		25
556	CRICKET	PU_CLERK		31-OCT-96	5000		

FOREIGN KEY Constraints and NULL Values

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

- By default (without any NOT NULL or CHECK clauses), the FOREIGN KEY constraint enforces the **match none** rule for composite foreign keys in the ANSI/ISO standard.
- To enforce the **match full** rule for NULL values in composite foreign keys, which requires that all components of the key be NULL or all be non-null, define a CHECK constraint that allows only all nulls or all non-nulls in the composite foreign key. For example, with a composite key comprised of columns A, B, and C:

```
CHECK ((A IS NULL AND B IS NULL AND C IS NULL) OR
       (A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL))
```

- In general, it is not possible to use declarative referential integrity to enforce the **match partial** rule for NULL values in composite foreign keys, which requires the non-null portions of the key to appear in the corresponding portions in the primary or unique key of a single row in the referenced table. You can often use triggers to handle this case, as described in *Oracle Database PL/SQL Language Reference*.

Defining Relationships Between Parent and Child Tables

Several relationships between parent and child tables can be determined by the other types of constraints defined on the foreign key in the child table.

No Constraints on the Foreign Key When no other constraints are defined on the foreign key, any number of rows in the child table can reference the same parent key value. This model allows nulls in the foreign key.

This model establishes a one-to-many relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. An example of such a relationship is shown in Figure 6–2 between the `employee` and `department` tables. Each department (parent key) has many employees (foreign key), and some employees might not be in a department (nulls in the foreign key).

NOT NULL Constraint on the Foreign Key When nulls are not allowed in a foreign key, each row in the child table must explicitly reference a value in the parent key because nulls are not allowed in the foreign key.

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

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

This model establishes a one-to-one relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. For example, assume that the employee table had a column named `MEMBERNO`, referring to an employee membership number in the company insurance plan. Also, a table named `INSURANCE` has a primary key named `MEMBERNO`, and other columns of the table keep respective information relating to an employee insurance policy. The `MEMBERNO` in the employee table must be both a foreign key and a unique key:

- To enforce referential integrity rules between the `EMP_TAB` and `INSURANCE` tables (the `FOREIGN KEY` constraint)
- To guarantee that each employee has a unique membership number (the `UNIQUE` key constraint)

UNIQUE and NOT NULL Constraints on the Foreign Key When both `UNIQUE` and `NOT NULL` constraints are defined on the foreign key, only one row in the child table can reference a given parent key value, and because `NULL` values are not allowed in the foreign key, each row in the child table must explicitly reference a value in the parent key.

This model establishes a one-to-one relationship between the parent and foreign keys that does not allow undetermined values (nulls) in the foreign key. If you expand the previous example by adding a `NOT NULL` constraint on the `MEMBERNO` column of the employee table, in addition to guaranteeing that each employee has a unique membership number, you also ensure that no undetermined values (nulls) are allowed in the `MEMBERNO` column of the employee table.

Rules for Multiple FOREIGN KEY Constraints

Oracle Database allows a column to be referenced by multiple `FOREIGN KEY` constraints; there is no limit on the number of dependent keys. This situation might be present if a single column is part of two different composite foreign keys.

Deferring Constraint Checks

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

Note: You cannot issue a `SET CONSTRAINTS` statement inside a trigger.

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

See Also: *Oracle Database SQL Language Reference* for more information about the `SET CONSTRAINTS` statement

Consider the following guidelines when deferring constraint checks:

- Select appropriate data.

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

  – Tables are snapshots.

  – Some tables contain a large amount of data being manipulated by another application, which may or may not return the data in the same order.

- Update cascade operations on foreign keys.
- Ensure that constraints are deferrable.

  After you have identified and selected the appropriate tables, ensure that their `FOREIGN`, `UNIQUE` and `PRIMARY` key constraints are created deferrable. You can do so by issuing statements similar to the following:

```
CREATE TABLE dept (
     deptno NUMBER PRIMARY KEY,
     dname VARCHAR2 (30)
     );
CREATE TABLE emp (
     empno NUMBER,
     ename VARCHAR2 (30),
     deptno NUMBER REFERENCES (dept),
     CONSTRAINT pk_emp_empno PRIMARY KEY (empno) DEFERRABLE,
     CONSTRAINT fk_emp_deptno 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 fk_emp_deptno DEFERRED;
UPDATE dept SET deptno = deptno + 10
     WHERE deptno = 20;

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

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

- Set all constraints deferred.

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

  ```
  SET CONSTRAINTS ALL DEFERRED;
  ```

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

- Check the `COMMIT` (optional)

  You can check for constraint violations before committing by issuing the `SET CONSTRAINTS ALL IMMEDIATE` statement just before issuing the `COMMIT`. If there are any problems with a constraint, this statement fails and the constraint causing the error is identified. If you commit while constraints are violated, the transaction rolls back and you receive an error message.

Minimizing Space and Time Overhead for Indexes Associated with Constraints

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

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

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

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

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

```
CREATE TABLE b
(
    b1 INTEGER,
    b2 INTEGER,
    CONSTRAINT u_b_1 (b1, b2) USING INDEX (CREATE UNIQUE INDEX b_index on b(b1,
b2),
    CONSTRAINT u_b_2 (b1, b2) USING INDEX b_index
);
```

Guidelines for Indexing Foreign Keys

Index foreign keys unless the matching unique or primary key is never updated or deleted.

See Also: *Oracle Database Concepts* for more information about indexing foreign keys

Referential Integrity in a Distributed Database

The declaration of a referential 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: *Oracle Database PL/SQL Language Reference* for more information about triggers that enforce referential integrity

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

For example, assume that the child table is in the SALES database, and the parent table is in the HQ database.

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

When to Use CHECK 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 constraints can provide the necessary checking.

See Also: "Choosing Between CHECK and NOT NULL Constraints" on page 6-13

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 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 or ROWNUM.
- The condition cannot contain the PRIOR operator.
- The condition cannot contain a user-defined SQL function.

See Also:

- *Oracle Database SQL Language Reference* for information about the LEVEL pseudocolumn
- *Oracle Database SQL Language Reference* for information about the ROWNUM pseudocolumn
- *Oracle Database SQL Language Reference* for information about the PRIOR operator (used in hierarchical queries)

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. Ensure that any CHECK constraint that you define is specific enough to enforce the rule.

For example, consider the following CHECK constraint:

```
CHECK (Sal > 0 OR Comm >= 0)
```

At first glance, this rule may be interpreted as "do not allow a row in the employee table unless the employee salary is greater than zero or the employee commission is greater than or equal to zero." But if a row is inserted with a null salary, that row does not violate the CHECK constraint, regardless of whether or not the commission value is valid, because the entire check condition is evaluated as unknown. In this case, you can prevent such violations by placing NOT NULL 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 conditions in *Oracle Database SQL Language 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 Constraints

According to the ANSI/ISO standard, a NOT NULL constraint is an example of a CHECK constraint, where the condition is:

```
CHECK (Column_name IS NOT NULL)
```

Therefore, you can write NOT NULL constraints for a single column using either a NOT NULL constraint or a CHECK constraint. The NOT NULL constraint is easier to use than the CHECK constraint.

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 Constraints

Here are some examples showing how to create simple constraints during the prototype phase of your database design.

Each constraint is given a name in these examples. Naming the constraints prevents the database from creating multiple copies of the same constraint, with different system-generated names, if the DDL is run multiple times.

See Also: *Oracle Database Administrator's Guide* for information on creating and maintaining constraints for a large production database

Example: Defining Constraints with the CREATE TABLE Statement

The following examples of CREATE TABLE statements show the definition of several constraints:

```
CREATE TABLE DeptTab (
    Deptno  NUMBER(3) CONSTRAINT pk_DeptTab_Deptno PRIMARY KEY,
    Dname   VARCHAR2(15),
    Loc     VARCHAR2(15),
    CONSTRAINT u_DeptTab_Dname_Loc UNIQUE (Dname, Loc),
    CONSTRAINT c_DeptTab_Loc
      CHECK (Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));

CREATE TABLE EmpTab (
    Empno    NUMBER(5) CONSTRAINT pk_EmpTab_Empno PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
    Mgr      NUMBER(5) CONSTRAINT r_EmpTab_Mgr REFERENCES EmpTab,
    Hiredate DATE,
    Sal      NUMBER(7,2),
    Comm     NUMBER(5,2),
    Deptno   NUMBER(3) NOT NULL
             CONSTRAINT r_EmpTab_DeptTab REFERENCES DeptTab ON DELETE CASCADE);
```

Example: Defining Constraints with the ALTER TABLE Statement

You can also define constraints using the constraint clause of the ALTER TABLE statement. For example:

```
CREATE UNIQUE INDEX u_DeptTab_Deptno ON DeptTab(Deptno);
ALTER TABLE DepTab
  ADD CONSTRAINT pk_DeptTab_Deptno PRIMARY KEY (Deptno);

ALTER TABLE EmpTab
  ADD CONSTRAINT fk_DeptTab_Deptno FOREIGN KEY (Deptno) REFERENCES DeptTab;
ALTER TABLE EmpTab MODIFY (Ename VARCHAR2(15) NOT NULL);
```

You cannot create a validated constraint on a table if the table already contains rows that violate the constraint.

Privileges Needed to Define 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 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 constraints also require some additional privileges.

See Also: "Privileges Required to Create FOREIGN KEY Constraints" on page 6-20

Naming Constraints

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

Choosing your own name makes error messages for constraint violations more understandable, and prevents the creation of duplicate constraints with different names if the SQL statements are run more than once.

See the previous examples of the CREATE TABLE and ALTER TABLE statements for examples of the CONSTRAINT option of the `constraint` clause. The name of each constraint is included with other information about the constraint in the data dictionary.

See Also: "Viewing Definitions of Constraints" on page 6-21 for examples of static data dictionary views

Enabling and Disabling Constraints

This section explains the mechanisms and procedures for manually enabling and disabling 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.

Topics:

- Why Disable Constraints?
- Creating Enabling Constraints (Default)
- Creating Disabled Constraints
- Enabling Existing Constraints
- Disabling Existing Constraints
- Guidelines for Enabling and Disabling Key Constraints
- Fixing Constraint Exceptions

Why Disable Constraints?

During day-to-day operations, keep constraints enabled. In certain situations, temporarily disabling the constraints of a table makes sense for performance reasons. For example:

- When loading large amounts of data into a table using SQL*Loader
- When performing batch operations that make massive changes to a table (such as changing each employee number by adding 1000 to the existing number)
- When importing or exporting one table at a time

Temporarily turning off constraints can speed up these operations.

Creating Enabling Constraints (Default)

When you define an integrity constraint in a `CREATE TABLE` or `ALTER TABLE` statement, Oracle Database automatically enables the constraint by default. For code clarity, you can explicitly enable the constraint by including the `ENABLE` clause in its definition.

Use this technique when creating tables that start off empty, and are populated a row at a time by individual transactions. In such cases, you want to ensure that data is consistent at all times, and the performance overhead of each DML operation is small.

The following `CREATE TABLE` and `ALTER TABLE` statements both define and enable 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 fails if any existing row of the table violates the integrity constraint. The statement rolls back and the constraint definition is neither stored nor enabled.

See Also: "Fixing Constraint Exceptions" on page 6-17 for more information about rows that violate constraints

Creating Disabled Constraints

The following CREATE TABLE and ALTER TABLE statements both define and disable constraints:

```
CREATE TABLE Emp_tab (
    Empno NUMBER(5) PRIMARY KEY DISABLE);

ALTER TABLE Emp_tab
    ADD PRIMARY KEY (Empno) DISABLE;
```

Use this technique when creating tables that will be loaded with large amounts of data before anybody else accesses them, particularly if you need to cleanse data after loading it, or need to fill empty columns with sequence numbers or parent/child relationships.

An ALTER TABLE statement that defines and disables an constraints never fails, because its rule is not enforced.

Enabling Existing Constraints

To enable an existing constraint, use the ALTER TABLE statement with the ENABLE clause.

Once you have finished cleansing data and filling empty columns, you can enable constraints that were disabled during data loading.

The following statements are examples of statements that enable disabled constraints:

```
ALTER TABLE DeptTab
  ENABLE CONSTRAINT uk_DeptTab_Dname_Loc;

ALTER TABLE DeptTab
    ENABLE PRIMARY KEY
    ENABLE UNIQUE (Dname)
    ENABLE UNIQUE (Loc);
```

An ALTER TABLE statement that attempts to enable an integrity constraint fails if any of the table rows violate the integrity constraint. The statement is rolled back and the constraint is not enabled.

See Also: "Fixing Constraint Exceptions" on page 6-17 for more information about rows that violate constraints

Disabling Existing Constraints

To disable an existing constraint, use the ALTER TABLE statement with the DISABLE clause.

If you need to perform a large load or update when a table already contains data, you can temporarily disable constraints to improve performance of the bulk operation.

The following statements are examples of statements that disable enabled constraints:

```
ALTER TABLE DeptTab
    DISABLE CONSTRAINT uk_DeptTab_Dname_Loc;

ALTER TABLE DeptTab
    DISABLE PRIMARY KEY
    DISABLE UNIQUE (Dname)
    DISABLE UNIQUE (Loc);
```

Guidelines for Enabling and Disabling Key Constraints

When enabling or disabling `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints, be aware of several important issues and prerequisites. `UNIQUE` key and `PRIMARY KEY` constraints are usually managed by the database administrator.

> **See Also:** *Oracle Database Administrator's Guide* and "Managing FOREIGN KEY Constraints" on page 6-20

Fixing Constraint Exceptions

If a row of a table disobeys an integrity constraint, then this row is in violation of the constraint and is called an **exception** to the constraint. If any exceptions exist, then the constraint cannot be enabled. The rows that violate the constraint must be updated or deleted before the constraint can be enabled.

You can identify exceptions for a specific integrity constraint as you try to enable the constraint.

> **See Also:** "Fixing Constraint Exceptions" on page 6-17 for more information on this procedure

When you try to create or enable a constraint, and the statement fails because integrity constraint exceptions exist, the statement is rolled back. You cannot enable the constraint until all exceptions are either updated or deleted. To determine which rows violate the integrity constraint, include the `EXCEPTIONS` option in the `ENABLE` clause of a `CREATE TABLE` or `ALTER TABLE` statement.

> **See Also:** *Oracle Database Administrator's Guide* for more information about responding to constraint exceptions

Altering Constraints

Starting with Oracle8*i*, you can alter the state of an existing constraint with the `MODIFY CONSTRAINT` clause.

> **See Also:** *Oracle Database SQL Language Reference* for information on the parameters you can modify

The following statements show several alternatives for whether the `CHECK` constraint is enforced, and when the constraint checking is done:

```
CREATE TABLE X1Tab (a1 NUMBER CONSTRAINT c_X1Tab_a1 CHECK (a1>3)
 DEFERRABLE DISABLE);
ALTER TABLE X1Tab MODIFY CONSTRAINT c_X1Tab_a1 ENABLE;
ALTER TABLE X1Tab MODIFY CONSTRAINT c_X1Tab_a1 RELY;
ALTER TABLE X1Tab MODIFY CONSTRAINT c_X1Tab_a1 INITIALLY DEFERRED;
ALTER TABLE X1Tab MODIFY CONSTRAINT c_X1Tab_a1 ENABLE NOVALIDATE;
```

The following statements show several alternatives for whether the `NOT NULL` constraint is enforced, and when the checking is done:

```
CREATE TABLE X1Tab (a1 NUMBER CONSTRAINT c_X1Tab_a1
NOT NULL DEFERRABLE INITIALLY DEFERRED NORELY DISABLE);

ALTER TABLE X1Tab ADD CONSTRAINT One_cnstrt UNIQUE(a1)
DEFERRABLE INITIALLY IMMEDIATE RELY USING INDEX PCTFREE = 30
ENABLE VALIDATE;

ALTER TABLE X1Tab MODIFY UNIQUE(a1)
INITIALLY DEFERRED NORELY USING INDEX PCTFREE = 40
ENABLE NOVALIDATE;
```

The following statements show several alternatives for whether the primary key constraint is enforced, and when the checking is done:

```
CREATE TABLE t1 (a1 INT, b1 INT);
ALTER TABLE t1 ADD CONSTRAINT pk_t1_a1 PRIMARY KEY(a1) DISABLE;
ALTER TABLE t1 MODIFY PRIMARY KEY INITIALLY IMMEDIATE
  USING INDEX PCTFREE = 30 ENABLE NOVALIDATE;
ALTER TABLE t1 MODIFY PRIMARY KEY
  USING INDEX PCTFREE = 35 ENABLE;
ALTER TABLE t1 MODIFY PRIMARY KEY ENABLE NOVALIDATE;
```

Renaming Constraints

Because constraint names must be unique, even across multiple schemas, you can encounter problems when you want to clone a table and all its constraints, because the constraint name for the new table conflicts with the one for the original table. Or, you might create a constraint with a default system-generated name, and later realize that you want to give the constraint a name that is easy to remember, so that you can easily enable and disable it.

One of the properties you can alter for a constraint is its name. The following SQL*Plus script finds the system-generated name for a constraint and changes it:

```
prompt Enter table name to find its primary key:
accept table_name
select constraint_name from user_constraints
  where table_name = upper('&table_name.')
  and constraint_type = 'P';

prompt Enter new name for its primary key:
accept new_constraint

set serveroutput on

declare
-- USER_CONSTRAINTS.CONSTRAINT_NAME is declared as VARCHAR2(30).
-- Using %TYPE here protects us if the length changes in a future release.
  constraint_name user_constraints.constraint_name%type;
begin
  select constraint_name into constraint_name from user_constraints
    where table_name = upper('&table_name.')
    and constraint_type = 'P';

  dbms_output.put_line('The primary key for ' || upper('&table_name.') || ' is: '
 || constraint_name);
```

```
  execute immediate
    'alter table &table_name. rename constraint ' || constraint_name ||
    ' to &new_constraint.';
end;
/
```

Dropping 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 statement and the DROP clause. For example, the following statements drop constraints:

```
ALTER TABLE DeptTab DROP UNIQUE (Dname);
ALTER TABLE DeptTab DROP UNIQUE (Loc);
ALTER TABLE EmpTab DROP PRIMARY KEY, DROP CONSTRAINT fk_EmpTab_Dname;
DROP TABLE EmpTab CASCADE CONSTRAINTS;
```

When dropping UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints, be aware of several important issues and prerequisites. UNIQUE and PRIMARY KEY constraints are usually managed by the database administrator.

See Also: *Oracle Database Administrator's Guide* and "Managing FOREIGN KEY Constraints" on page 6-20

Managing FOREIGN KEY Constraints

General information about defining, enabling, disabling, and dropping all types of constraints is given in section "Dropping Constraints" on page 6-19. The present section supplements this information, focusing specifically on issues regarding FOREIGN KEY constraints, which enforce relationships between columns in different tables.

Note: FOREIGN KEY constraints cannot be enabled if the constraint of the referenced primary or unique key is not present or not enabled.

Datatypes and Names for Foreign Key Columns

You must use the same datatype for corresponding columns in the dependent and referenced tables. The column names do not need to match.

Limit on Columns in Composite Foreign Keys

Because foreign keys reference primary and unique keys of the parent table, and PRIMARY KEY and UNIQUE key constraints are enforced using indexes, composite foreign keys are limited to 32 columns.

Foreign Key References Primary Key by Default

If the column list is not included in the REFERENCES option when defining a FOREIGN KEY constraint (single column or composite), then Oracle Database assumes that you intend to reference the primary key of the specified table. Alternatively, you can explicitly specify the column(s) to reference in the parent table within parentheses. Oracle Database automatically checks to verify that this column list references a primary or unique key of the parent table. If it does not, then an informative error is returned.

Privileges Required to Create FOREIGN KEY Constraints

To create a `FOREIGN KEY` constraint, the creator of the constraint must have privileged access to the parent and child tables.

- **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.
- **Child Table** The creator of the referential integrity constraint must have the ability to create tables (that is, the `CREATE TABLE` or `CREATE ANY TABLE` system privilege) or the ability to alter the child table (that is, the `ALTER` object privilege for the child table or the `ALTER ANY TABLE` system privilege).

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

These restrictions allow:

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

Choosing How Foreign Keys Enforce Referential Integrity

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

- **Prevent Delete or Update of Parent Key** The default setting prevents the deletion or update of a parent key if there is a row in the child table that references the key. For example:

  ```
  CREATE TABLE Emp_tab (
  FOREIGN KEY (Deptno) REFERENCES Dept_tab);
  ```

- **Delete Child Rows When Parent Key Deleted** The `ON DELETE CASCADE` action allows parent key data that is referenced from the child table to be deleted, but not updated. When data in the parent key is deleted, all rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the `ON DELETE CASCADE` option in the definition of the `FOREIGN KEY` constraint. For example:

  ```
  CREATE TABLE Emp_tab (
      FOREIGN KEY (Deptno) REFERENCES Dept_tab
          ON DELETE CASCADE);
  ```

- **Set Foreign Keys to Null When Parent Key Deleted** The `ON DELETE SET NULL` action allows data that references the parent key to be deleted, but not updated. When referenced data in the parent key is deleted, all rows in the child table that depend on those parent key values have their foreign keys set to null. To specify this referential action, include the `ON DELETE SET NULL` option in the definition of the `FOREIGN KEY` constraint. For example:

  ```
  CREATE TABLE Emp_tab (
      FOREIGN KEY (Deptno) REFERENCES Dept_tab
          ON DELETE SET NULL);
  ```

Viewing Definitions of Constraints

To find the names of constraints, what columns they affect, and other information to help you manage them, query the static data dictionary views `*_CONSTRAINTS` and `*_CONS_COLUMNS`.

> **See Also:** *Oracle Database Reference* for information on `*_CONSTRAINTS` and `*_CONS_COLUMNS`

Examples of Defining and Viewing Constraints

The following `CREATE TABLE` statements define a number of constraints:

```
CREATE TABLE DeptTab (
   Deptno  NUMBER(3) PRIMARY KEY,
   Dname   VARCHAR2(15),
   Loc     VARCHAR2(15),
   CONSTRAINT uk_DeptTab_Dname_Loc UNIQUE (Dname, Loc),
   CONSTRAINT c_DeptTab_Loc
     CHECK (Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO'))
);

CREATE TABLE EmpTab (
  Empno  NUMBER(5) PRIMARY KEY,
  Ename  VARCHAR2(15) NOT NULL,
  Job    VARCHAR2(10),
  Mgr    NUMBER(5) CONSTRAINT r_EmpTab_Mgr
           REFERENCES Emp_tab ON DELETE CASCADE,
  Hiredate DATE,
  Sal      NUMBER(7,2),
  Comm     NUMBER(5,2),
  Deptno   NUMBER(3) NOT NULL
    CONSTRAINT r_EmpTab_Deptno REFERENCES DeptTab
);
```

Examples:

- Example 1: Listing All of Your Accessible Constraints
- Example 2: Distinguishing NOT NULL Constraints from CHECK Constraints
- Example 3: Listing Column Names that Constitute an Integrity Constraint

Example 1: Listing All of Your Accessible Constraints

The following query lists all constraints defined on all tables accessible to the user:

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME, R_CONSTRAINT_NAME
  FROM USER_CONSTRAINTS;
```

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

```
CONSTRAINT_NAME  C TABLE_NAME  R_CONSTRAINT_NAME
---------------  - ----------  -----------------
SYS_C00275       P DEPTTAB
UK_DEPTTAB_DNAME U DEPTTAB
C_DEPTTAB_LOC    C DEPTTAB
SYS_C00278       C EMPTAB
SYS_C00279       C EMPTAB
SYS_C00280       P EMPTAB
```

```
FK_EMPTAB_MGR    R EMPTAB     SYS_C00280
R_EMPTAB_DEPT    R EMPTAB     SYS_C00275
```

Notice the following:

- Some constraint names are user specified (such as `UK_DEPTTAB_DNAME`), while others are system specified (such as `SYS_C00275`).
- Each constraint type is denoted with a different character in the `CONSTRAINT_TYPE` column. The following table summarizes the characters used for each constraint type.

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

Note: An additional constraint type is indicated by the character "`V`" in the `CONSTRAINT_TYPE` column. This constraint type corresponds to constraints created using the `WITH CHECK OPTION` for views.

Example 2: Distinguishing NOT NULL Constraints from CHECK Constraints

In the previous example, several constraints are listed with a constraint type of `C`. To distinguish which constraints are `NOT NULL` constraints and which are `CHECK` constraints in the `EMPTAB` and `DEPTTAB` tables, submit the following query:

```
SELECT CONSTRAINT_NAME, SEARCH_CONDITION
  FROM USER_CONSTRAINTS
    WHERE (TABLE_NAME = 'DEPTTAB' OR TABLE_NAME = 'EMPTAB') AND
        CONSTRAINT_TYPE = 'C';
```

Considering the example `CREATE TABLE` statements at the beginning of this section, a list similar to this is returned:

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

Notice that the following are explicitly listed in the `SEARCH_CONDITION` column:

- `NOT NULL` constraints
- The conditions for user-defined `CHECK` constraints

Example 3: Listing Column Names that Constitute an Integrity Constraint

The following query lists all columns that constitute the constraints defined on all tables accessible to you, the user:

```
SELECT CONSTRAINT_NAME, TABLE_NAME, COLUMN_NAME
  FROM USER_CONS_COLUMNS;
```

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

```
CONSTRAINT_NAME  TABLE_NAME  COLUMN_NAME
---------------  ----------- ---------------
FK_EMPTAB_DEPT   EMPTAB     DEPTNO
UK_DEPTTAB_DNAME DEPTTAB    DNAME
UK_DEPTTAB_LOC   DEPTTAB    LOC
C_DEPTTAB_LOC    DEPTTAB    LOC
FK_EMPTAB_MGR    EMPTAB     MGR
SYS_C00275       DEPTTAB    DEPTNO
SYS_C00278       EMPTAB     ENAME
SYS_C00279       EMPTAB     DEPTNO
SYS_C00280       EMPTAB     EMPNO
```

Part II

PL/SQL for Application Developers

This part presents information that application developers need about PL/SQL, the Oracle procedural extension of SQL.

Chapters:

See Also: *Oracle Database PL/SQL Language Reference* for a complete description of PL/SQL

7

Coding PL/SQL Subprograms and Packages

This chapter describes some of the procedural capabilities of Oracle Database for application development, including:

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL subprograms
- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL packages

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 subprograms supplied by Oracle to perform data definition language (DDL) statements.

PL/SQL code runs on the server, so using PL/SQL lets you centralize significant parts of your database applications for increased maintainability and security. It also enables you to achieve a significant reduction of network overhead in client/server applications.

Note: Some Oracle tools, such as Oracle Forms, contain a PL/SQL engine that lets you run PL/SQL locally.

You can even use PL/SQL for some database applications in place of 3GL programs that use embedded SQL or Oracle Call Interface (OCI).

PL/SQL program units include:

- Anonymous Blocks
- Stored PL/SQL Program Units
- Triggers

See Also:

- *Oracle Database PL/SQL Language Reference* for syntax and examples of operations on PL/SQL packages
- *Oracle Database PL/SQL Packages and Types Reference* for information about the PL/SQL packages that come with Oracle Database
- *Oracle Database Concepts* for information about dependencies among stored PL/SQL program units

Anonymous Blocks

An anonymous block is a PL/SQL program unit that has no name. 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 invoked 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 example of a PL/SQL anonymous block prints the names of all employees in department 20 in the `hr.employees` table by using the `DBMS_OUTPUT` package:

```
DECLARE
   Last_name    VARCHAR2(10);
   Cursor       c1 IS SELECT last_name
                       FROM employees
                       WHERE department_id = 20;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO Last_name;
      EXIT WHEN c1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(Last_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.

Exceptions let you handle Oracle Database error conditions with PL/SQL program logic. This enables your application to prevent the server from issuing an error that can cause the client application to end. The following anonymous block handles the predefined Oracle Database exception NO_DATA_FOUND (which results 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;
```

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 invoke stored subprograms or to open cursor variables.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for complete information about the DBMS_OUTPUT package
- *Oracle Database PL/SQL Language Reference* and "Handling Run-Time PL/SQL Errors" on page 7-20
- "Cursor Variables" on page 7-17

Stored PL/SQL Program Units

A stored PL/SQL program unit is a subprogram (procedure or function) or package that:

- Has a name.
- Can take parameters, and can return values.
- Is stored in the data dictionary.
- Can be invoked by many users.

If a subprogram belongs to a package, it is called a **package subprogram**; if not, it is called a **standalone subprogram**.

Topics:

Naming Subprograms

Because a subprogram is stored in the database, it must be named. This distinguishes it from other stored subprograms and makes it possible for applications to invoke it. Each publicly-visible subprogram in a schema must have a unique name, and the name must be a legal PL/SQL identifier.

Note: If you plan to invoke a stored subprogram using a stub generated by SQL*Module, then the stored subprogram name must also be a legal identifier in the invoking host 3GL language, such as Ada or C.

Subprogram Parameters

Stored subprograms can take parameters. The following example shows a stored subprogram that is similar to the anonymous block in "Anonymous Blocks" on page 7-2.

Caution: To execute the following, use CREATE OR REPLACE PROCEDURE.

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER) IS
   Emp_name       VARCHAR2(10);
   CURSOR         c1 (Depno NUMBER) IS
                     SELECT Ename FROM Emp_tab
                        WHERE deptno = Depno;
BEGIN
   OPEN c1(Dept_num);
   LOOP
      FETCH c1 INTO Emp_name;
      EXIT WHEN C1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(Emp_name);
   END LOOP;
   CLOSE c1;
END;
```

In the procedure `Get_emp_names`, the department number is an input parameter that is used when the parameterized cursor `c1` is opened.

The formal parameters of a subprogram have three major attributes, described in Table 7–1.

Table 7–1 Attributes of Subprogram Parameters

Parameter Attribute	Description
Name	This must be a legal PL/SQL identifier.
Mode	This indicates whether the parameter is an input-only parameter (`IN`), an output-only parameter (`OUT`), or is both an input and an output parameter (`IN OUT`). If the mode is not specified, then `IN` is assumed.
Datatype	This is a standard PL/SQL datatype.

Topics:

- Parameter Modes
- Parameter Datatypes
- %TYPE and %ROWTYPE Attributes
- Tables and Records
- Default Parameter Values

Parameter Modes Parameter modes define the action of formal parameters. You can use the three parameter modes, `IN` (the default), `OUT`, and `IN OUT`, with any subprogram. Avoid using the `OUT` and `IN OUT` modes with functions. Good programming practice dictates that a function returns a single value and does not change the values of variables that are not local to the subprogram.

Table 7–2 summarizes the information about parameter modes.

Table 7–2 Parameter Modes

IN	OUT	IN OUT
The default.	Must be specified.	Must be specified.

Table 7–2 (Cont.) Parameter Modes

IN	OUT	IN OUT
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 must be assigned a value.
Actual parameter can be a constant, initialized variable, literal, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.

See Also: *Oracle Database PL/SQL Language Reference* for details about parameter modes

Parameter Datatypes The datatype of a formal parameter consists of one of the following:

- An unconstrained type name, such as NUMBER or VARCHAR2.
- A type that is constrained using the %TYPE or %ROWTYPE attributes.

Note: Numerically constrained types such as NUMBER(2) or VARCHAR2(20) are not allowed in a parameter list.

%TYPE and %ROWTYPE Attributes Use the type attributes %TYPE and %ROWTYPE to constrain the parameter. For example, the Get_emp_names procedure specification in "Subprogram Parameters" on page 7-4 can 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 the statement 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 can invoke 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);   -- invoke 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 subprograms. 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 subprogram, 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 can 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 invoke the subprogram. When you do specify the parameter value on the invocation, 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 subprogram. In fact, it is an error to use it.

Creating Subprograms

Use a text editor to write the subprogram. At the beginning of the subprogram, place the following statement:

```
CREATE PROCEDURE Procedure_name AS   ...
```

For example, to use the example in "%TYPE and %ROWTYPE Attributes" on page 7-6, 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). The slash (/) at the end of the code is not part of the code, it only activates the loading of the procedure.

Caution: When developing a new subprogram, it is usually preferable to use the statement `CREATE OR REPLACE PROCEDURE` or `CREATE OR REPLACE FUNCTION`. This statement replaces any previous version of that subprogram in the same schema with the newer version, but without warning.

You can use either the keyword `IS` or `AS` after the subprogram parameter list.

See Also:

- *Oracle Database SQL Language Reference* for the syntax of the `CREATE FUNCTION` statement
- *Oracle Database SQL Language Reference* for the syntax of the `CREATE PROCEDURE` statement

Privileges Needed

To create a subprogram, a package specification, or a package body, you must meet the following prerequisites:

- You must have the `CREATE PROCEDURE` system privilege to create a subprogram or package in your schema, or the `CREATE ANY PROCEDURE` system privilege to create a subprogram or package in another user's schema. In either case, the package body must be created in the same schema as the package.

Note: To create without errors (to compile the subprogram or package successfully) requires the following additional privileges:

- The owner of the subprogram or package must be explicitly granted the necessary object privileges for all objects referenced within the body of the code.
- The owner cannot obtain required privileges through roles.

If the privileges of the owner of a subprogram or package change, then the subprogram must be reauthenticated before it is run. If a necessary privilege to a referenced object is revoked from the owner of the subprogram or package, then the subprogram cannot be run.

The `EXECUTE` privilege on a subprogram gives a user the right to run a subprogram owned by another user. Privileged users run the subprogram under the security domain of the owner of the subprogram. Therefore, users never need to be granted the privileges to the objects referenced by a subprogram. This allows for more disciplined and efficient security strategies with database applications and their users. Furthermore, all subprograms 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 subprograms and packages.

Note: Package creation requires a sort. The user creating the package must be able to create a sort segment in the temporary tablespace with which the user is associated.

Altering Subprograms

To alter a subprogram, 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 subprogram if it exists, then re-creates it as specified.

Caution: The subprogram is dropped without warning.

Dropping Subprograms and Packages

A standalone subprogram, a standalone function, a package body, or an entire package can be dropped using the SQL statements `DROP PROCEDURE`, `DROP FUNCTION`, `DROP PACKAGE BODY`, and `DROP PACKAGE`, respectively. A `DROP PACKAGE` statement drops both the specification and body of a package.

The following statement drops the `Old_sal_raise` procedure in your schema:

```
DROP PROCEDURE Old_sal_raise;
```

Privileges Needed

To drop a subprogram or package, the subprogram or package must be in your schema, or you must have the `DROP ANY PROCEDURE` privilege. An individual subprogram within a package cannot be dropped; the containing package specification and body must be re-created without the subprograms to be dropped.

External Subprograms

A PL/SQL subprogram executing on an Oracle Database instance can invoke an external subprogram written in a third-generation language (3GL). The 3GL subprogram runs in a separate address space from that of the database.

> **See Also:** Chapter 14, "Developing Applications Using Multiple Programming Languages" for information about external subprograms

Cross-Session PL/SQL Function Result Cache

Using the PL/SQL cross-session function result cache can save significant space and time. Each time a **result-cached** PL/SQL function is invoked with different parameter values, those parameters and their result are stored in the cache. Subsequently, when the same function is invoked with the same parameter values, the result is retrieved from the cache, instead of being recomputed. Because the cache is stored in a shared global area (SGA), it is available to any session that runs your application.

If a database object that was used to compute a cached result is updated, the cached result becomes invalid and must be recomputed.

The best candidates for result-caching are functions that are invoked frequently but depend on information that changes infrequently or never.

For more information about the PL/SQL cross-session function result cache, see *Oracle Database PL/SQL Language Reference*.

PL/SQL Packages

A **package** is a collection of related program objects (for example, subprogram, variables, constants, cursors, and exceptions) stored together in the database.

Using packages is an alternative to creating subprograms as standalone schema objects. Packages have many advantages over standalone subprograms. For example, they:

- Let you organize your application development more efficiently.
- Let you grant privileges more efficiently.
- Let you modify package objects without recompiling dependent schema objects.
- Enable Oracle Database to read multiple package objects into memory at once.
- Can contain global variables and cursors that are available to all subprograms in the package.
- Let you **overload** subprograms. Overloading a subprogram means creating multiple subprograms with the same name in the same package, each taking arguments of different number or datatype.

> **See Also:** *Oracle Database PL/SQL Language Reference* for more information about subprogram name overloading

The **specification** part of a package declares the public types, variables, constants, and subprograms that are visible outside the immediate scope of the package. The **body** of a package defines the objects declared in the specification, as well as private objects that are not visible to applications outside the package.

Example of a PL/SQL Package Specification and Body The following example shows a package specification for a package named `Employee_management`. The package contains one stored function and two stored procedures. The body for this package defines the function and the procedures:

```
CREATE PACKAGE BODY Employee_management AS
   FUNCTION Hire_emp (ename 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 invocation of this function.

   BEGIN
      New_empno := Emp_sequence.NEXTVAL;
      INSERT INTO emp VALUES (New_empno, ename, 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 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
         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 Limits

The size limit for PL/SQL stored database objects such as subprograms, triggers, and packages is the size of the **Descriptive Intermediate Attributed Notation for Ada (DIANA)** code in the shared pool in bytes. The Linux and UNIX limit on the size of the flattened DIANA/code size is 64K but the limit might be 32K on desktop platforms.

The most closely related number that a user can access is the `PARSED_SIZE` in the static data dictionary view `*_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.

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 subprograms declared in the package specification.

You can also define private, or local, package subprograms, and variables in a package body. These objects can only be accessed by other subprograms 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 are:

```
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:

- Subprograms declared in the package specification.
- Definitions of cursors declared in the package specification.
- Local subprograms, not declared in the package specification.
- Local variables.

Subprograms, cursors, and variables that are declared in the package specification are **global**. They can be invoked, 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, ensure that each subprogram 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 Needed 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 subprogram. See "Creating Subprograms" on page 7-8 and "Dropping Subprograms and Packages" on page 7-9.

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 subprogram names is desired.

Package Invalidations and Session State

Each session that references a package object has its own instance of the corresponding package, including persistent state for any public and private variables, cursors, and constants. If any of the session's instantiated packages (specification or body) are invalidated, then all package instances in the session are invalidated and recompiled. As a result, the session state is lost for all package instances in the session.

When a package in a given session is invalidated, the session receives the following error the first time it attempts to use any object of the invalid package instance:

```
ORA-04068: existing state of packages has been discarded
```

The second time a session makes such a package call, the package is reinstantiated for the session without error.

Note: For optimal performance, Oracle Database returns this error message only once—each time the package state is discarded.

If you handle this error in your application, ensure that your error handling strategy can accurately handle this error. For example, when a subprogram in one package invokes a subprogram in another package, your application must be aware that the session state is lost for both packages.

In most production environments, DDL operations that can cause invalidations are usually performed during inactive working hours; therefore, this situation might not be a problem for end-user applications. However, if package invalidations are common in your system during working hours, then you might want to code your applications to handle this error when package calls are made.

Packages Supplied with Oracle Database

There are many packages provided with Oracle Database, either to extend the functionality of the database or to give PL/SQL access to SQL features. You can invoke these packages from your application.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for an overview of these Oracle Database packages

Overview of Bulk Binding

Oracle Database uses two engines to run PL/SQL blocks and subprograms. The PL/SQL engine runs procedural statements, while the SQL engine runs SQL statements. During execution, every SQL statement causes a context switch between the two engines, resulting in performance overhead.

Performance can be improved substantially by minimizing the number of context switches required to run a particular block or subprogram. When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required by the block can cause poor performance. Collections include the following:

- Varrays
- Nested tables
- Index-by tables
- Host arrays

Binding is the assignment of values to PL/SQL variables in SQL statements. **Bulk binding** is binding an entire collection at once. Bulk binds pass the entire collection back and forth between the two engines in a single operation.

Typically, using bulk binds improves performance for SQL statements that affect four or more database rows. The more rows affected by a SQL statement, the greater the performance gain from bulk binds.

Note: This section provides an overview of bulk binds to help you decide whether to 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 *Oracle Database PL/SQL Language Reference.*

Parallel DML is disabled with bulk binds.

When to Use Bulk Binds

Consider using bulk binds to improve the performance of the following:

- DML Statements that Reference Collections
- SELECT Statements that Reference Collections
- FOR Loops that Reference Collections and Return DML

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 PLS_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 Return DML 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:** *Oracle Database PL/SQL Language Reference*for more information about triggers

Compiling PL/SQL Subprograms for Native Execution

You can speed up PL/SQL subprograms by compiling them into native code residing in shared libraries.

You can use native compilation with both the supplied Oracle packages and subprograms you write yourself. Subprograms compiled this way work in all server environments, such as the shared server configuration (formerly known as multithreaded server) and Oracle Real Application Clusters (Oracle RAC).

This technique is most effective for computation-intensive subprograms that do not spend much time executing SQL, because it can do little to speed up SQL statements invoked from these subprograms.

With Java, you can use the `ncomp` tool to compile your own packages and classes.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for details on PL/SQL native compilation
> - *Oracle Database Java Developer's Guide* for details on Java native compilation

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 subprograms. A cursor variable can also refer to different cursors in its lifetime.

Additional advantages of cursor variables include the following:

- Encapsulation

  Queries are centralized in the stored subprogram that opens the cursor variable.

- Easy maintenance

  If you need to change the cursor, then you only need to make the change in one place: the stored subprogram. 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 subprogram 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 subprograms.

See Also: *Oracle Database PL/SQL Language Reference* for more information about cursor variables

Topics:

- Declaring and Opening Cursor Variables
- Examples 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:

- *Pro*C/C++ Programmer's Guide*
- *Pro*COBOL Programmer's Guide*
- *Oracle Call Interface Programmer's Guide*

Example 7–1 creates a package, `Emp_data`, that 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. Example 7–2 invokes the `Emp_data` package procedures from a PL/SQL block.

Example 7–1   Package for Fetching Data with 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;
```

Example 7–2   Invoking 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;
```

The power of cursor variables comes from their ability to point to different cursors. Example 7–3 uses a discriminant to open a cursor variable to point to one of two different cursors.

Example 7–3   Cursor Variable with Discriminator

```
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 invoke the Open_cv procedure in Example 7–3 to open the cursor variable and point it to a query on either the Emp_tab table or the Dept_tab table. Example 7–4 uses the cursor variable to fetch data and then uses the ROWTYPE_MISMATCH predefined exception to handle either fetch.

Example 7–4 ROWTYPE_MISMATCH Predefined Exception

```
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

To list compile-time errors, query the static data dictionary view *_ERRORS. From these views, you can retrieve original source code. The error text associated with the compilation of a subprogram is updated when the subprogram is replaced, and it is deleted when the subprogram is dropped.

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
```

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 subprogram, package, or package body:

```
SQL> SHOW ERRORS PROC1
SQL> SHOW ERRORS PROCEDURE PROC1
```

Note: Before issuing the SHOW ERRORS statement, use the SET LINESIZE statement to get long lines on output. The value 132 is usually a good choice. For example:

```
SET LINESIZE 132
```

Assume that you want to create a simple procedure that deletes records from the employee table using SQL*Plus:

```
CREATE OR REPLACE PROCEDURE Fire_emp(Emp_id NUMBER) AS
   BEGIN
      DELETE FROM Emp_tab WHER Empno = Emp_id;
   END
/
```

Notice that the CREATE PROCEDURE statement has two errors: the DELETE statement has an error (the E is absent from WHERE), and the semicolon is missing after END.

After the CREATE PROCEDURE statement is entered and an error is returned, a SHOW ERRORS statement returns the following lines:

```
SHOW ERRORS;

ERRORS FOR PROCEDURE Fire_emp:
LINE/COL       ERROR
-------------- --------------------------------------------
3/27           PL/SQL-00103: Encountered the symbol "EMPNO" wh. . .
5/0            PL/SQL-00103: Encountered the symbol "END" when . . .
2 rows selected.
```

Notice that each line and column number where errors were found is listed by the SHOW ERRORS statement.

See Also:

- *Oracle Database Reference* for more information about the static data dictionary view *_SOURCE.
- *SQL*Plus User's Guide and Reference* for more information about the SHOW ERRORS statement

Handling Run-Time PL/SQL Errors

Oracle Database allows user-defined errors in PL/SQL code to be handled so that user-specified error numbers and messages are returned to the client application. After received, the client application can handle the error based on the user-specified error number and message returned by Oracle Database.

User-specified error messages are returned using the RAISE_APPLICATION_ERROR procedure. For example:

```
RAISE_APPLICATION_ERROR(Error_number, 'text', Keep_error_stack)
```

This procedure stops subprogram execution, rolls back any effects of the subprogram, 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.

Use error number -20000 as a generic number for messages where it is important to relay information to the user, but having a unique error number is not required. `Text` must be a character expression, 2 Kbytes or less (longer messages are ignored). `Keep_error_stack` can be `TRUE` if you want to add the error to any already on the stack, or `FALSE` if you want to replace the existing errors. By default, this option is `FALSE`.

Note: Some of the Oracle Database packages, such as `DBMS_OUTPUT`, `DBMS_DESCRIBE`, and `DBMS_ALERT`, use application error numbers in the range -20000 to -20005. See the descriptions of these packages for more information.

The `RAISE_APPLICATION_ERROR` procedure is often used in exception handlers or in the logic of PL/SQL code. For example, the following exception handler selects the string for the associated user-defined error message and invokes the `RAISE_APPLICATION_ERROR` procedure:

```
...
WHEN NO_DATA_FOUND THEN
   SELECT Error_string INTO Message
   FROM Error_table,
   V$NLS_PARAMETERS V
   WHERE Error_number = -20101 AND Lang = v.value AND
      v.parameter = "NLS_LANGUAGE";
   Raise_application_error(-20101, Message);
...
```

See Also: "Handling Errors in Remote Subprograms" on page 7-23 for information on exception handling when invoking remote subprograms

Topics:

- Declaring Exceptions and Exception Handling Routines
- Unhandled Exceptions
- Handling Errors in Distributed Queries
- Handling Errors in Remote Subprograms

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 invoked. Specific exception handlers can be written to handle any internal or user-defined exception.

Application code can check for a condition that requires special attention using an `IF` statement. If there is an error condition, then two options are available:

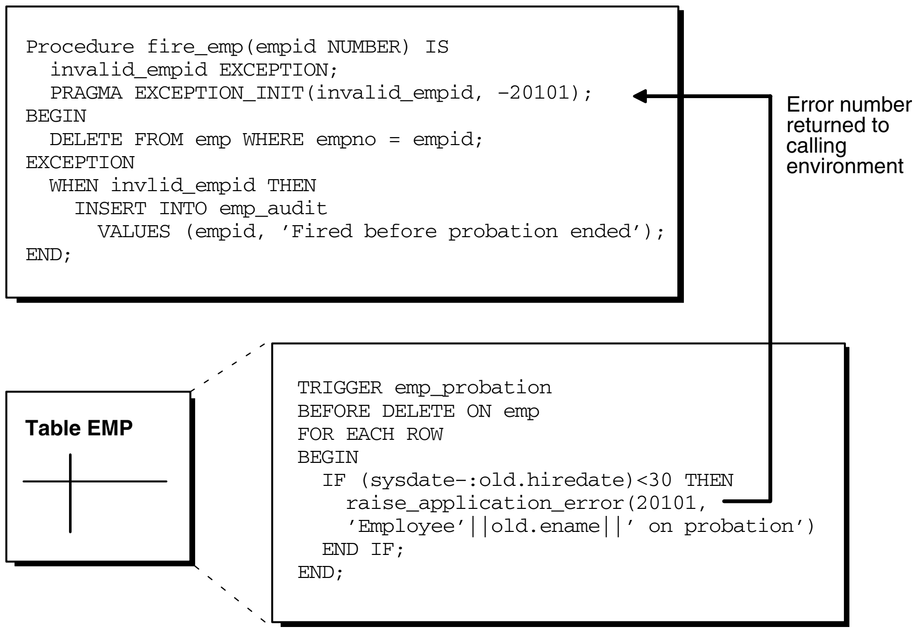
- Enter a `RAISE` statement that names the appropriate exception. A `RAISE` statement stops the execution of the subprogram, and control passes to an exception handler (if any).
- Invoke the `RAISE_APPLICATION_ERROR` procedure to return a user-specified error number and message.

You can also define an exception handler to handle user-specified error messages. For example, Figure 7–1 shows the following:

- An exception and associated exception handler in a subprogram
- 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 invoking environment (in this case, a subprogram), and how that application can define an exception that corresponds to the user-specified error number

Declare a user-defined exception in a subprogram or package body (private exceptions), or in the specification of a package (public exceptions). Define an exception handler in the body of a subprogram (standalone or package).

Figure 7–1 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 invoke the containing program unit. In such an application, only the application program unit invocation 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, modify the database PL/SQL code to handle the exceptions. Your application can also trap for unhandled exceptions when invoking database program units and handle such errors appropriately.

Handling Errors in Distributed Queries

You can use a trigger or a stored subprogram to create a distributed query. This distributed query is decomposed by the local Oracle Database instance into a corresponding number of remote queries, which are sent to the remote nodes for execution. The remote nodes run the queries and send the results back to the local node. The local node then performs any necessary post-processing and returns the results to the user or application.

If a portion of a distributed statement fails, possibly due to a constraint violation, then Oracle Database returns error number `ORA-02055`. Subsequent statements, or subprogram invocations, return error number `ORA-02067` until a rollback or a rollback to savepoint is entered.

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, rollback the entire transaction (or rollback to a savepoint) before allowing the application to proceed.

Handling Errors in Remote Subprograms

When a subprogram 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 subprograms, 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 invoking a remote subprogram, exceptions are also handled by creating a local exception handler. The remote subprogram must return an error number to the local invoking subprogram, which then handles the exception, as shown in the previous example. Because PL/SQL user-defined exceptions always return `ORA-06510` to the local subprogram, these exceptions cannot be handled. All other remote exceptions can be handled in the same manner as local exceptions.

Debugging Stored Subprograms

Compiling a stored subprogram involves fixing any syntax errors in the code. You might need to do additional debugging to ensure that the subprogram works correctly, performs well, and recovers from errors. Such debugging might involve:

- Adding extra output statements to verify execution progress and check data values at certain points within the subprogram.
- Running a separate debugger to analyze execution in greater detail.

Topics:

- PL/Scope
- PL/SQL Hierarchical Profiler
- Oracle JDeveloper
- DBMS_OUTPUT Package
- Privileges for Debugging PL/SQL and Java Stored Subprograms
- Writing Low-Level Debugging Code
- DBMS_DEBUG_JDWP Package
- DBMS_DEBUG Package

PL/Scope

PL/Scope is a compiler-driven tool that collects and organizes data about user-defined identifiers from PL/SQL source code. Because PL/Scope is a compiler-driven tool, you use it through interactive development environments (such as SQL Developer and JDeveloper), rather than directly.

PL/Scope enables the development of powerful and effective PL/Scope source code browsers that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

For more information about PL/Scope, see Chapter 8, "Using PL/Scope".

PL/SQL Hierarchical Profiler

The PL/SQL hierarchical profiler reports the dynamic execution profile of your PL/SQL program, organized by subprogram calls. It accounts for SQL and PL/SQL execution times separately. Each subprogram-level summary in the dynamic execution profile includes information such as number of calls to the subprogram, time spent in the subprogram itself, time spent in the subprogram's subtree (that is, in its descendent subprograms), and detailed parent-children information.

You can browse the generated HTML reports in any browser. The browser's navigational capabilities, combined with well chosen links, provide a powerful way to analyze performance of large applications, improve application performance, and lower development costs.

For a detailed description of PL/SQL hierarchical profiler, see Chapter 9, "Using the PL/SQL Hierarchical Profiler".

Oracle JDeveloper

Recent releases of Oracle JDeveloper have extensive features for debugging PL/SQL, Java, and multi-language programs. You can get Oracle JDeveloper as part of various Oracle product suites. Often, a more recent release is available as a download at `http://www.oracle.com/technology/`.

DBMS_OUTPUT Package

You can also debug stored subprograms and triggers using the Oracle Database package `DBMS_OUTPUT`. Put `PUT` and `PUT_LINE` statements in your code to output the value of variables and expressions to your terminal.

Privileges for Debugging PL/SQL and Java Stored Subprograms

Starting with Oracle Database 10*g*, a new privilege model applies to debugging PL/SQL and Java code running within the database. This model applies whether you are using Oracle JDeveloper, Oracle Developer, or any of the various third-party PL/SQL or Java development environments, and it affects both the `DBMS_DEBUG` and `DBMS_DEBUG_JDWP` APIs.

For a session to connect to a debugger, the effective user at the time of the connect operation must have the `DEBUG CONNECT SESSION` system privilege. This effective user might be the owner of a DR routine involved in making the connect call.

When a debugger becomes connected to a session, the session login user and the currently enabled session-level roles are fixed as the privilege environment for that debugging connection. Any `DEBUG` or `EXECUTE` privileges needed for debugging must be granted to that combination of user and roles.

- To be able to display and change Java public variables or variables declared in a PL/SQL package specification, the debugging connection must be granted either `EXECUTE` or `DEBUG` privilege on the relevant code.
- To be able to either display and change private variables or breakpoint and execute code lines step by step, the debugging connection must be granted `DEBUG` privilege on the relevant code

Caution: The `DEBUG` privilege allows a debugging session to do anything that the subprogram being debugged could have done if that action had been included in its code.

In addition to these privilege requirements, the ability to stop on individual code lines and debugger access to variables are allowed only in code compiled with debug information generated. Use the `PLSQL_DEBUG` parameter and the `DEBUG` keyword on statements such as `ALTER PACKAGE` to control whether the PL/SQL compiler includes debug information in its results. If not, variables are not accessible, and neither stepping nor breakpoints stop on code lines. The PL/SQL compiler never generates debug information for code hidden with the PL/SQL `wrap` utility.

See Also: *Oracle Database PL/SQL Language Reference*, for information about the `wrap` utility

The `DEBUG ANY PROCEDURE` system privilege is equivalent to the `DEBUG` privilege granted on all objects in the database. Objects owned by `SYS` are included if the value of the `O7_DICTIONARY_ACCESSIBILITY` parameter is `TRUE`.

A debug role mechanism is available to carry privileges needed for debugging that are not normally enabled in the session. See the documentation on the `DBMS_DEBUG` and `DBMS_DEBUG_JDWP` packages for details on how to specify a debug role and any necessary related password.

The `JAVADEBUGPRIV` role carries the `DEBUG CONNECT SESSION` and `DEBUG ANY PROCEDURE` privileges. Grant it only with the care those privileges warrant.

Caution: Granting `DEBUG ANY PROCEDURE` privilege, or granting `DEBUG` privilege on any object owned by `SYS`, means granting complete rights to the database.

Writing Low-Level Debugging Code

If you are writing code for part of a debugger, you might need to use packages such as `DBMS_DEBUG_JDWP` or `DBMS_DEBUG`.

DBMS_DEBUG_JDWP Package

The `DBMS_DEBUG_JDWP` package, provided starting with Oracle9*i* Release 2, provides a framework for multi-language debugging that is expected to supersede the `DBMS_DEBUG` package over time. It is especially useful for programs that combine PL/SQL and Java.

DBMS_DEBUG Package

The `DBMS_DEBUG` package, provided starting with Oracle8*i*, implements server-side debuggers and provides a way to debug server-side PL/SQL program units. Several of the debuggers available, such as Oracle Procedure Builder and various third-party vendor solutions, use this API.

See Also:

- *Oracle Procedure Builder Developer's Guide*
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the`DBMS_DEBUG` package and associated privileges
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the`DBMS_OUTPUT` package and associated privileges
- The Oracle JDeveloper documentation for information on using package `DBMS_DEBUG_JDWP`
- *Oracle Database SQL Language Reference* for more details on privileges
- The PL/SQL page at `http://www.oracle.com/technology/` for information about writing low-level debug code

Invoking Stored Subprograms

Note: You might 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;
```

PL/SQL subprograms can be invoked from many different environments. For example:

- From the body of another subprogram
- From the body of a trigger
- Interactively, using an Oracle Database tool
- From within an application (such as a SQL*Forms or a precompiler application)
- From a SQL statement

The following topics include common examples of invoking subprograms from within these environments (except from an SQL statement, which is covered in "Invoking Stored PL/SQL Functions from SQL Statements" on page 7-32). For more information about invoking PL/SQL subprograms, including passing parameters, see *Oracle Database PL/SQL Language Reference*.

Topics:

- Privileges Required to Execute a Subprogram
- Invoking a Subprogram from a Trigger or Another Subprogram
- Invoking a Subprogram Interactively from Oracle Database Tools
- Invoking a Subprogram from a 3GL Application

Privileges Required to Execute a Subprogram

If you are the owner of a standalone subprogram or package, then you can run the standalone subprogram or packaged subprogram, or any public subprogram or packaged subprogram at any time, as described in the previous sections. If you want to run a standalone or packaged subprogram owned by another user, then the following conditions apply:

- You must have the EXECUTE privilege for the standalone subprogram or package containing the subprogram, or you must have the EXECUTE ANY PROCEDURE system privilege. If you are executing a remote subprogram, then you must be granted the EXECUTE privilege or EXECUTE ANY PROCEDURE system privilege directly, not through a role.

- You must include the name of the owner in the invocation. For example:

```
EXECUTE Jward.Fire_emp (1043);
EXECUTE Jward.Hire_fire.Fire_emp (1043);
```

- If the subprogram is a **definer's-rights (DR) subprogram**, then it runs with the privileges of the subprogram owner. The owner must have all the necessary object privileges for any referenced objects.
- If the subprogram is an **invoker's-rights (IR) subprogram**, then it runs with your privileges (as the invoker). In this case, you also need privileges on all referenced objects; that is, all objects accessed by the subprogram through external references that are resolved in your schema. You might hold these privileges directly or through a role. Roles are enabled unless an IR subprogram is invoked directly or indirectly by a DR subprogram.

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

```
CONNECT SYS/password AS SYSDBA;
CREATE USER Jward IDENTIFIED BY Jward;
GRANT CREATE ANY PACKAGE TO Jward;
GRANT CREATE ANY SESSION TO Jward;
GRANT EXECUTE ANY PROCEDURE TO Jward;
CONNECT SCOTT/password AS SYSDBA;
```

Invoking a Subprogram from a Trigger or Another Subprogram

A subprogram or trigger can invoke another stored subprogram. For example, included in the body of one subprogram might be the following line:

```
. . .
Sal_raise(Emp_id, 200);
. . .
```

This line invokes the Sal_raise procedure. Emp_id is a variable within the context of the procedure. Recursive subprogram invocations are allowed within PL/SQL: A subprogram can invoke itself.

Invoking a Subprogram Interactively from Oracle Database Tools

A subprogram can be invoked interactively from an Oracle Database tool, such as SQL*Plus. For example, to invoke a procedure named SAL_RAISE, owned by you, you can use an anonymous PL/SQL block, as follows:

```
BEGIN
   Sal_raise(7369, 200);
END;
```

Note: Interactive tools, such as SQL*Plus, require you to follow these lines with a slash (/) to run the PL/SQL block.

An easier way to run a block is to use the SQL*Plus statement EXECUTE, which wraps BEGIN and END statements around the code you enter. For example:

```
EXECUTE Sal_raise(7369, 200);
```

Some interactive tools allow session variables to be created. For example, when using SQL*Plus, the following statement creates a session variable:

```
VARIABLE Assigned_empno NUMBER
```

After defined, any session variable can be used for the duration of the session. For example, you might run a function and capture the return value using a session variable:

```
EXECUTE :Assigned_empno := Hire_emp('JSMITH', 'President',
   1032, SYSDATE, 5000, NULL, 10);
PRINT Assigned_empno;
ASSIGNED_EMPNO
--------------
          2893
```

See Also:

- *SQL*Plus User's Guide and Reference* for information about the `EXECUTE` command
- Your tools documentation for information about performing similar operations using your development tool

Invoking a Subprogram from a 3GL Application

A 3GL database application, such as a precompiler or an OCI application, can include an invocation to a subprogram within the code of the application.

To run a subprogram within a PL/SQL block in an application, simply invoke the subprogram. The following line within a PL/SQL block invokes 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 subprogram within the code of a precompiler application, you must use the `EXEC` call interface. For example, the following statement invokes the `Fire_emp` procedure in the code of a precompiler application:

```
EXEC SQL EXECUTE
   BEGIN
      Fire_emp1(:Empnum);
   END;
END-EXEC;
```

See Also: For information about invoking PL/SQL subprograms from within 3GL applications:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*

Invoking Remote Subprograms

Invoke remote subprograms using an appropriate database link and the subprogram name. The following SQL*Plus statement runs the procedure `Fire_emp` located in the database and pointed to by the local database link named `BOSTON_SERVER`:

```
EXECUTE fire_emp1@boston_server(1043);
```

See Also: "Handling Errors in Remote Subprograms" on page 7-23 for information on exception handling when invoking remote subprograms

Topics:

- Remote Subprogram Invocations and Parameter Values
- Referencing Remote Objects
- Synonyms for Subprograms and Packages

Remote Subprogram Invocations and Parameter Values

You must explicitly pass values to all remote subprogram 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 subprogram. The following procedure deletes a row from the remote employee table:

```
CREATE OR REPLACE PROCEDURE fire_emp(emp_id NUMBER) IS
BEGIN
    DELETE FROM emp@boston_server WHERE empno = emp_id;
END;
```

The following list explains how to properly invoke remote subprograms, depending on the invoking environment.

- Remote subprograms (standalone and packaged) can be invoked from within a subprogram, an OCI application, or a precompiler application by specifying the remote subprogram name, a database link, and the arguments for the remote subprogram.

  ```
  CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
  BEGIN
    fire_emp1@boston_server(arg);
  END;
  ```

- In the previous example, you can create a synonym for `FIRE_EMP1@BOSTON_SERVER`. This enables you to invoke the remote subprogram from an Oracle Database tool application, such as a SQL*Forms application, as well from within a subprogram, 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 can write a local cover subprogram to invoke the remote subprogram.

  ```
  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 Subprograms and Packages" on page 7-32

Caution: Unlike stored subprograms, which use compile-time binding, run-time binding is used when referencing remote subprograms. The user account to which you connect depends on the database link.

All invocations to remotely stored subprograms are assumed to perform updates; therefore, this type of referencing always requires two-phase commit of that transaction (even if the remote subprogram is read-only). Furthermore, if a transaction that includes a remote subprogram invocation is rolled back, then the work done by the remote subprogram is also rolled back.

A subprogram invoked remotely can usually execute a `COMMIT`, `ROLLBACK`, or `SAVEPOINT` statement, the same as a local subprogram. However, there are some differences in action:

- If the transaction was originated by a database that is not an Oracle database, as might be the case in XA applications, these operations are not allowed in the remote subprogram.
- After doing one of these operations, the remote subprogram cannot start any distributed transactions of its own.
- If the remote subprogram does not commit or roll back its work, the commit is done implicitly when the database link is closed. In the meantime, further invocations to the remote subprogram are not allowed because it is still considered to be performing a transaction.

A **distributed update** modifies data on two or more databases. A distributed update is possible using a subprogram that includes two or more remote updates that access data on different databases. Statements in the construct are sent to the remote databases, 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 subprograms that perform distributed updates.

Pay special attention when using a local subprogram that invokes a remote subprogram. If a timestamp mismatch is found during execution of the local subprogram, then the remote subprogram is not run, and the local subprogram is invalidated.

Synonyms for Subprograms and Packages

Synonyms can be created for standalone subprograms and packages to do the following:

- Hide the identity of the name and owner of a subprogram or package.
- Provide location transparency for remotely stored subprograms (standalone or within a package).

When a privileged user needs to invoke a subprogram, an associated synonym can be used. Because the subprograms defined within a package are not individual objects (the package is the object), synonyms cannot be created for individual subprograms within a package.

For more information about synonyms, see *Oracle Database Concepts*.

Invoking Stored PL/SQL Functions from SQL Statements

To be invoked from a SQL statement, a stored PL/SQL function must be declared either at schema level or in a package specification.

The following SQL statements can invoke stored PL/SQL functions:

- `INSERT`
- `UPDATE`
- `DELETE`
- `SELECT`
- `CALL`

  (`CALL` can also invoke a stored PL/SQL procedure.)

To invoke a PL/SQL subprogram from SQL, you must either own or have `EXECUTE` privileges on the subprogram. 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.

For general information about invoking subprograms, including passing parameters, see *Oracle Database PL/SQL Language Reference*.

Topics:

- Why Invoke Stored PL/SQL Subprograms from SQL Statements?
- Where PL/SQL Functions Can Appear in SQL Statements
- When PL/SQL Functions Can Appear in SQL Expressions
- Controlling Side Effects
- Serially Reusable PL/SQL Packages

Why Invoke Stored PL/SQL Subprograms from SQL Statements?

Invoking PL/SQL subprograms in SQL statements enables you to 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 must otherwise 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 subprogram might also be run in parallel (using the parallel query option).

Where PL/SQL Functions Can Appear in SQL Statements

A PL/SQL function can appear in a SQL statement wherever a built-in SQL function or an expression can appear in a SQL statement. For example, a PL/SQL function can appear in the following:

- Select list of the `SELECT` statement
- Condition of the `WHERE` or `HAVING` clause
- `CONNECT BY`, `START WITH`, `ORDER BY`, or `GROUP BY` clause
- `VALUES` clause of the `INSERT` statement
- `SET` clause of the `UPDATE` statement

A PL/SQL table function (which returns a collection of rows) can appear in a `SELECT` statement in place of the following:

- Column name in the `SELECT` list
- Table name in the `FROM` clause

A PL/SQL function cannot appear in the following contexts, which require unchanging definitions:

- `CHECK` constraint clause of a `CREATE` or `ALTER TABLE` statement
- Default value specification for a column

When PL/SQL Functions Can Appear in SQL Expressions

To be invoked from a SQL expression, a PL/SQL function must satisfy the following additional requirements:

- It must be a row function, not a column (group) function; that is, its argument cannot be an entire column.
- Its formal parameters must be `IN` parameters, not `OUT` or `IN OUT` parameters.
- Its formal parameters and its result value (if any) must have Oracle built-in datatypes (such as `CHAR`, `DATE`, or `NUMBER`), not PL/SQL datatypes (such as `BOOLEAN`, `RECORD`, or `TABLE`).

  There is an exception to this rule: A formal parameter can have a PL/SQL datatype if the corresponding actual parameter is implicitly converted to the datatype of the formal parameter (as in Example 7–6).

The function in Example 7–5 satisfies the preceding requirements. It uses the table `payroll`:

```
CREATE TABLE payroll (srate  NUMBER,
                      orate  NUMBER,
                      acctno NUMBER);
```

Example 7–5 PL/SQL Function that Can Appear in a SQL Expression

```
CREATE FUNCTION gross_pay (emp_id IN NUMBER,
                           st_hrs IN NUMBER DEFAULT 40,
                           ot_hrs IN NUMBER DEFAULT 0)
  RETURN NUMBER IS
  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;
```

In the SQL*Plus script in Example 7–6, the SQL statement `CALL` invokes the PL/SQL function `f1`, whose formal parameter has PL/SQL datatype `PLS_INTEGER`. The `CALL` statement succeeds because the actual parameter, 2, is implicitly converted to the datatype `PLS_INTEGER`. If the actual parameter had a value outside the range of `PLS_INTEGER`, the `CALL` statement would fail.

Example 7–6 PL/SQL Function with Formal Parameter of PL/SQL Datatype, Invoked from a SQL Expression

```
CREATE OR REPLACE FUNCTION f1 (b IN PLS_INTEGER)
  RETURN PLS_INTEGER IS
BEGIN
  RETURN
    CASE
      WHEN b > 0 THEN 1
      WHEN b <= 0 THEN -1
      ELSE NULL
    END;
END f1;
/
VARIABLE x NUMBER
CALL f1(b=>2) INTO :x
/
PRINT x
1
```

Controlling Side Effects

The **purity** of a stored subprogram refers to the side effects of that subprogram on database tables or package variables. Side effects can prevent the parallelization of a query, yield order-dependent (and therefore, indeterminate) results, or require that package state be maintained across user sessions. Various side effects are not allowed when a function is invoked from a SQL query or DML statement.

In releases prior to Oracle8*i*, Oracle Database leveraged the PL/SQL compiler to enforce restrictions during the compilation of a stored subprogram or a SQL statement. Starting with Oracle8*i*, the compile-time restrictions were relaxed, and a smaller set of restrictions are enforced during execution.

This change provides uniform support for stored subprograms written in PL/SQL, Java, and C, and it allows programmers the most flexibility possible.

Topics:

- Restrictions
- Declaring a Function
- Parallel Query and Parallel DML
- PRAGMA RESTRICT_REFERENCES for Backward Compatibility

Restrictions

When a SQL statement is run, checks are made to see if it is logically embedded within the execution of an already running SQL statement. This occurs if the statement is run from a trigger or from a subprogram that was in turn invoked from the already running SQL statement. In these cases, further checks occur to determine if the new SQL statement is safe in the specific context.

The following restrictions are enforced on subprograms:

- A subprogram invoked from a query or DML statement might not end the current transaction, create or rollback to a savepoint, or `ALTER` the system or session.
- A subprogram invoked from a query (`SELECT`) statement or from a parallelized DML statement might not execute a DML statement or otherwise modify the database.
- A subprogram invoked from a DML statement might not read or modify the particular table being modified by that DML statement.

These restrictions apply regardless of what mechanism is used to run the SQL statement inside the subprogram or trigger. For example:

- They apply to a SQL statement invoked from PL/SQL, whether embedded directly in a subprogram or trigger body, run using the native dynamic mechanism (`EXECUTE IMMEDIATE`), or run using the `DBMS_SQL` package.
- They apply to statements embedded in Java with SQLJ syntax or run using JDBC.
- They apply to statements run with OCI using the callback context from within an "external" C function.

You can avoid these restrictions if the execution of the new SQL statement is not logically embedded in the context of the already running statement. PL/SQL's autonomous transactions provide one escape (see "Autonomous Transactions" on page 2-23). Another escape is available using Oracle Call Interface (OCI) from an external C function, if you create a new connection rather than using the handle available from the `OCIExtProcContext` argument.

Declaring a Function

You can use the keywords `DETERMINISTIC` and `PARALLEL_ENABLE` in the syntax for declaring a function. These are optimization hints that inform the query optimizer and other software components about the following:

- Functions that need not be invoked redundantly
- Functions permitted 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 deterministic function depends solely on the values passed into it as arguments and does not reference or modify the contents of package variables or the database or have other side-effects. Such a function produces the same result value for any combination of argument values passed into it.

You place the `DETERMINISTIC` keyword 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;
```

You might place this keyword in the following places:

- On a function defined in a `CREATE FUNCTION` statement

- In a function declaration in a `CREATE PACKAGE` statement
- On a method declaration in a `CREATE TYPE` statement

Do not repeat the keyword on the function or method body in a `CREATE PACKAGE BODY` or `CREATE TYPE BODY` statement.

Certain performance optimizations occur on invocations of functions that are marked `DETERMINISTIC` without any other action being required. The following features require that any function used with them be declared `DETERMINISTIC`:

- Any user-defined function used in a function-based index.
- Any function used in a materialized view, if that view is to qualify for Fast Refresh or is marked `ENABLE QUERY REWRITE`.

The preceding functions features attempt to use previously calculated results rather than invoking the function when it is possible to do so.

It is good programming practice to make functions that fall in the following categories `DETERMINISTIC`:

- Functions used in a `WHERE`, `ORDER BY`, or `GROUP BY` clause
- Functions that `MAP` or `ORDER` methods of a SQL type
- Functions that help determine whether or where a row appears in a result set

Keep the following points in mind when you create `DETERMINISTIC` functions:

- The database cannot recognize if the action of the function is indeed deterministic. If the `DETERMINISTIC` keyword is applied to a function whose action is not truly deterministic, then the result of queries involving that function is unpredictable.
- If you change the semantics of a `DETERMINISTIC` function and recompile it, then existing function-based indexes and materialized views report results for the prior version of the function. Thus, if you change the semantics of a function, you must manually rebuild any dependent function-based indexes and materialized views.

See Also: *Oracle Database SQL Language Reference* for an account of `CREATE FUNCTION` restrictions

Parallel Query and Parallel DML

Oracle Database's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions invoked from a SQL statement that is run in parallel might have a separate copy run in each of these processes, with each copy invoked for only the subset of rows that are handled by that process.

Each process has its own copy of package variables. When parallel execution begins, these are initialized based on the information in the package specification and body as if a new user is logging into the system; the values in package variables are not copied from the original login session. And changes made to package variables are not automatically propagated between the various sessions or back to the original session. Java `STATIC` class attributes are similarly initialized and modified independently in each process. Because a function can use package (or Java `STATIC`) variables to accumulate some value across the various rows it encounters, Oracle Database cannot assume that it is safe to parallelize the execution of all user-defined functions.

For `SELECT` statements in Oracle Database versions prior to 8.1.5, the parallel query optimization allowed functions noted as both `RNPS` and `WNPS` in a `PRAGMA RESTRICT_REFERENCES` declaration to run in parallel. Functions defined with `CREATE FUNCTION` statements had their code implicitly examined to determine if they were pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

See Also: "PRAGMA RESTRICT_REFERENCES for Backward Compatibility" on page 7-38

For DML statements in Oracle Database versions prior to 8.1.5, the parallelization optimization looked to see if a function was noted as having all four of RNDS, WNDS, RNPS and WNPS specified in a PRAGMA RESTRICT_REFERENCES declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a CREATE FUNCTION statement had their code implicitly examined to determine if they were actually pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

Oracle Database versions 8.1.5 and later continue to parallelize those functions that earlier versions recognize as parallelizable. The PARALLEL_ENABLE keyword is the preferred way to mark your code as safe for parallel execution. This keyword is syntactically similar to DETERMINISTIC as described in "Declaring a Function" on page 7-36; 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;
```

A PL/SQL function defined with CREATE FUNCTION might 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 invokes 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 run-time restriction is imposed on functions run in parallel as part of a parallelized DML statement. Such a function is not permitted to in turn execute a DML statement; it is subject to the same restrictions that are enforced on functions that are run inside a query (SELECT) statement.

See Also: "Restrictions" on page 7-35

PRAGMA RESTRICT_REFERENCES for Backward Compatibility

In Oracle Database versions prior to 8.1.5 (Oracle8*i*), programmers used the pragma RESTRICT_REFERENCES to assert the purity level of a subprogram. In subsequent versions, use the hints PARALLEL-ENABLE and DETERMINISTIC, instead, to communicate subprogram purity to Oracle Database.

You can remove RESTRICT_REFERENCES from your code. However, this pragma remains available for backward compatibility in situations where one of the following is true:

- It is impossible or impractical to edit existing code to remove RESTRICT_REFERENCES completely. If you do not remove it from a subprogram S1 that depends on another subprogram S2, then RESTRICT_REFERENCES might also be needed in S2, so that S1 will compile.
- Replacing RESTRICT_REFERENCES in existing code with hints parallel-enable and deterministic would negatively affect the action of

new, dependent code. Use `RESTRICT_REFERENCES` to preserve the action of the existing code.

An existing PL/SQL application can thus continue using the pragma even on new functionality, to ease integration with the existing code. Do not use the pragma in a wholly new application.

If you use the pragma `RESTRICT_REFERENCES`, place it in a package specification, not in a package body. It must follow the declaration of a subprogram, but it need not follow immediately. Only one pragma can reference a given subprogram declaration.

To code the pragma `RESTRICT_REFERENCES`, use the following syntax:

```
PRAGMA RESTRICT_REFERENCES (
    Function_name, WNDS [, WNPS] [, RNDS] [, RNPS] [, TRUST] );
```

Where:

Option	**Description**
`WNDS`	The subprogram writes no database state (does not modify database tables).
`RNDS`	The subprogram reads no database state (does not query database tables).
`WNPS`	The subprogram writes no package state (does not change the values of packaged variables).
`RNPS`	The subprogram reads no package state (does not reference the values of packaged variables).
`TRUST`	The other restrictions listed in the pragma are not enforced; they are simply assumed to be true. This allows easy invocation from functions that have `RESTRICT_REFERENCES` declarations to those that do not.

You can pass the arguments in any order. If any SQL statement inside the subprogram body violates a rule, then you get an error when the statement is parsed.

In the following example, the function `compound` neither reads nor writes database or package state; therefore, you can assert the maximum purity level. Always assert the highest purity level that a subprogram allows. That way, the PL/SQL compiler never rejects the subprogram unnecessarily.

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

```
CREATE TABLE Accts (
 Yrs      NUMBER,
 Amt      NUMBER,
 Acctno   NUMBER,
 Rte      NUMBER);
```

```
CREATE PACKAGE Finance AS  -- package specification
   FUNCTION Compound
         (Years  IN NUMBER,
          Amount IN NUMBER,
          Rate   IN NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES (Compound, WNDS, WNPS, RNDS, RNPS);
END Finance;

CREATE PACKAGE BODY Finance AS  --package body
   FUNCTION Compound
```

```
         (Years  IN NUMBER,
          Amount IN NUMBER,
          Rate   IN NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN Amount * POWER((Rate / 100) + 1, Years);
   END Compound;
                   -- no pragma in package body
END Finance;
```

Later, you might invoke `compound` from a PL/SQL block, as follows:

```
DECLARE
   Interest NUMBER;
   Acct_id NUMBER;
BEGIN
   SELECT Finance.Compound(Yrs, Amt, Rte)  -- function invocation
   INTO   Interest
   FROM   Accounts
   WHERE  Acctno = Acct_id;
```

Topics:

- Using the Keyword TRUST
- Differences between Static and Dynamic SQL Statements
- Overloading Packaged PL/SQL Functions

Using the Keyword TRUST The keyword `TRUST` in the `RESTRICT_REFERENCES` syntax allows easy invocation from functions that have `RESTRICT_REFERENCES` declarations to those that do not. When `TRUST` is present, the restrictions listed in the pragma are not actually enforced, but rather are simply assumed to be true.

When invoking a function 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 invoked, for example on a call specification for a Java method. Then, invocations from PL/SQL to this method complain if the method is less restricted than the invoking subprogram. For example:

```
CREATE OR REPLACE PACKAGE P1 IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
      PRAGMA RESTRICT_REFERENCES(F1,WNDS,TRUST);
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;

   PRAGMA RESTRICT_REFERENCES(F2,WNDS);
END;

CREATE OR REPLACE PACKAGE BODY P1 IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

Here, `F2` can invoke `F1`, as `F1` was declared to be `WNDS`.

The other approach is to mark only the invoker, which might then invoke any subprogram without complaint. For example:

```
CREATE OR REPLACE PACKAGE P1a IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
```

```
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES(F2,WNDS,TRUST);
END;

CREATE OR REPLACE PACKAGE BODY P1a IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

Here, `F2` can invoke `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 invocation of `F1` is allowed, even though there is no `PRAGMA RESTRICT_REFERENCES` for F1.

Differences between Static and Dynamic SQL Statements Static `INSERT`, `UPDATE`, and `DELETE` statements do not violate `RNDS` if these statements do not explicitly read any database states, such as columns of a table. However, dynamic `INSERT`, `UPDATE`, and `DELETE` statements always violate `RNDS`, regardless of whether or not the statements explicitly read database states.

The following `INSERT` violates `RNDS` if it is executed dynamically, but it does not violate `RNDS` if it is executed statically.

```
INSERT INTO my_table values(3, 'SCOTT');
```

The following `UPDATE` always violates `RNDS` statically and dynamically, because it explicitly reads the column `name` of `my_table`.

```
UPDATE my_table SET id=777 WHERE name='SCOTT';
```

Overloading Packaged PL/SQL Functions PL/SQL lets you **overload** packaged (but not standalone) functions: You can use the same name for different functions if their formal parameters differ in number, order, or datatype family.

However, a `RESTRICT_REFERENCES` pragma can apply to only one function declaration. Therefore, a pragma that references the name of overloaded functions always applies to the nearest preceding function declaration.

In this example, the pragma applies to the second declaration of `valid`:

```
CREATE PACKAGE Tests AS
    FUNCTION Valid (x NUMBER) RETURN CHAR;
    FUNCTION Valid (x DATE) RETURN CHAR;
    PRAGMA RESTRICT_REFERENCES (valid, WNDS);
 END;
```

Serially Reusable PL/SQL Packages

PL/SQL packages usually consume user global area (UGA) memory corresponding to the number of package variables and cursors in the package. This limits scalability, because the memory increases linearly with the number of users. The solution is to allow some packages to be marked as `SERIALLY_REUSABLE` (using pragma syntax).

For serially reusable packages, the package global memory is not kept in the UGA for each user; rather, it is kept in a small pool and reused for different users. This means that the global memory for such a package is only used within a unit of work. At the end of that unit of work, the memory can therefore be released to the pool to be reused by another user (after running the initialization code for all the global variables).

The unit of work for serially reusable packages is implicitly a call to the server; for example, an OCI call to the server, or a PL/SQL RPC call from a client to a server, or an RPC call from a server to another server.

Topics:

- Package States
- Why Serially Reusable Packages?
- Syntax of Serially Reusable Packages
- Semantics of Serially Reusable Packages
- Examples of Serially Reusable Packages

Package States

The state of a nonreusable package (one not marked `SERIALLY_REUSABLE`) persists for the lifetime of a session. A package **state** includes global variables, cursors, and so on.

The state of a serially reusable package persists only for the lifetime of a call to the server. On a subsequent call to the server, if a reference is made to the serially reusable package, then Oracle Database creates a new **instantiation** of the serially reusable package and initializes all the global variables to `NULL` or to the default values provided. Any changes made to the serially reusable package state in the previous calls to the server are not visible.

Note: Creating a new instantiation of a serially reusable package on a call to the server does not necessarily imply that Oracle Database allocates memory or configures the instantiation object. Oracle Database looks for an available instantiation work area (which is allocated and configured) for this package in a least-recently used (LRU) pool in the SGA.

At the end of the call to the server, this work area is returned back to the LRU pool. The reason for keeping the pool in the SGA is that the work area can be reused across users who have requests for the same package.

Why Serially Reusable Packages?

Because the state of a nonreusable package persists for the lifetime of the session, this locks up UGA memory for the whole session. In some applications, such as Oracle Office, a log-on session typically exists for days. Applications often need to use certain packages only for short periods of the session. Ideally, such applications could de-instantiate the package state in after they finish using the package (the middle of the session).

`SERIALLY_REUSABLE` packages enable you to design applications that manage memory better for scalability. Package states that matter only for the duration of a call to the server can 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 nonreusable secondary memory is freed (such as memory for collection variables or long `VARCHAR2`s).
  - This package instantiation is returned back to the pool of reusable instantiations kept for this package.
- Serially reusable packages cannot be accessed from database triggers or other PL/SQL subprograms that are invoked from SQL statements. If you try, then Oracle Database generates an error.

Examples of Serially Reusable Packages

- Example 1: How Package Variables Act Across Call Boundaries
- Example 2: How Package Variables Act Across Call Boundaries
- Example 3: Open Cursors in Serially Reusable Packages at Call Boundaries

Example 1: How Package Variables Act Across Call Boundaries This example has a serially reusable package specification (there is no body).

```
CONNECT SCOTT/password

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/password

# first CALL to server
BEGIN
   Sr_pkg.N := 10;
END;

# second CALL to server
BEGIN
   DBMS_OUTPUT.PUT_LINE(Sr_pkg.N);
END;
```

This program prints:

```
5
```

Note: If the package had not had the pragma `SERIALLY_REUSABLE`, the program would have printed '10'.

Example 2: How Package Variables Act Across Call Boundaries This example has both a package specification and package body, which are serially reusable.

```
CONNECT SCOTT/password

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 PLS_INTEGER;
   Num     NUMBER        := 10;
   Str     VARCHAR2(200) := 'default-init-str';
   Str_tab STR_TABLE_TYPE;

    PROCEDURE Print_pkg;
    PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2);
END Sr_pkg;
CREATE OR REPLACE PACKAGE BODY Sr_pkg IS
   -- the body is required to have the pragma because the
  -- specification of this package has the pragma
  PRAGMA SERIALLY_REUSABLE;
   PROCEDURE Print_pkg IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('num: ' || Sr_pkg.Num);
      DBMS_OUTPUT.PUT_LINE('str: ' || Sr_pkg.Str);
      DBMS_OUTPUT.PUT_LINE('number of table elems: ' || Sr_pkg.Str_tab.Count);
      FOR i IN 1..Sr_pkg.Str_tab.Count LOOP
         DBMS_OUTPUT.PUT_LINE(Sr_pkg.Str_tab(i));
      END LOOP;
   END;
   PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2) IS
   BEGIN
   -- init the package globals
      Sr_pkg.Num := N;
      Sr_pkg.Str := V;
      FOR i IN 1..n LOOP
         Sr_pkg.Str_tab(i) := V || ' ' || i;
   END LOOP;
   -- print the package
   Print_pkg;
   END;
```

```
 END Sr_pkg;

SET SERVEROUTPUT ON;

Rem SR package access in a CALL:

BEGIN
   -- initialize and print the package
   DBMS_OUTPUT.PUT_LINE('Initing and printing pkg state..');
   Sr_pkg.Init_and_print_pkg(4, 'abracadabra');
   -- print it in the same call to the server.
   -- we should see the initialized values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the same CALL...');
   Sr_pkg.Print_pkg;
END;

Initing and printing pkg state..
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
Printing package state in the same CALL...
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4

REM SR package access in subsequent CALL:
BEGIN
   -- print the package in the next call to the server.
   -- The package state should be reset to the initial (default) values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the next CALL...');
   Sr_pkg.Print_pkg;
END;
Statement processed.
Printing package state in the next CALL...
num: 10
str: default-init-str
number of table elems: 0
```

Example 3: Open Cursors in Serially Reusable Packages at Call Boundaries This example demonstrates that any open cursors in serially reusable packages get closed automatically at the end of a work boundary (which is a call). Also, in a new call, these cursors need to be opened again.

```
REM  For serially reusable pkg: At the end work boundaries
REM  (which is currently the OCI call boundary) all open
REM  cursors will be closed.
REM
REM  Because the cursor is closed - every time we fetch we
REM  will start at the first row again.

CONNECT SCOTT/password
DROP PACKAGE  Sr_pkg;
DROP TABLE People;
CREATE TABLE People (Name VARCHAR2(20));
INSERT INTO  People  VALUES ('ET');
INSERT INTO  People  VALUES ('RAMBO');
CREATE OR REPLACE PACKAGE Sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   CURSOR C IS SELECT Name FROM People;
END Sr_pkg;
SQL> SET SERVEROUTPUT ON;
SQL>
CREATE OR REPLACE PROCEDURE Fetch_from_cursor IS
Name VARCHAR2(200);
BEGIN
   IF (Sr_pkg.C%ISOPEN) THEN
      DBMS_OUTPUT.PUT_LINE('cursor is already open.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('cursor is closed; opening now.');
      OPEN Sr_pkg.C;
   END IF;
   -- fetching from cursor.
   FETCH sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   FETCH Sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   -- Oops forgot to close the cursor (Sr_pkg.C).
   -- But, because it is a Serially Reusable pkg's cursor,
   -- it will be closed at the end of this CALL to the server.
END;
EXECUTE fetch_from_cursor;
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
```

Returning Large Amounts of Data from a Function

In a data warehousing environment, you might use a PL/SQL function to transform large amounts of data. Perhaps the data is passed through a series of transformations, each performed by a different function. PL/SQL table functions let you perform such transformations without significant memory overhead or the need to store the data in tables between each transformation stage. These functions can accept and return multiple rows, can return rows as they are ready rather than all at once, and can be parallelized.

In this technique:

- The producer function uses the `PIPELINED` keyword in its declaration.
- The producer function uses an `OUT` parameter that is a record, corresponding to a row in the result set.
- As each output record is completed, it is sent to the consumer function using `PIPE ROW`.
- 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 invoked 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 invocation of the aggregate function.

See Also: *Oracle Database Data Cartridge Developer's Guide* for more information about user-defined aggregate functions

8 Using PL/Scope

PL/Scope is a compiler-driven tool that collects data about user-defined identifiers from PL/SQL source code at program-unit compilation time and makes it available in static data dictionary views. The collected data includes information about identifier types, usages (declaration, definition, reference, call, assigment) and the location of each usage in the source code.

PL/Scope enables the development of powerful and effective PL/Scope source code browsers that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

Note: PL/Scope cannot collect data for a PL/SQL program unit whose source code is wrapped. For information about wrapping PL/SQL source code, see *Oracle Database PL/SQL Language Reference*.

Topics:

Specifying Identifier Collection

By default, PL/Scope does not collect data for identifiers in the PL/SQL source program. To have PL/Scope collect data for all identifiers in the PL/SQL source program, including identifiers in package bodies, use the following SQL statement:

```
ALTER SESSION SET PLSCOPE_SETTINGS='IDENTIFIERS:ALL'
```

Note: Collecting all identifiers might generate large amounts of data and slow compile time.

`PLSCOPE_SETTINGS='IDENTIFIERS:ALL'` affects only the PL/SQL code compiled after you specify it. If you compile a PL/SQL program unit with `PLSCOPE_SETTINGS='IDENTIFIERS:NONE'` (the default), PL/Scope does not collect its identifiers, and drops any identifiers that it previously collected for that unit. To have

PL/Scope collect its identifiers, recompile the program unit with PLSCOPE_SETTINGS='IDENTIFIERS:ALL'. To see the value that IDENTIFIERS had when a compilation unit was compiled, see the static data dictionary view *_PLSQL_OBJECT_SETTINGS.

PL/Scope stores the data that it collects in the SYSAUX tablespace. If the SYSAUX tablespace is unavailable, and you compile a program unit with PLSCOPE_SETTINGS='IDENTIFIERS:ALL', PL/Scope does not collect data for the compiled object. The compiler does not issue a warning, but it saves a warning in USER_ERRORS.

How Much Space is PL/Scope Data Using?

Because PL/Scope stores its data in the SYSAUX tablespace, you can use the following query to display the amount of space that the data is using:

```
SELECT SPACE_USAGE_KBYTES FROM V$SYSAUX_OCCUPANTS
  WHERE OCCUPANT_NAME='PL/SCOPE';
```

For information about managing the SYSAUX tablespace and monitoring its occupants, see *Oracle Database Administrator's Guide*.

Viewing PL/Scope Data

To view the data that PL/Scope has collected, you can use any of the following:

- Static Data Dictionary Views
- Demo Tool
- SQL Developer

Static Data Dictionary Views

The static data dictionary views *_IDENTIFIERS display information about PL/Scope identifiers, including their types and usages. For general information about these views, see *Oracle Database Reference*.

Topics:

- Unique Keys
- Context
- Signature

Unique Keys

Each row of a *_IDENTIFIERS view represents a unique usage of an identifier in the PL/SQL program unit. In each of these views, the following are equivalent unique keys within a compilation unit:

- LINE, COL, and USAGE
- USAGE_ID

For the usages in the *_IDENTIFIERS views, see "Usages that PL/Scope Reports" on page 8-6.

Note: An identifier that is passed to a subprogram in `IN OUT` mode has two rows in `*_IDENTIFIERS`: a `REFERENCE` usage (corresponding to `IN`) and an `ASSIGNMENT` usage (corresponding to `OUT`).

Context

Context is useful for discovering relationships between usages. Except for top-level schema object declarations and definitions, every usage of an identifier happens within the context of another usage. For example:

- A local variable declaration happens within the context of a top-level procedure declaration.
- If an identifier is declared as a variable, such as `x VARCHAR2(10)`, the `USAGE_CONTEXT_ID` of the `VARCHAR2` type reference contains the `USAGE_ID` of the `x` declaration, allowing you to associate the variable declaration with its type.

In other words, `USAGE_CONTEXT_ID` is a reflexive foreign key to `USAGE_ID`, as Example 8–1 shows.

Example 8–1 USAGE_CONTEXT_ID and USAGE_ID

```
CONNECT USR/password
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';
/
CREATE PROCEDURE a (p1 IN BOOLEAN) IS
  v PLS_INTEGER;
BEGIN
  v := 42;
  DBMS_OUTPUT.PUT_LINE(v);
  RAISE_APPLICATION_ERROR (-20000, 'Bad');
EXCEPTION
  WHEN Program_Error THEN NULL;
END a;
/
CREATE PROCEDURE b (p2 OUT PLS_INTEGER, p3 IN OUT VARCHAR2) IS
  n NUMBER;
  q BOOLEAN := TRUE;
BEGIN
  FOR j IN 1..5 LOOP
    a(q); a(TRUE); a(TRUE);
    IF j > 2 THEN
       GOTO z;
    END IF;
  END LOOP;
<<z>> DECLARE
  d CONSTANT CHAR(1) := 'X';
  BEGIN
    SELECT COUNT(*) INTO n FROM Dual WHERE Dummy = d;
  END z;
END b;
/
WITH v AS (
  SELECT    Line,
            Col,
            INITCAP(NAME) Name,
            LOWER(TYPE)   Type,
            LOWER(USAGE)  Usage,
            USAGE_ID,
            USAGE_CONTEXT_ID
    FROM USER_IDENTIFIERS
      WHERE Object_Name = 'B'
        AND Object_Type = 'PROCEDURE'
)
SELECT RPAD(LPAD(' ', 2*(Level-1)) ||
                 Name, 20, '.')||' '||
                 RPAD(Type, 20)||
                 RPAD(Usage, 20)
                 IDENTIFIER_USAGE_CONTEXTS
  FROM v
  START WITH USAGE_CONTEXT_ID = 0
  CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
  ORDER SIBLINGS BY Line, Col
/

IDENTIFIER_USAGE_CONTEXTS
-------------------------------------------------------------
B.................. procedure           declaration
 B................. procedure           definition
   P2.............. formal out          declaration
   P3.............. formal in out       declaration
   N............... variable            declaration
   Q............... variable            declaration
     Q............. variable            assignment
   J............... iterator            declaration
     A............. procedure           call
       Q........... variable            reference
     A............. procedure           call
     A............. procedure           call
     J............. iterator            reference
     Z............. label               reference
   Z............... label               declaration
     D............. constant            declaration
       D........... constant            assignment
     N............. variable            assignment
     D............. constant            reference
```

Signature

The signature of an identifier is unique, within and across program units. That is, the signature distinguishes the identifier from other identifiers with the same name, whether they are defined in the same program unit or different program units.

For the program unit in Example 8–2, which has two identifiers named `p`, the static data dictionary view `USER_IDENTIFIERS` has several rows in which `NAME` is `p`, but in these rows, `SIGNATURE` varies. The rows associated with the outer procedure `p` have one signature, and the rows associated with the inner procedure `p` have another signature. If program unit `q` calls procedure `p`, the `USER_IDENTIFIERS` view for `q` has a row in which `NAME` is `p` and `SIGNATURE` is the signature of the outer procedure `p`.

Example 8–2 Program Unit with Two Identifiers Named p

```
CREATE OR REPLACE PROCEDURE p IS
  PROCEDURE p IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inner p');
  END p;
BEGIN
```

```
  DBMS_OUTPUT.PUT_LINE('Outer p');
  p();
END p;
```

Demo Tool

$ORACLE_HOME/plsql/demo/plscopedemo.sql is an HTML-based demo implemented as a PL/SQL Web Application using the PL/SQL Web Toolkit. For more information about PL/SQL Web Applications, see "Implementing PL/SQL Web Applications" on page 10-2.

SQL Developer

PL/Scope is a feature of SQL Developer. For information about using PL/Scope from SQL Developer, see the SQL Developer online documentation.

Identifier Types that PL/Scope Collects

Table 8–1 shows the identifier types that PL/Scope collects, in alphabetical order. The identifier types in Table 8–1 appear in the TYPE column of the *_IDENTIFIER static data dictionary views, which are described in *Oracle Database Reference*.

Note: Identifiers declared in compilation units that were not compiled with PLSCOPE_SETTINGS='IDENTIFIERS:ALL' do not appear in *_IDENTIFIER static data dictionary views.

Table 8–1 Identifier Types that PL/Scope Collects

TYPE Column Value	Comment
ASSOCIATIVE ARRAY	
CONSTANT	
CURSOR	
BFILE DATATYPE BLOB DATATYPE BOOLEAN DATATYPE CHARACTER DATATYPE CLOB DATATYPE DATE DATATYPE INTERVAL DATATYPE NUMBER DATATYPE TIME DATATYPE TIMESTAMP DATATYPE	Each DATATYPE is a base type declared in package STANDARD. In order to collect and view these identifiers, package STANDARD must be compiled with PLSCOPE_SETTINGS='IDENTIFIERS:ALL'.
EXCEPTION	
FORMAL IN FORMAL IN OUT FORMAL OUT	
FUNCTION	
INDEX TABLE	
ITERATOR	An iterator is the index of a FOR loop.
LABEL	A label declaration also acts as a context.
LIBRARY	

Table 8–1 (Cont.) Identifier Types that PL/Scope Collects

TYPE Column Value	Comment
NESTED TABLE	
OBJECT	
OPAQUE	Examples of internal opaque types are ANYDATA and XMLType.
PACKAGE	
PROCEDURE	
RECORD	
REFCURSOR	
SUBTYPE	
SYNONYM	PL/Scope does not resolve the base object name of a synonym. To find the base object name of a synonym, query *_SYNONYMS.
TRIGGER	
UROWID	
VARRAY	
VARIABLE	Can be object attribute, local variable, package variable, or record field.

Usages that PL/Scope Reports

Table 8–2 shows the usages that PL/Scope reports, in alphabetical order. The identifier types in Table 8–2 appear in the USAGE column of the *_IDENTIFIER static data dictionary views, which are described in *Oracle Database Reference*.

Table 8–2 Usages that PL/Scope Reports

USAGE Column Value	Description
ASSIGNMENT	An assignment can be made only to an identifier that can have a value, such as a VARIABLE. Examples of assignments are: ■ Using an identifier to the left of an assignment operator ■ Using an identifier in the INTO clause of a FETCH statement ■ Passing an identifier to a subprogram by reference (OUT mode) ■ Using an identifier as the bind argument in the USING clause of an EXECUTE IMMEDIATE statement in either OUT or IN OUT mode An identifier that is passed to a subprogram in IN OUT mode has both a REFERENCE usage (corresponding to IN) and an ASSIGNMENT usage (corresponding to OUT).
CALL	In the context of PL/Scope, a CALL is an operation that pushes a new call stack; that is: ■ A call to a FUNCTION or PROCEDURE ■ Executing or fetching a cursor identifier (a logical call to SQL) A GOTO statement or raise of an exception is not a CALL, because neither pushes a new call stack.

Table 8–2 (Cont.) Usages that PL/Scope Reports

USAGE Column Value	Description
DECLARATION	A DECLARATION tells the compiler that an identifier exists, and each identifier has exactly one DECLARATION. Each DECLARATION can have an associated datatype. For a loop index declaration, LINE and COL (in *_IDENTIFIERS views) are the line and column of the FOR clause that implicitly declares the loop index. For a label declaration, LINE and COL are the line and column on which the label appears (and is implicitly declared) within the delimiters << and >>.
DEFINITION	A DEFINITION tells the compiler how to implement or use a previously declared identifier. Each of the following types of identifiers has a DEFINITION: ■ EXCEPTION (can have multiple definitions) ■ FUNCTION ■ OBJECT ■ PACKAGE ■ PROCEDURE ■ TRIGGER For a top-level identifier only, the DEFINITION and DECLARATION are in the same place.
REFERENCE	A REFERENCE uses an identifier without changing its value. Examples of references are: ■ Raising an exception identifier ■ Using a type identifier in the declaration of a variable or formal parameter ■ Using a variable identifier whose type contains fields to access a field. For example, in myrecordvar.myfield := 1, a reference is made to myrecordvar, and an assignment is made to myfield. ■ Using a cursor for any purpose except FETCH ■ Passing an identifier to a subprogram by value (IN mode) ■ Using an identifier as the bind argument in the USING clause of an EXECUTE IMMEDIATE statement in either IN or IN OUT mode An identifier that is passed to a subprogram in IN OUT mode has both a REFERENCE usage (corresponding to IN) and an ASSIGNMENT usage (corresponding to OUT).

Sample PL/Scope Session

The sample PL/Scope session uses the following PL/SQL procedure, `example.sql`:

```
CREATE OR REPLACE PACKAGE PACK1 IS
   TYPE r1 is RECORD (rf1 VARCHAR2(10));
   FUNCTION F1(fp1 NUMBER) RETURN NUMBER;
   PROCEDURE P1(pp1 VARCHAR2);
END PACK1;

CREATE OR REPLACE PACKAGE BODY PACK1 IS
   FUNCTION F1(fp1 NUMBER) RETURN NUMBER IS
      a NUMBER := 10;
   BEGIN
      RETURN a;
   END F1;
   PROCEDURE P1(pp1 VARCHAR2) IS
      pr1 r1;
   BEGIN
      pr1.rf1 := pp1;
   END;
END PACK1;
```

In the following sample session, assume that you are logged in as `HR`:

1. Set the session parameter:

   ```
   SQL> ALTER SESSION SET PLSCOPE_SETTINGS='IDENTIFIERS:ALL';
   ```

2. Compile the PL/SQL procedure `example.sql`:

   ```
   SQL> @example.sql
   ```

3. Verify that PL/Scope collected all identifiers for the package body:

   ```
   SQL> SELECT PLSCOPE_SETTINGS
         FROM USER_PLSQL_OBJECT_SETTINGS
         WHERE NAME='PACK1' AND TYPE='PACKAGE BODY'

   PLSCOPE_SETTINGS
   ----------------
   IDENTIFIERS:ALL
   ```

4. Display unique identifiers in `HR` by querying for all `DECLARATION` usages. For example, to see all unique identifiers with name like `%1`, use the following query:

   ```
   SQL> SELECT NAME, SIGNATURE, TYPE
         FROM USER_IDENTIFIERS
         WHERE NAME LIKE '%1' AND USAGE='DECLARATION'
         ORDER BY OBJECT_TYPE, USAGE_ID;

   NAME           SIGNATURE                          TYPE
   ---------------------------------------------------------------
   PACK1          41820FA4D5EF6BE707895178D0C5C4EF   PACKAGE

   R1             EEBB6849DEE31BC77BF186EBAE5D4E2D   RECORD

   RF1            41D70040337349634A7F547BC83517C7   VARIABLE

   F1             EEFCF8352A41F4F264B4EF20D7F63A74   FUNCTION

   FP1            70648EC9E1C3C7FA10C0AE6415FAEC3B   FORMAL IN

   P1             0BE2106B9EFA719D49AF60965EBD69AE   PROCEDURE

   PP1            85B6C0F3BBA39185B00465082322444B   FORMAL IN

   FP1            771368AE41084ADD477DE62A7B1D4278   FORMAL IN

   PP1            D98482491487F39B4CBC8B776130B739   FORMAL IN

   PR1            174C2528B929953F4FE2A43DEBA2B5D0   VARIABLE

   P1             3D1CA191D63523E40E25A72D89424324   FORMAL IN
   ```

The *_IDENTIFIERS static data dictionary views display only basic type names; for example, the TYPE of a local variable or record field is VARIABLE. To determine the exact type of a VARIABLE, you must use its USAGE_CONTEXT_ID.

5. Find all local variables:

```
SQL> SELECT a.NAME variable_name,
            b.NAME context_name,
            a.SIGNATURE
      FROM USER_IDENTIFIERS a, USER_IDENTIFIERS b
      WHERE a.USAGE_CONTEXT_ID = b.USAGE_ID
      AND a.TYPE = 'VARIABLE'
      AND a.USAGE = 'DECLARATION'
      AND a.OBJECT_NAME = 'PACK1'
      AND a.OBJECT_NAME = b.OBJECT_NAME
      AND a.OBJECT_TYPE =  b.OBJECT_TYPE
      AND (b.TYPE = 'FUNCTION' or b.TYPE = 'PROCEDURE')
      ORDER BY a.OBJECT_TYPE, a.USAGE_ID;

VARIABLE_NAME   CONTEXT_NAME   SIGNATURE
----------------------------------------------------------------
A               F1             2268998957D20FACD63493B7A77BC55B
PR1             P1             174C2528B929953F4FE2A43DEBA2B5D0
```

6. Find all usages performed on the local variable A:

```
SQL> SELECT USAGE, USAGE_ID, OBJECT_NAME, OBJECT_TYPE
      FROM USER_IDENTIFIERS
      WHERE SIGNATURE='2268998957D20FACD63493B7A77BC55B'
      ORDER BY OBJECT_TYPE, USAGE_ID;

USAGE         USAGE_ID     OBJECT_NAME     OBJECT_TYPE
-------------------------------------------------------
DECLARATION    4           PACK1           PACKAGE BODY
ASSIGNMENT     5           PACK1           PACKAGE BODY
REFERENCE      6           PACK1           PACKAGE BODY
```

The usages performed on the local identifier A are the identifier declaration (USAGE_ID 6), an assignment (USAGE_ID 8), and a reference (USAGE_ID 9).

7. From the declaration of the local identifier A, find its type:

```
SQL> SELECT a.NAME, a.TYPE
      FROM USER_IDENTIFIERS a, USER_IDENTIFIERS b
      WHERE a.USAGE = 'REFERENCE'
      AND a.USAGE_CONTEXT_ID = b.USAGE_ID
      AND b.USAGE = 'DECLARATION'
      AND b.SIGNATURE = '2268998957D20FACD63493B7A77BC55B'
      AND a.OBJECT_TYPE = b.OBJECT_TYPE
      AND a.OBJECT_NAME = b.OBJECT_NAME;

NAME                    TYPE
--------------------------------
NUMBER DATATYPE         STANDARD
```

Note: This query produces this output only if package STANDARD was compiled with PLSCOPE_SETTINGS='IDENTIFIERS:ALL'. By default, this query returns no identifier data. Please see the 11gR1 release notes for more information on how to compile package STANDARD for PL/Scope.

---

8. Find out where the assignment to local identifier A occurred:

```
SQL> SELECT LINE, COL, OBJECT_NAME, OBJECT_TYPE
      FROM USER_IDENTIFIERS
      WHERE SIGNATURE='666CEC3A2180DF4013CEBE330A8CE747'
      AND USAGE='ASSIGNMENT';

LINE      COL      OBJECT_NAME      OBJECT_TYPE
------------------------------------------------
3         7        PACK1            PACKAGE BODY
```

9 Using the PL/SQL Hierarchical Profiler

You can use the PL/SQL hierarchical profiler to identify bottlenecks and performance-tuning opportunities in PL/SQL applications.

The profiler reports the dynamic execution profile of a PL/SQL program organized by function calls, and accounts for SQL and PL/SQL execution times separately. No special source or compile-time preparation is required; any PL/SQL program can be profiled.

This chapter describes the PL/SQL hierarchical profiler and explains how to use it to collect and analyze profile data for a PL/SQL program.

Topics:

Overview of PL/SQL Hierarchical Profiler

Nonhierarchical (**flat**) profilers record the time that a program spends within each subprogram—the **function time** or **self time** of each subprogram. Function time is helpful, but often inadequate. For example, it is helpful to know that a program spends 40% of its time in the subprogram `INSERT_ORDER`, but it is more helpful to know which subprograms call `INSERT_ORDER` often and the total time the program spends under `INSERT_ORDER` (including its descendant subprograms). Hierarchical profilers provide such information.

The PL/SQL hierarchical profiler does the following:

- Reports the dynamic execution profile of your PL/SQL program, organized by subprogram calls
- Accounts for SQL and PL/SQL execution times separately
- Requires no special source or compile-time preparation
- Stores results in database tables (**hierarchical profiler tables**) for custom report generation by integrated development environment (IDE) tools (such as SQL Developer and third-party tools)
- Provides subprogram-level execution summary information, such as:
  - Number of calls to the subprogram

- Time spent in the subprogram itself (**function time** or **self time**)
- Time spent in the subprogram itself and in its descendent subprograms (**subtree time**)
- Detailed parent-children information, for example:
  - All callers of a given subprogram (parents)
  - All subprograms that a given subprogram called (children)
  - How much time was spent in subprogram *x* when called from *y*
  - How many calls to subprogram *x* were from *y*

The PL/SQL hierarchical profiler is implemented by the `DBMS_HPROF` package and has two components:

- Data collection

  The data collection component is an intrinsic part of the PL/SQL Virtual Machine. The `DBMS_HPROF` package provides APIs to turn hierarchical profiling on and off. The **raw profiler output** is written to a file.

- Analyzer

  The analyzer component processes the raw profiler output and stores the results in hierarchical profiler tables.

Note: To generate simple HTML reports directly from raw profiler output, without using the Analyzer, you can use the `plshprof` command-line utility.

Collecting Profile Data

To collect profile data from your PL/SQL program for the PL/SQL hierarchical profiler, follow these steps:

1. Ensure that you have the following privileges:
   - `EXECUTE` privilege on the `DBMS_HPROF` package
   - `WRITE` privilege on the directory that you specify when you call `DBMS_HPROF.START_PROFILING`
2. Use the `DBMS_HPROF.START_PROFILING` PL/SQL API to start hierarchical profiler data collection in a session.
3. Run your PL/SQL program long enough to get adequate code coverage.

   To get the most accurate measurements of elapsed time, avoid unrelated activity on the system on which your PL/SQL program is running.
4. Use the `DBMS_HPROF.STOP_PROFILING` PL/SQL API to stop hierarchical profiler data collection.

For more information about `DBMS_HPROF.START_PROFILING` and `DBMS_HPROF.STOP_PROFILING`, see *Oracle Database PL/SQL Packages and Types Reference*.

Consider the following PL/SQL procedure, `test`:

```
CREATE OR REPLACE PROCEDURE test IS
  n NUMBER;

  PROCEDURE foo IS
  BEGIN
    SELECT COUNT(*) INTO n FROM EMPLOYEES;
  END foo;

BEGIN  -- test
  FOR i IN 1..3 LOOP
    foo;
  END LOOP;
END test;
/
SHOW ERRORS;
```

The SQL script fragment in Example 9–1 profiles the execution of the PL/SQL procedure `test`. The parameters to `DBMS_HPROF.START_PROFILING` specify that raw profiler output is written to the file `test.trc` in the OS directory, which is mapped to the directory object `PLSHPROF_DIR` (see note following example).

Example 9–1 Profiling a PL/SQL Procedure

```
BEGIN
  -- Start profiling:
  DBMS_HPROF.START_PROFILING('PLSHPROF_DIR', 'test.trc');
END;
/
-- Execute procedure to be profiled:
BEGIN
  test;
END;
/
BEGIN
  -- Stop profiling:
  DBMS_HPROF.STOP_PROFILING;
END;
/
```

Note: A directory object is an alias for a file system pathname. For example, the following `CREATE DIRECTORY` statement creates the directory object `PLSHPROF_DIR` and maps it to the file system directory `/private/plshprof/results`:

```
CONNECT / AS SYSDBA;
CREATE DIRECTORY PLSHPROF_DIR as '/private/plshprof/results';
```

To execute the SQL script fragment in Example 9–1, you must have `READ` and `WRITE` privileges on `PLSHPROF_DIR`. The following `GRANT` statement grants `READ` and `WRITE` privileges on `PLSHPROF_DIR` to `HR`:

```
CONNECT / AS SYSDBA;
GRANT READ, WRITE ON DIRECTORY PLSHPROF_DIR TO HR;
```

For more information about using directory objects, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Understanding Raw Profiler Output

Raw profiler output is intended to be processed by the analyzer component of the PL/SQL hierarchical profiler. However, even without such processing, it provides a simple function-level trace of the program. This topic explains how to understand raw profiler output.

Note: The raw profiler format shown in this chapter is intended only to illustrate conceptual features of raw profiler output. Format specifics are subject to change at each Oracle Database release.

The SQL script fragment in Example 9–1 wrote the following raw profiler output to the file `test.trc`:

```
P#V PLSHPROF Internal Version 1.0
P#! PL/SQL Timer Started
P#C PLSQL."".""."__plsql_vm"
P#X 3
P#C PLSQL."".""."__anonymous_block"
P#X 54
P#C PLSQL."SYS"."DBMS_OUTPUT"::11."GET_LINES"#660bd56a1b1640db #180
P#X 15
P#R
P#X 155
P#R
P#X 2
P#R
P#C PLSQL."".""."__plsql_vm"
P#X 3
P#C PLSQL."".""."__anonymous_block"
P#X 33
P#C PLSQL."HR"."TEST"::7."TEST"#980980e97e42f8ec #1
P#X 4
P#C PLSQL."HR"."TEST"::7."TEST.FOO"#980980e97e42f8ec #4
P#X 37
P#C SQL."HR"."TEST"::7."__static_sql_exec_line6" #6
P#X 182
P#R
P#X 19
P#R
P#X 2
P#C PLSQL."HR"."TEST"::7."TEST.FOO"#980980e97e42f8ec #4
P#X 5
P#C SQL."HR"."TEST"::7."__static_sql_exec_line5" #6
P#X 81
P#R
P#X 3
P#R
P#X 1
P#C PLSQL."HR"."TEST"::7."TEST.FOO"#980980e97e42f8ec #4
P#X 3
P#C SQL."HR"."TEST"::7."__static_sql_exec_line6" #6
P#X 78
P#R
P#X 2
P#R
P#X 1
P#R
P#X 1
P#R
P#X 3
P#R
P#C PLSQL."".""."__plsql_vm"
P#X 3
P#C PLSQL."".""."__anonymous_block"
```

```
P#X 54
P#C PLSQL."SYS"."DBMS_OUTPUT"::11."GET_LINES"#660bd56a1b1640db #180
P#X 14
P#R
P#X 55
P#R
P#X 2
P#R
P#C PLSQL."".""."__plsql_vm"
P#X 3
P#C PLSQL."".""."__anonymous_block"
P#X 29
P#C PLSQL."SYS"."DBMS_HPROF"::11."STOP_PROFILING"#980980e97e42f8ec #53
P#R
P#R
P#R
P#! PL/SQL Timer Stopped

PL/SQL procedure successfully completed.
```

Table 9–1 Raw Profiler Output File Indicators

Indicator	Meaning
`P#V`	`PLSHPROF` banner with version number
`P#C`	Call to a subprogram (call event)
`P#R`	Return from a subprogram (return event)
`P#X`	Elapsed time between preceding and following events
`P#!`	Comment

Call events (`P#C`) and return events (`P#R`) are always properly nested (like matched parentheses). If a called subprogram is exited due to an unhandled exception, the profiler still reports a matching return event.

Each call event (`P#C`) entry in the raw profiler output includes the following information:

- **Namespace** to which the called subprogram belongs

  See "Namespaces of Tracked Subprograms" on page 9-6.
- **Name of the PL/SQL module** in which the called subprogram is defined
- **Type of the PL/SQL module** in which the called subprogram is defined
- **Name of the called subprogram**

  This name might be one of the "Special Function Names" on page 9-6.
- Hexadecimal value that represents an MD5 hash of the **signature** of the called subprogram

  The `DBMS_HPROF.analyze` PL/SQL API (described in "Analyzing Profile Data" on page 9-7) stores the hash value in a hierarchical profiler table, which allows both you and `DBMS_HPROF.analyze` to distinguish between **overloaded subprograms** (subprograms with the same name).
- **Line number** at which the called subprogram is defined in the PL/SQL module

  PL/SQL hierarchical profiler does not measure time spent at individual lines within modules, but you can use line numbers to identify the source locations of subprograms in the module (as IDE/Tools do) and to distinguish between overloaded subprograms.

For example, consider the following entry in the preceding example of raw profiler output:

```
P#C PLSQL."HR"."TEST"::7."TEST.FOO"#980980e97e42f8ec #4
```

The components of the preceding entry have the following meanings:

Component	Meaning
`PLSQL`	`PLSQL` is the **namespace** to which the called subprogram belongs.
`"HR"."TEST"`	`HR.TEST` is the **name of the PL/SQL module** in which the called subprogram is defined.
`7`	7 is the internal enumerator for the **module type** of `HR.TEST`. Examples of module types are procedure, package, and package body.
`"TEST.FOO"`	`TEST.FOO` is the **name of the called subprogram**.
`#980980e97e42f8ec`	`#980980e97e42f8ec` is a hexadecimal value that represents an MD5 hash of the **signature** of `TEST.FOO`.
`#3`	3 is the **line number** in the PL/SQL module `HR.TEST` at which `TEST.FOO` is defined.

Namespaces of Tracked Subprograms

The subprograms that the PL/SQL hierarchical profiler tracks are classified into the namespaces PLSQL and SQL, as follows:

- Namespace PLSQL includes:
  - PL/SQL subprogram calls
  - PL/SQL triggers
  - PL/SQL anonymous blocks
  - Remote subprogram calls
  - Package-initialization blocks
- Namespace SQL includes SQL statements executed from PL/SQL, such as queries, DML statements, DDL statements, and native dynamic SQL statements

Special Function Names

PL/SQL hierarchical profiler tracks certain operations as if they were functions with the names and namespaces shown in Table 9–2.

Table 9–2 Function Names of Operations that the PL/SQL Hierarchical Profiler Tracks

Tracked Operation	Function Name	Namespace
Call to PL/SQL Virtual Machine	`__plsql_vm`	PL/SQL
PL/SQL anonymous block	`__anonymous_block`	PL/SQL
Initialization code in package specification or package body	`__pkg_init`	PL/SQL
Static SQL statement at line *`line#`*	`__static_sql_exec_line`*`line#`*	SQL

Table 9–2 (Cont.) Function Names of Operations that the PL/SQL Hierarchical Profiler

Tracked Operation	Function Name	Namespace
Dynamic SQL statement at line *line#*	`__dyn_sql_exec_line`*line#*	SQL
SQL `FETCH` statement at line *line#*	`__sql_fetch_line`*line#*	SQL

Analyzing Profile Data

The analyzer component of the PL/SQL hierarchical profiler, `DBMS_HPROF.analyze`, processes the raw profiler output and stores the results in the **hierarchical database tables** described in Table 9–3.

Table 9–3 PL/SQL Hierarchical Profiler Database Tables

Table	Description
`DBMSHP_RUNS`	Top-level information for this run of `DBMS_HPROF.analyze`. For column descriptions, see Table 9–4 on page 9-8.
`DBMSHP_FUNCTION_INFO`	Information for each subprogram profiled in this run of `DBMS_HPROF.analyze`. For column descriptions, see Table 9–5 on page 9-9.
`DBMSHP_PARENT_CHILD_INFO`	Parent-child information for each subprogram profiled in this run of `DBMS_HPROF.analyze`. For column descriptions, see Table 9–6 on page 9-8.

Topics:

- Creating Hierarchical Profiler Tables
- Understanding Hierarchical Profiler Tables

Note: To generate simple HTML reports directly from raw profiler output, without using the Analyzer, you can use the `plshprof` command-line utility. For details, see "plshprof Utility" on page 9-13.

Creating Hierarchical Profiler Tables

To create the hierarchical profiler tables in Table 9–3 and the other data structures required for persistently storing profile data, follow these steps:

1. Run the script `dbmshptab.sql` (located in the directory `rdbms/admin`).

   This script creates both the hierarchical profiler tables in Table 9–3 and the other data structures required for persistently storing profile data.

   Note: Running the script `dbmshptab.sql` drops any previously created hierarchical profiler tables.

2. Ensure that you have the following privileges:
   - `EXECUTE` privilege on the `DBMS_HPROF` package
   - `READ` privilege on the directory that `DBMS_HPROF.analyze` specifies
3. Use the PL/SQL API `DBMS_HPROF.analyze` to analyze a single raw profiler output file and store the results in hierarchical profiler tables.

(For an example of a raw profiler output file, see `test.trc` in "Collecting Profile Data" on page 9-2.)

For more information about `DBMS_HPROF.analyze`, see *Oracle Database PL/SQL Packages and Types Reference*.

4. Use the hierarchical profiler tables to generate custom reports.

The call to `DBMS_HPROF.analyze` in Example 9–2 does the following:

- Analyzes the profile data in the raw profiler output file `test.trc` (from "Collecting Profile Data" on page 2), which is in the directory that is mapped to the directory object `PLSHPROF_DIR`, and stores the data in the hierarchical profiler tables in Table 9–3 on page 9-7.
- Returns in the variable `runid` a unique identifier that you can use to query the hierarchical profiler tables in Table 9–3 on page 9-7. (By querying these hierarchical profiler tables, you can produce customized reports.)

Example 9–2 Call to DBMS_HPROF.analyze

```
CONNECT HR/password;
VARIABLE runid NUMBER;
BEGIN
:runid := DBMS_HPROF.analyze(LOCATION=>'PLSHPROF_DIR',
                             FILENAME=>'test.trc'
                             RUN_COMMENT=>'First run of TEST');
END;
/
PRINT :runid;
```

Understanding Hierarchical Profiler Tables

The following topics explain how to use the hierarchical profiler tables in Table 9–3:

- Hierarchical Profiler Database Table Columns
- Distinguishing Between Overloaded Subprograms
- Hierarchical Profiler Tables for Sample PL/SQL Procedure
- Examples of Calls to DBMS_HPROF.analyze with Options

Hierarchical Profiler Database Table Columns

Table 9–4 describes the columns of the hierarchical profiler table `DBMSHP_RUNS`, which contains one row of top-level information for each run of `DBMS_HPROF.analyze`.

The primary key for the hierarchical profiler table `DBMSHP_RUNS` is `RUNID`.

Table 9–4 DBMSHP_RUNS Table Columns

Column Name	Column Datatype	Column Contents
`RUNID`	`NUMBER PRIMARY KEY`	Unique identifier for this run of `DBMS_HPROF.analyze`, generated from `DBMSHP_RUNNUMBER` sequence.
`RUN_TIMESTAMP`	`TIMESTAMP`	Timestamp for this run of `DBMS_HPROF.analyze`.
`RUN_COMMENT`	`VARCHAR2(2047)`	Comment that you provided for this run of `DBMS_HPROF.analyze`.

Table 9–4 (Cont.) DBMSHP_RUNS Table Columns

Column Name	Column Datatype	Column Contents
TOTAL_ELAPSED_TIME	INTEGER	Total elapsed timefor this run of DBMS_HPROF.analyze.

Table 9–5 describes the columns of the hierarchical profiler table DBMSHP_FUNCTION_INFO, which contains one row of information for each subprogram profiled in this run of DBMS_HPROF.analyze. If a subprogram is overloaded, Table 9–5 has a row for each variation of that subprogram. Each variation has its own LINE# and HASH (see "Distinguishing Between Overloaded Subprograms" on page 9-10).

The primary key for the hierarchical profiler table DBMSHP_FUNCTION_INFO is RUNID, SYMBOLID.

Table 9–5 DBMSHP_FUNCTION_INFO Table Columns

Column Name	Column Datatype	Column Contents
RUNID	NUMBER	References RUNID column of DBMSHP_RUNS table. For a description of that column, see Table 9–4.
SYMBOLID	NUMBER	Symbol identifier for subprogram (unique for this run of DBMS_HPROF.analyze).
OWNER	VARCHAR2(32)	Owner of module in which each subprogram is defined (for example, SYS or HR).
MODULE	VARCHAR2(2047)	Module in which subprogram is defined (for example, DBMS_LOB, UTL_HTTP, or MY_PACKAGE).
TYPE	VARCHAR2(32)	Type of module in which subprogram is defined (for example, PACKAGE, PACKAGE_BODY, or PROCEDURE).
FUNCTION	VARCHAR2(4000)	Name of subprogram (not necessarily a function) (for example, INSERT_ORDER, PROCESS_ITEMS, or TEST). This name might be one of the "Special Function Names" on page 9-6. For subprogram B defined within subprogram A, this name is A.B. A recursive call to function X is denoted X@*n*, where *n* is the recursion depth. For example, X@1 is the first recursive call to X.
LINE#	NUMBER	Line number in OWNER.MODULE at which FUNCTION is defined.
HASH	RAW(32)	Hash code for signature of subprogram (unique for this run of DBMS_HPROF.analyze).
NAMESPACE	VARCHAR2(32)	Namespace of subprogram. For details, see "Namespaces of Tracked Subprograms" on page 9-6.
SUBTREE_ELAPSED_TIME	INTEGER	Elapsed time, in microseconds, for subprogram, including time spent in descendant subprograms.

Table 9–5 (Cont.) DBMSHP_FUNCTION_INFO Table Columns

Column Name	Column Datatype	Column Contents
FUNCTION_ELAPSED_TIME	INTEGER	Elapsed time, in microseconds, for subprogram, excluding time spent in descendant subprograms.
CALLS	INTEGER	Number of calls to subprogram.

Table 9–6 describes the columns of the hierarchical profiler table DBMSHP_PARENT_CHILD_INFO, which contains one row of parent-child information for each unique parent-child subprogram combination profiled in this run of DBMS_HPROF.analyze.

Table 9–6 DBMSHP_PARENT_CHILD_INFO_RUNS Table Columns

Column Name	Column Datatype	Column Contents
RUNID	NUMBER	References RUNID column of DBMSHP_FUNCTION_INFO table. For a description of that column, see Table 9–5.
PARENTSYMID	NUMBER	Parent symbol ID. RUNID, PARENTSYMID references DBMSHP_FUNCTION_INFO(RUNID, SYMBOLID).
CHILDSYMID	VARCHAR2(32)	Child symbol ID. RUNID, CHILDSYMID references DBMSHP_FUNCTION_INFO(RUNID, SYMBOLID).
SUBTREE_ELAPSED_TIME	INTEGER	Elapsed time, in microseconds, for RUNID, CHILDSYMID when called from RUNID, PARENTSYMID, including time spent in descendant subprograms.
FUNCTION_ELAPSED_TIME	INTEGER	Elapsed time, in microseconds, for RUNID, CHILDSYMID when called from RUNID, PARENTSYMID, excluding time spent in descendant subprograms.
CALLS	INTEGER	Number of calls to RUNID, CHILDSYMID from RUNID, PARENTSYMID.

Distinguishing Between Overloaded Subprograms

Overloaded subprograms are different subprograms with the same name (see *Oracle Database PL/SQL Language Reference*).

Suppose that a program declares three subprograms named compute—the first at line 50, the second at line 76, and the third at line 100. In the DBMSHP_FUNCTION_INFO table, each of these subprograms has compute in the FUNCTION column. To distinguish between the three subprograms, use either the LINE# column (which has 50 for the first subprogram, 76 for the second, and 100 for the third) or the HASH column (which has a unique value for each subprogram).

In the profile data for two different runs, the LINE# and HASH values for a function might differ. The LINE# value of a function changes if you insert or delete lines before the function definition. The HASH value changes only if the signature of the function changes; for example, if you change the parameter list.

Hierarchical Profiler Tables for Sample PL/SQL Procedure

The hierarchical profiler tables for the PL/SQL procedure `test` in "Collecting Profile Data" on page 9-2 are shown in Example 9–3 through Example 9–5.

Example 9–3 DBMSHP_RUNS Table for Sample PL/SQL Procedure

```
RUNID  RUN_TIMESTAMP                  TOTAL_ELAPSED_TIME  RUN_COMMENT
1      10-APR-06 12.01.56.766743 PM  2637                 First run of TEST
```

Example 9–4 DBMSHP_FUNCTION_INFO Table for Sample PL/SQL Procedure

```
RUNID  SYMBOLID  OWNER  MODULE      TYPE          NAMESPACE  FUNCTION
1      1                                          PLSQL      __anonymous_block
1      2                                          PLSQL      __plsql_vm
1      3         HR     TEST        PROCEDURE     PLSQL      TEST
1      4         HR     TEST        PROCEDURE     PLSQL      TEST.FOO
1      5         SYS    DBMS_HPROF  PACKAGE_BODY  PLSQL      STOP_PROFILING
1      6         HR     TEST        PROCEDURE     SQL        __static_sql_exec_line5

LINE#  CALLS  HASH              SUBTREE_ELAPSED_TIME  FUNCTION_ELAPSED_TIME
0      2      980980E97E42F8EC  2554                  342
0      2      980980E97E42F8EC  2637                  83
1      1      980980E97E42F8EC  2212                  28
3      3      980980E97E42F8EC  2184                  126
57     1      980980E97E42F8EC  0                     0
5      3      980980E97E42F8EC  1998                  1998
```

Example 9–5 DBMSHP_PARENT_CHILD_INFO Table for Sample PL/SQL Procedure

```
RUNID  PARENTSYMID  CHILDSYMID  SUBTREE_ELAPSED_TIME  FUNCTION_ELAPSED_TIME  CALLS
1      2            1           2554                   342                   2
1      1            3           2212                   28                    1
1      3            4           2184                   126                   3
1      1            5           0                      0                     1
1      4            6           1998                   1998                  3
```

Consider the third row of the table `DBMSHP_PARENT_CHILD_INFO` (Example 9–5). The `RUNID` column shows that this row corresponds to the first run. The columns `PARENTSYMID` and `CHILDSYMID` show that the symbol IDs of the parent (caller) and child (called subprogram) are `3` and `4`, respectively. The table `DBMSHP_FUNCTION_INFO` (Example 9–4) shows that for the first run, the symbol IDs `3` and `4` correspond to procedures `TEST` and `TEST.FOO`, respectively. Therefore, the information in this row is about calls from the procedure `TEST` to the procedure `FOO` (defined within `TEST`) in the module `HR.TEST`. This row shows that, when called from the procedure `TEST`, the function time for the procedure `FOO` is 126 microseconds, and the time spent in the `FOO` subtree (including descendants) is 2184 microseconds.

Examples of Calls to DBMS_HPROF.analyze with Options

For an example of a call to `DBMS_HPROF.analyze` without options, see Example 9–2 on page 9-8.

The examples in this topic use the following sample package:

```
CREATE OR REPLACE PACKAGE pkg IS
  PROCEDURE myproc (in IN out NUMBER);
  FUNCTION myfunc (v VARCHAR2) RETURN VARCHAR2;
  FUNCTION myfunc (n PLS_INTEGER) RETURN PLS_INTEGER;
END PACKAGE;
/
```

```
CREATE OR REPLACE PACKAGE BODY pkg IS
  PROCEDURE myproc (in IN out NUMBER) IS
  BEGIN
    n := n + 5;
  END;

  FUNCTION myfunc (v VARCHAR2) RETURN VARCHAR2 IS
    n NUMBER;
  BEGIN
    n := LENGTH(v);
    myproc(n);
    IF n > 20 THEN
      RETURN SUBSTR(v, 1, 20);
    ELSE
      RETURN v || '...';
    END IF;
  END;

  FUNCTION myfunc (n PLS_INTEGER) RETURN PLS_INTEGER IS
    i PLS_INTEGER;
    PROCEDURE myproc (in IN out PLS_INTEGER) IS
    BEGIN
      n := n + 1;
    END;
  BEGIN
    i := n;
    myproc(i);
    RETURN i;
  END;
END pkg;
/
```

In each of the following calls to `DBMS_HRPROF.analyze`, the raw profiler output file, `test.trc`, is in the directory corresponding to the `PLSHPROF_DIR` directory object.

- The following call analyzes only the subtrees rooted at the trace entry `"HR"."PKG"."MYPROC"`:

  ```
  VARIABLE runid NUMBER;
  BEGIN
    :runid := DBMS_HRPROF.analyze('PLSHPROF_DIR', 'test.trc',
                                  trace => '"HR"."PKG"."MYPROC"');
  END;
  ```

- The following call analyzes up to 20 calls to the subtrees rooted at the trace entry `"HR"."PKG"."MYFUNC"`. Because `"HR"."PKG"."MYFUNC"` is an overloaded subprogram, both of its overloads are considered for analysis.

  ```
  VARIABLE runid NUMBER;
  BEGIN
    :runid := DBMS_HRPROF.analyze('PLSHPROF_DIR', 'test.trc',
                                  collect => 20,
                                  trace => '"HR"."PKG"."MYFUNC"');
  END;
  ```

- The following call analyzes the second call to the PL/SQL virtual machine:

  ```
  VARIABLE runid NUMBER;
  BEGIN
    :runid := DBMS_HRPROF.analyze('PLSHPROF_DIR', 'test.trc',
                                  skip => 1, collect => 1,
  ```

```
                                             trace => '"."".""__plsql_vm"');
    END;
```

plshprof Utility

The `plshprof` command-line utility (located in the directory `$ORACLE_HOME/bin/`) generates simple HTML reports from either one or two raw profiler output files. (For an example of a raw profiler output file, see `test.trc` in "Collecting Profile Data" on page 9-2.)

You can browse the generated HTML reports in any browser. The browser's navigational capabilities, combined with well chosen links, provide a powerful way to analyze performance of large applications, improve application performance, and lower development costs.

Topics:

- plshprof Options
- HTML Report from a Single Raw Profiler Output File
- HTML Difference Report from Two Raw Profiler Output Files

plshprof Options

The command to run the `plshprof` utility is:

```
plshprof [option...] profiler_output_filename_1 profiler_output_filename_2
```

Each *`option`* is one of the following:

Option	Description	Default
`-trace` *`symbol`*	Specifies function name of tree root	Not applicable
`-skip` *`count`*	Skips first *`count`* calls. Use only with `-trace` *`symbol`*.	0
`-collect` *`count`*	Collects information for *`count`* calls. Use only with `-trace` *`symbol`*.	1
`-output` *`filename`*	Specifies name of output file	*`symbol`*`.html` or *`tracefile1`*`.html`
`-summary`	Prints only elapsed time	None
`-trace` *`symbol`*	Specifies function name of tree root	Not applicable

Suppose that your raw profiler output file, `test.trc`, is in the current directory. You want to analyze and generate HTML reports, and you want the root file of the HTML report to be named `report.html`. Use the following command (`%` is the prompt):

```
% plshprof -output report test.trc
```

HTML Report from a Single Raw Profiler Output File

To generate a PL/SQL hierarchical profiler HTML report from a single raw profiler output file, use these commands:

```
% cd target_directory
% plshprof -output html_root_filename profiler_output_filename
```

`target_directory` is the directory in which you want the HTML files to be created.

html_root_filename is the name of the root HTML file to be created.

profiler_output_filename is the name of a raw profiler output file.

The preceding `plshprof` command generates a set of HTML files. Start browsing them from *html_root_filename*`.html`.

Topics:

- First Page of Report
- Function-Level Reports
- Module-Level Reports
- Namespace-Level Reports
- Parents and Children Report for a Function

First Page of Report

The first page of an HTML report from a single raw profiler output file includes summary information and hyperlinks to other pages of the report.

Sample First Page

PL/SQL Elapsed Time (microsecs) Analysis

2831 microsecs (elapsed time) & 12 function calls

The PL/SQL Hierarchical Profiler produces a collection of reports that present information derived from the profiler's output log in a variety of formats. The following reports have been found to be the most generally useful as starting points for browsing:

- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

In addition, the following reports are also available:

- Function Elapsed Time (microsecs) Data sorted by Function Name
- Function Elapsed Time (microsecs) Data sorted by Total Descendants Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Mean Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Function Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Descendants Elapsed Time (microsecs)
- Module Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- Module Elapsed Time (microsecs) Data sorted by Module Name
- Module Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

- Namespace Elapsed Time (microsecs) Data sorted by Namespace
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Parents and Children Elapsed Time (microsecs) Data

Function-Level Reports

The function-level reports provide a flat view of the profile information. Each function-level report includes the following information for each function:

- Function time (time spent in the function itself, also called "self time")
- Descendants time (time spent in the descendants of the function)
- Subtree time (time spent in the subtree of the function—function time plus descendants time)
- Number of calls to the function
- Function name

  The function name is hyperlinked to the Parents and Children Report for the function.

Each function-level report is sorted on a particular attribute; for example, function time or subtree time.

The following sample report is sorted in descending order of the total subtree elapsed time for the function, which is why information in the Subtree and Ind% columns is in **bold type**.

Sample Report

Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)

2831 microsecs (elapsed time) & 12 function calls

Subtree	Ind%	Function	Descendant	Ind%	Calls	Ind%	Function Name
2831	**100%**	93	2738	96.7%	2	16.7%	__plsq_vm
2738	**96.7%**	310	2428	85.8%	2	16.7%	__anonymous_block
2428	**85.8%**	15	2413	85.2%	1	8.3%	HR.TEST.TEST (Line 1)
2413	**85.2%**	435	1978	69.9%	3	25.0%	HR.TEST.TEST.FOO (Line 3)
1978	**69.9%**	1978	0	0.0%	3	25.0%	HR.TEST.__static_sql_exec_line5 (Line 5)
0	**0.0%**	0	0	0.0%	1	8.3%	SYS.DBMS_HPROF.STOP_PROFILING (Line 53)

Module-Level Reports

Each module-level report includes the following information for each module (for example, package or type):

- Module time (time spent in the module—sum of the function times of all functions in the module)
- Number of calls to functions in the module

Each module-level report is sorted on a particular attribute; for example, module time or module name.

The following sample report is sorted by module time, which is why information in the Module, Ind%, and Cum% columns is in **bold type**.

Sample Report

Module Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

166878 microsecs (elapsed time) & 1099 function calls

Module	Ind%	Cum%	Calls	Ind%	Module Name
84932	**50.9%**	**50.9%**	6	0.5%	HR.P
67749	**40.6%**	**91.5%**	216	19.7%	SYS.DBMS_LOB
13340	**8.0%**	**99.5%**	660	60.1%	SYS.UTL_FILE
839	**0.5%**	**100%**	214	19.5%	SYS.UTL_RAW
18	**0.0%**	**100%**	2	0.2%	HR.UTILS
0	**0.0%**	**100%**	1	0.1%	SYS.DBMS_HPROF

Namespace-Level Reports

Each namespace-level report includes the following information for each namespace:

- Namespace time (time spent in the namespace—sum of the function times of all functions in the namespace)
- Number of calls to functions in the namespace

Each namespace-level report is sorted on a particular attribute; for example, namespace time or number of calls to functions in the namespace.

The following sample report is sorted by function time, which is why information in the Function, Ind%, and Cum% columns is in **bold type**.

Sample Report

Namespace Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

166878 microsecs (elapsed time) & 1099 function calls

Function	Ind%	Cum%	Calls	Ind%	Namespace
93659	**56.1%**	**56.1%**	1095	99.6%	PLSQL
73219	**43.9%**	**100%**	4	0.4%	SQL

Parents and Children Report for a Function

For each function tracked by the profiler, the Parents and Children Report provides information about parents (functions that call it) and children (functions that it calls). For each parent, the report gives the function's execution profile (subtree time, function time, descendants time, and number of calls). For each child, the report gives the execution profile for the child when called from this function (but not when called from other functions).

The execution profile for a function includes the same information for that function as a function-level report includes for each function (for details, see "Function-Level Reports" on page 9-15).

The following Sample Report is a fragment of a Parents and Children Report that corresponds to a function named `HR.P.UPLOAD`. The first row has the following summary information:

- There are two calls to the function `HR.P.UPLOAD`.
- The total subtree time for the function is 166,860 microseconds—11,713 microseconds (7.0%) in the function itself and 155,147 microseconds (93.0%) in its descendants.

After the row "Parents" are the execution profile rows for the two parents of `HR.P.UPLOAD`, which are `HR.UTILS.COPY_IMAGE` and `HR.UTILS.COPY_FILE`.

The first parent execution profile row, for `HR.UTILS.COPY_IMAGE`, shows the following:

- `HR.UTILS.COPY_IMAGE` calls `HR.P.UPLOAD` once, which is 50% of the number of calls to `HR.P.UPLOAD`.
- The subtree time for `HR.P.UPLOAD` when called from `HR.UTILS.COPY_IMAGE` is 106,325 microseconds, which is 63.7% of the total subtree time for `HR.P.UPLOAD`.
- The function time for `HR.P.UPLOAD` when called from `HR.UTILS.COPY_IMAGE` is 6,434 microseconds, which is 54.9% of the total subtree time for `HR.P.UPLOAD`.

After the row "Children" are the execution profile rows for the children of `HR.P.UPLOAD` when called from `HR.P.UPLOAD`.

The third child execution profile row, for `SYS.UTL_FILE.GET_RAW`, shows the following:

- `HR.P.UPLOAD` calls `SYS.UTL_FILE.GET_RAW` 216 times.
- The subtree time, function time and descendants time for `SYS.UTL_FILE.GET_RAW` when called from `HR.P.UPLOAD` are 12,487 microseconds, 3,969 microseconds, and 8,518 microseconds, respectively.
- Of the total descendants time for `HR.P.UPLOAD` (155,147 microseconds), the child `SYS.UTL_FILE.GET_RAW` is responsible for 12,487 microsecs (8.0%).

Sample Report

HR.P.UPLOAD (Line 3)

Subtree	Ind%	Function	Ind%	Descendant	Ind%	Calls	Ind%	Function Name
166860	**100%**	11713	7.0%	155147	93.0%	2	0.2%	HR.P.UPLOAD (Line 3)
Parents:								
106325	**63.7%**	6434	54.9%	99891	64.4%	1	50.0%	HR.UTILS.COPY_IMAGE (Line 3)
60535	**36.3%**	5279	45.1%	55256	35.6%	1	50.0%	HR.UTILS.COPY_FILE (Line 8))
Children:								
71818	**46.3%**	71818	100%	0	N/A	2	100%	HR.P.__static_sql_exec_line38 (Line 38)
67649	**43.6%**	67649	100%	0	N/A	214	100%	SYS.DBMS_LOB.WRITEAPPEND (Line 926)

Subtree	Ind%	Function	Ind%	Descendant	Ind%	Calls	Ind%	Function Name
12487	**8.0%**	3969	100%	8518	100%	216	100%	SYS.UTL_FILE.GET_RAW (Line 1089)
1401	**0.9%**	1401	100%	0	N/A	2	100%	HR.P.__static_sql_exec_line39 (Line 39)
839	**0.5%**	839	100%	0	N/A	214	100%	SYS.UTL_FILE.GET_RAW (Line 246)
740	**0.5%**	73	100%	667	100%	2	100%	SYS.UTL_FILE.FOPEN (Line 422)
113	**0.1%**	11	100%	102	100%	2	100%	SYS.UTL_FILE.FCLOSE (Line 585)
100	**0.1%**	100	100%	0	N/A	2	100%	SYS.DBMS_LOB.CREATETEMPORARY (Line 536)

HTML Difference Report from Two Raw Profiler Output Files

To generate a PL/SQL hierarchical profiler HTML difference report from two raw profiler output files, use these commands:

```
% cd target_directory
% plshprof -output html_root_filename profiler_output_filename_1 profiler_output_filename_2
```

`target_directory` is the directory in which you want the HTML files to be created.

`html_root_filename` is the name of the root HTML file to be created.

`profiler_output_filename_1` and *`profiler_output_filename_2`* are the names of raw profiler output files.

The preceding `plshprof` command generates a set of HTML files. Start browsing them from *`html_root_filename`*`.html`.

Topics:

- Difference Report Conventions
- First Page of Difference Report
- Function-Level Difference Reports
- Module-Level Difference Reports
- Namespace-Level Difference Reports
- Parents and Children Difference Report for a Function

Difference Report Conventions

Difference reports use the following conventions:

- In a report title, **Delta** means **difference**, or **change**.
- A **positive value** indicates that the number increased (**regressed**) from the first run to the second run.
- A **negative value** for a difference indicates that the number decreased (**improved**) from the first run to the second run.

- The symbol **#** after a function name means that the function was called in only one of the two runs.

First Page of Difference Report

The first page of an HTML difference report from two raw profiler output files includes summary information and hyperlinks to other pages of the report.

Sample First Page

PL/SQL Elapsed Time (microsecs) Analysis – Summary Page

This analysis finds a net **regression** of **2709589** microsecs (elapsed time) or **80%** (**3393719** versus **6103308**). Here is a summary of the 7 most important individual function regressions and improvements:

Regressions: 3399382 microsecs (elapsed time)

Function	Rel%	Ind%	Calls	Rel%	Function Name
2075627	**+941%**	**61.1%**	0		HR.P.G (Line 35)
1101384	**+54.6%**	**32.4%**	5	+55.6%	HR.P.H (Line 18)
222371		**6.5%**	1		HR.P.J (Line 10)

Improvements: 689793 microsecs (elapsed time)

Function	Rel%	Ind%	Calls	Rel%	Function Name
-467051	**-50.0%**	**67.7%**	-2	-50.0%	HR.P.F (Line 25)
-222737		**32.3%**	-1		HR.P.I (Line 2)#
-5	**-21.7%**	**0.0%**	0		HR.P.TEST (Line 46)

The PL/SQL Timing Analyzer produces a collection of reports that present information derived from the profiler's output logs in a variety of formats. The following reports have been found to be the most generally useful as starting points for browsing:

- Function Elapsed Time (microsecs) Data for Performance Regressions
- Function Elapsed Time (microsecs) Data for Performance Improvements

In addition, the following reports are also available:

- Function Elapsed Time (microsecs) Data sorted by Function Name
- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs) Delta
- Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs) Delta
- Function Elapsed Time (microsecs) Data sorted by Total Descendants Elapsed Time (microsecs) Delta
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count Delta
- Module Elapsed Time (microsecs) Data sorted by Module Name
- Module Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs) Delta

- Module Elapsed Time (microsecs) Data sorted by Total Function Call Count Delta
- Namespace Elapsed Time (microsecs) Data sorted by Namespace
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Call Count
- File Elapsed Time (microsecs) Data Comparison with Parents and Children

Function-Level Difference Reports

Each function-level difference report includes, for each function, the change in the following from the first run to the second run:

- Function time (time spent in the function itself, also called "self time")
- Descendants time (time spent in the descendants of the function)
- Subtree time (time spent in the subtree of the function—function time plus descendants time)
- Number of calls to the function
- Mean function time

  The mean function time is the function time divided by number of calls to the function.
- Function name

  The function name is hyperlinked to the Parents and Children Difference Report for the function.

The report in Sample Report 1 shows the difference information for all functions that performed better in the first run than they did in the second run. Note the following:

- For `HR.P.G`, the function time increased by 2,075,627 microseconds (941%), which accounts for 61.1% of all regressions.
- For `HR.P.H`, the function time and number of calls increased by 1,101,384 microseconds (54.6%) and 5 (55.6%), respectively, but the mean function time improved by 1,346 microseconds (-0.6%).
- `HR.P.J` was called in only one of the two runs.

Sample Report 1

Function Elapsed Time (microsecs) Data for Performance Regressions

Subtree	Function	Rel%	Ind%	Cum%	Descendant	Calls	Rel%	Mean Function	Rel%	Function Name
4075787	**2075627**	**+941%**	**61.1%**	61.1%	2000160	0		2075627	+941%	HR.P.G (Line 35)
1101384	**1101384**	**+54.6%**	**32.4%**	93.5%	0	5	+55.6%	-1346	-0.6%	HR.P.H (Line 18)
222371	**222371**		6.5%	100%	0	1				HR.P.J (Line 10)#

The report in Sample Report 2 shows the difference information for all functions that performed better in the second run than they did in the first run.

Sample Report 2

Function Elapsed Time (microsecs) Data for Performance Improvements

Subtree	Function	Rel%	Ind%	Cum%	Descendant	Calls	Rel%	Mean Function	Rel%	Function Name
-1365827	**-467051**	**-50.0%**	**67.7%**	67.7%	-898776	-2	-50.0%	-32	0.0%	HR.P.F (Line 25)
-222737	**-222737**		**32.3%**	100%	0	-1				HR.P.I (Line 2)
2709589	**-5**	**-21.7%**	**0.0%**	100%	2709594	0		-5	-20.8	HR.P.TEST (Line 46)#

The report in Sample Report 3 summarizes the difference information for all functions.

Sample Report 3

Function Elapsed Time (microsecs) Data sorted by Total Function Call Count Delta

Subtree	Function	Rel%	Ind%	Descendant	Calls	Rel%	Mean Function	Rel%	Function Name
1101384	1101384	+54.6%	32.4%	0	**5**	**+55.6%**	-1346	-0.6%	HR.P.H (Line 18)
-1365827	-467051	+50.0%	67.7%	-898776	**-2**	**-50.0%**	-32	-0.0%	HR.P.F (Line 25)
-222377	-222377		32.3%	0	**-1**				HR.P.I (Line 2)#
222371	222371		6.5%	0	**1**				HR.P.J(Line 10)#
4075787	2075627	+941%	61.1%	2000160	**0**		2075627	+941%	HR.P.G (Line 35)
2709589	-5	-21.7%	0.0%	2709594	**0**		-5	-20.8%	HR.P.TEST (Line 46)
0	0			0	**0**				SYS.DBMS_HPROF.STOP_ PROFILING (Line 53)

Module-Level Difference Reports

Each module-level report includes, for each module, the change in the following from the first run to the second run:

- Module time (time spent in the module—sum of the function times of all functions in the module)
- Number of calls to functions in the module

Sample Report

Module Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs) Delta

Module	Calls	Module Name
2709589	3	HR.P
0	0	SYS.DBMS_HPROF

Namespace-Level Difference Reports

Each namespace-level report includes, for each namespace, the change in the following from the first run to the second run:

- Namespace time (time spent in the namespace—sum of the function times of all functions in the namespace)
- Number of calls to functions in the namespace

Sample Report

Namespace Elapsed Time (microsecs) Data sorted by Namespace

Function	Calls	Namespace
2709589	3	**PLSQL**

Parents and Children Difference Report for a Function

The Parents and Children Difference Report for a function shows changes in the execution profiles of the following from the first run to the second run:

- Parents (functions that call the function)
- Children (functions that the function calls)

  Execution profiles for children include only information from when this function calls them, not for when other functions call them.

The execution profile for a function includes the following information:

- Function time (time spent in the function itself, also called "self time")
- Descendants time (time spent in the descendants of the function)
- Subtree time (time spent in the subtree of the function—function time plus descendants time)
- Number of calls to the function
- Function name

The following example is a fragment of a Parents and Children Difference Report that corresponds to a function named `HR.P.X`.

The first row, a summary of the difference between the first and second runs, shows regression: function time increased by 1,094,099 microseconds (probably because the function was called five more times).

The "Parents" rows show that `HR.P.G` called `HR.P.X` nine more times in the second run than it did in the first run, while `HR.P.F` called it four fewer times.

The "Children" rows show that `HR.P.X` called each child five more times in the second run than it did in the first run.

Sample Report

HR.P.X (Line 11)

Subtree	Function	Descendant	Calls	Function Name
3322196	1094099	2228097	5	**HR.P.X (Line 11)**
Parents:				
6037490	1993169	4044321	9	**HR.P.G (Line 38)**
-2715294	-899070	-1816224	-4	**HR.P.F (Line 28)**
Children:				
1125489	1125489	0	5	**HR.P.J (Line 10)**
1102608	1102608	0	5	**HR.P.I (Line 2)**

The Parents and Children Difference Report for a function is accompanied by a Function Comparison Report, which shows the execution profile of the function for the first and second runs and the difference between them. The following example is the Function Comparison Report for the function `HR.P.X`.

Sample Report

Elapsed Time (microsecs) for HR.P.X (Line 11) (20.1% of total regression)

HR.P.X (Line 11)	First Trace	Ind%	Second Trace	Ind%	Diff	Diff%
Function Elapsed Time (microsecs)	1999509	26.9%	3093608	24.9%	1094099	+54.7%
Descendants Elapsed Time (microsecs)	4095943	55.1%	6324040	50.9%	2228097	+54.4%
Subtree Elapsed Time (microsecs)	6095452	81.9%	9417648	75.7%	3322196	+54.5%
Function Calls	9	25.0%	14	28.6%	5	+55.6%
Mean Function Elapsed Time (microsecs)	222167.7		220972.0		-1195.7	-0.5%
Mean Descendants Elapsed Time (microsecs)	455104.8		451717.1		-3387.6	-0.7%
Mean Subtree Elapsed Time (microsecs)	677272.4		672689.1		-4583.3	-0.7%

10

Developing PL/SQL Web Applications

This chapter explains how to develop PL/SQL Web applications, which let you make your database available on the intranet.

Topics:

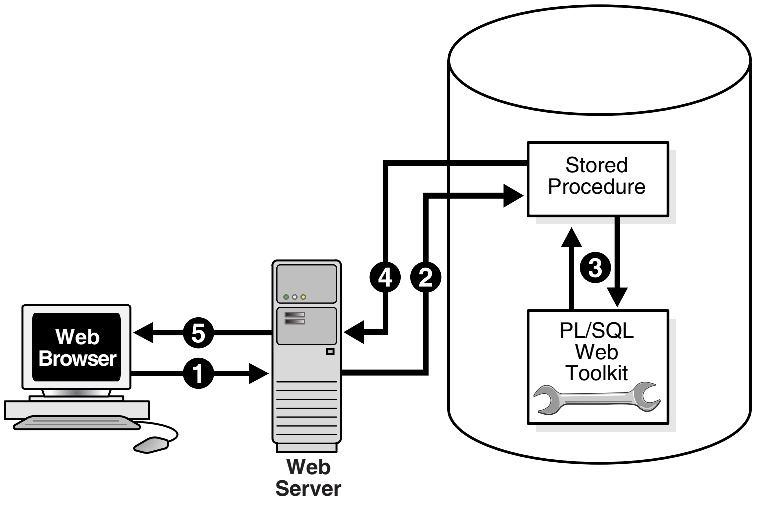
Overview of PL/SQL Web Applications

Typically, a Web application written in PL/SQL is a set of stored subprograms that interact with Web browsers through HTTP. A set of interlinked, dynamically generated HTML pages forms the user interface of a web application.

The program flow of a PL/SQL Web application is similar to that in a CGI Perl script. Developers often use CGI scripts to produce Web pages dynamically, but such scripts are often not optimal for accessing Oracle Database. Delivering Web content with PL/SQL stored subprograms provides the power and flexibility of database processing. For example, you can use DML, dynamic SQL, and cursors. You also eliminate the process overhead of forking a new CGI process to handle each HTTP request.

Figure 10–1 illustrates the generic process for a PL/SQL Web application.

Figure 10–1 PL/SQL Web Application



The process includes the following steps:

1. A user visits a Web page, follows a hypertext link, or submits data in a form, which causes the browser to send a HTTP request for a URL to an HTTP server.
2. The HTTP server calls a stored subprogram on an Oracle database according to the data encoded in the URL. The data in the URL takes the form of parameters to be passed to the stored subprogram.
3. The stored subprogram invokes subprograms in the PL/SQL Web Toolkit. Typically, subprograms such as `HTP.Print` generate Web pages dynamically. A generated Web page varies depending on the database contents and the input parameters.
4. The subprograms pass the dynamically generated page to the Web server.
5. The Web server delivers the page to the client.

Implementing PL/SQL Web Applications

You can implement a Web browser-based application entirely in PL/SQL with the following Oracle Database components:

- PL/SQL Gateway
- PL/SQL Web Toolkit

PL/SQL Gateway

The PL/SQL gateway enables a Web browser to invoke a PL/SQL stored subprogram through an HTTP listener. The gateway is a platform on which PL/SQL users develop and deploy PL/SQL Web applications.

mod_plsql

`mod_plsql` is one implementation of the PL/SQL gateway. The module is a plug-in of Oracle HTTP Server and enables Web browsers to invoke PL/SQL stored subprograms. Oracle HTTP Server is a component of both Oracle Application Server and Oracle Database.

The mod_plsql plug-in enables you to use PL/SQL stored subprograms to process HTTP requests and generate responses. In this context, an HTTP request is a URL that includes parameter values to be passed to a stored subprogram. PL/SQL gateway translates the URL, invokes the stored subprogram with the parameters, and returns output (typically HTML) to the client.

Some of the advantages of using mod_plsql over the embedded form of the PL/SQL gateway are as follows:

- You can run it in a firewall environment in which the Oracle HTTP Server runs on a firewall-facing host while the database is hosted behind a firewall. You cannot use this configuration with the embedded gateway.
- The embedded gateway does not support mod_plsql features such as dynamic HTML caching, system monitoring, and logging in the Common Log Format.

Embedded PL/SQL Gateway

You can use an embedded version of the PL/SQL gateway that runs in the XML DB HTTP Listener in the Oracle database. It provides the core features of mod_plsql in the database but does not require the Oracle HTTP Server. You configure the embedded PL/SQL gateway with the DBMS_EPG package in the PL/SQL Web Toolkit.

Some of the advantages of using the embedded gateway over mod_plsql are as follows:

- You can invoke PL/SQL Web applications such as Application Express without the need to install Oracle HTTP Server, thereby simplifying installation, configuration, and administration of PL/SQL based Web applications.
- You use the same configuration approach that is currently used to deliver content from Oracle XML DB in response to FTP and HTTP requests.

PL/SQL Web Toolkit

This set of PL/SQL packages is a generic interface that enables you to use stored subprograms invoked by mod_plsql at run time.

In response to a browser request, a PL/SQL subprogram updates or retrieves data from Oracle Database according to the user input. It then generates an HTTP response to the browser, typically in the form of a file download or HTML to be displayed. The Web Toolkit API enables stored subprograms to perform actions such as the following:

- Obtain information about an HTTP request
- Generate HTTP headers such as content-type and mime-type
- Set browser cookies
- Generate HTML pages

Table 10–1 describes commonly used PL/SQL Web Toolkit packages.

Table 10–1 Commonly Used Packages in the PL/SQL Web Toolkit

Package	Description of Contents
`HTF`	Function versions of the subprograms in the `htp` package. The function versions do not directly generate output in a Web page. Instead, they pass their output as return values to the statements that invoke them. Use these functions when you need to nest function calls.
`HTP`	Subprograms that generate HTML tags. For example, the procedure `htp.anchor` generates the HTML anchor tag, `<A>`.

Table 10–1 (Cont.) Commonly Used Packages in the PL/SQL Web Toolkit

Package	Description of Contents
`OWA_CACHE`	Subprograms that enable the PL/SQL gateway cache feature to improve performance of your PL/SQL Web application. You can use this package to enable expires-based and validation-based caching with the PL/SQL gateway file system.
`OWA_COOKIE`	Subprograms that send and retrieve HTTP cookies to and from a client Web browser. Cookies are strings a browser uses to maintain state between HTTP calls. State can be maintained throughout a client session or longer if a cookie expiration date is included.
`OWA_CUSTOM`	The authorize function used by cookies.
`OWA_IMAGE`	Subprograms that obtain the coordinates where a user clicked an image. Use this package when you have an image map whose destination links invoke a PL/SQL gateway.
`OWA_OPT_LOCK`	Subprograms that impose database optimistic locking strategies to prevent lost updates. Lost updates can otherwise occur if a user selects, and then attempts to update, a row whose values were changed in the meantime by another user.
`OWA_PATTERN`	Subprograms that perform string matching and string manipulation with regular expressions.
`OWA_SEC`	Subprograms used by the PL/SQL gateway for authenticating requests.
`OWA_TEXT`	Subprograms used by package `OWA_PATTERN` for manipulating strings. You can also use them directly.
`OWA_UTIL`	The following types of utility subprograms: ▪ Dynamic SQL utilities to produce pages with dynamically generated SQL code. ▪ HTML utilities to retrieve the values of CGI environment variables and perform URL redirects. ▪ Date utilities for correct date-handling. Date values are simple strings in HTML, but must be properly treated as an Oracle Database datatype.
`WPG_DOCLOAD`	Subprograms that download documents from a document repository that you define using the DAD configuration.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for syntax, descriptions, and examples for the PL/SQL Web Toolkit packages

Using mod_plsql Gateway to Map Client Requests to a PL/SQL Web Application

As explained in detail in the *Oracle HTTP Server mod_plsql User's Guide*, `mod_plsql` maps Web client requests to PL/SQL stored subprograms over HTTP. See this documentation for instructions.

See Also:

- *Oracle HTTP Server mod_plsql User's Guide* to learn how to configure and use `mod_plsql`
- *Oracle Fusion Middleware Administrator's Guide for Oracle HTTP Server* to obtain `mod_plsql` reference material

Using Embedded PL/SQL Gateway

The embedded gateway functions very similar to the `mod_plsql` gateway. Before using the embedded version of the gateway, familiarize yourself with the *Oracle HTTP Server mod_plsql User's Guide*. Much of the information is the same or similar.

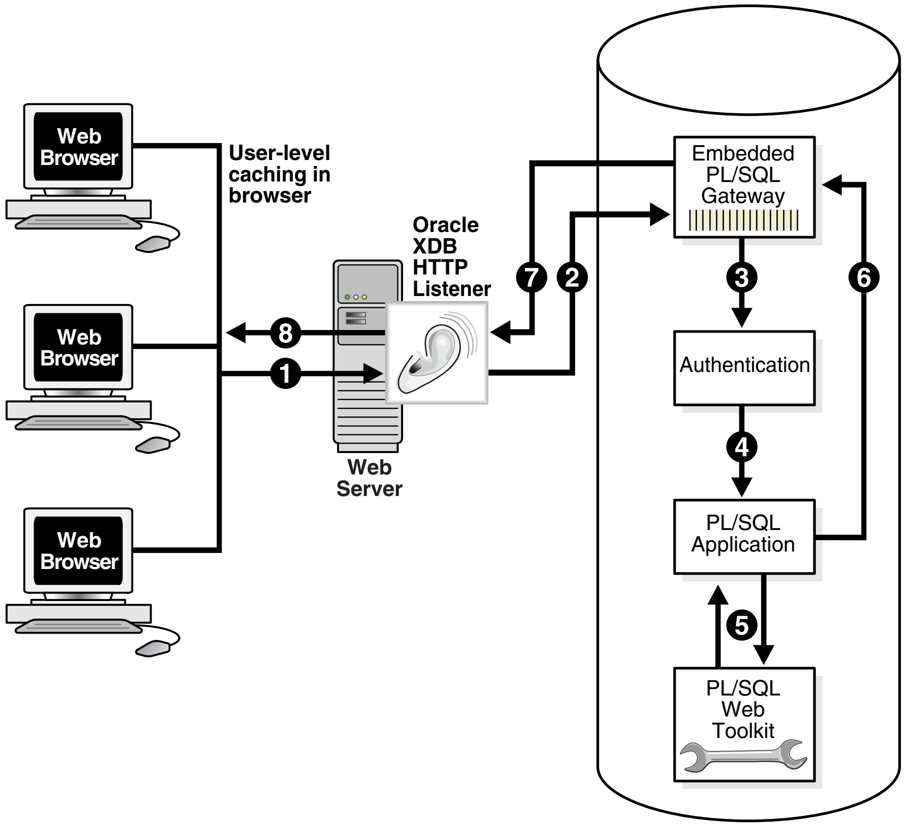
Topics:

- How Embedded PL/SQL Gateway Processes Client Requests
- Installing Embedded PL/SQL Gateway
- Configuring Embedded PL/SQL Gateway
- Invoking PL/SQL Stored Subprograms Through Embedded PL/SQL Gateway
- Securing Application Access with Embedded PL/SQL Gateway
- Restrictions in Embedded PL/SQL Gateway
- Using Embedded PL/SQL Gateway: Scenario

How Embedded PL/SQL Gateway Processes Client Requests

Figure 10–2 illustrates the process by which the embedded gateway handles client HTTP requests.

Figure 10–2 Processing Client Requests with Embedded PL/SQL Gateway



The explanation of the steps in Figure 10–2 is as follows:

1. The Oracle XML DB HTTP Listener receives a request from a client browser to request to invoke a PL/SQL subprogram. The subprogram can either be written

directly in PL/SQL or indirectly generated when a PL/SQL Server Page is uploaded to the database and compiled.

2. The XML DB HTTP Listener routes the request to the embedded PL/SQL gateway as specified in its virtual-path mapping configuration.
3. The embedded gateway uses the HTTP request information and the gateway configuration to determine which database account to use for authentication.
4. The embedded gateway prepares the call parameters and invokes the PL/SQL subprogram in the application.
5. The PL/SQL subprogram generates an HTML page out of relational data and the PL/SQL Web Toolkit accessed from the database.
6. The application sends the page to the embedded gateway.
7. The embedded gateway sends the page to the XML DB HTTP Listener.
8. The XML DB HTTP Listener sends the page to the client browser.

Unlike `mod_plsql`, the embedded gateway processes HTTP requests with the Oracle XML DB Listener. This listener is the same server-side process as the Oracle Net Listener and supports Oracle Net Services, HTTP, and FTP.

Configure general HTTP listener settings through the XML DB interface, which is described in *Oracle XML DB Developer's Guide*. Configure the HTTP listener either by using Oracle Enterprise Manager or by editing the `xdbconfig.xml` file. Use the `DBMS_EPG` package for all embedded PL/SQL gateway configuration, for example, creating or setting attributes for a DAD.

Installing Embedded PL/SQL Gateway

The embedded gateway requires the following components:

- XML DB HTTP Listener
- PL/SQL Web Toolkit

The embedded PL/SQL gateway is installed as part of Oracle XML DB. If you are using a preconfigured database created during an installation or by the Database Configuration Assistant (DBCA), then Oracle XML DB is already installed and configured. For information about manually adding Oracle XML DB to an existing database, see *Oracle XML DB Developer's Guide*.

The PL/SQL Web Toolkit is part of the standard installation of Oracle Database, so no supplementary installation is necessary.

Configuring Embedded PL/SQL Gateway

You configure `mod_plsql` by editing the Oracle HTTP Server configuration files. Because the embedded gateway is installed as part of the Oracle XML DB HTTP Listener, you manage the embedded gateway as a servlet through the Oracle XML DB servlet management interface.

The configuration interface to the embedded gateway is the PL/SQL package `DBMS_EPG`. This package modifies the underlying `xdbconfig.xml` configuration file that XML DB uses. The default values of the embedded gateway configuration parameters are sufficient for most users.

This section contains the following topics:

- Configuring Embedded PL/SQL Gateway: Overview

- Configuring User Authentication for Embedded PL/SQL Gateway

Configuring Embedded PL/SQL Gateway: Overview

As in `mod_plsql`, each request for a PL/SQL stored subprogram is associated with a Database Access Descriptor (DAD). A DAD is a set of configuration values used for database access. A DAD specifies information such as:

- The database account to use for authentication
- The subprogram to use for uploading and downloading documents

In the embedded PL/SQL gateway, a DAD is represented as a servlet in the XML DB HTTP Listener configuration. Each DAD attribute maps to an XML element in the configuration file `xdbconfig.xml`. The value of the DAD attribute corresponds to the element content. For example, the `database-username` DAD attribute corresponds to the `<database-username>` XML element; if the value of the DAD attribute is `HR` it corresponds to `<database-username>HR<database-username>`. Note that DAD attribute names are case-sensitive.

Use the `DBMS_EPG` package to perform the following embedded PL/SQL gateway configurations:

1. Create a new DAD with the `DBMS_EPG.CREATE_DAD` procedure.
2. Set DAD attributes with the `DBMS_EPG.SET_DAD_ATTRIBUTE` procedure. Note that all DAD attributes are optional. If you do not specify an attribute, then the default value is used.

Table 10–2 lists the embedded PL/SQL gateway attributes and the corresponding `mod_plsql` DAD parameters. Note that all enumeration values in the "Legal Values" column are case-sensitive.

Table 10–2 Mapping Between mod_plsql and Embedded PL/SQL Gateway DAD Attributes

mod_plsql DAD Attribute	Embedded PL/SQL Gateway DAD Attribute	Multiple Occurr.	Legal Values
`PlsqlAfterProcedure`	`after-procedure`	No	String
`PlsqlAlwaysDescribeProcedure`	`always-describe-procedure`	No	Enumeration of On, Off
`PlsqlAuthenticationMode`	`authentication-mode`	No	Enumeration of Basic, SingleSignOn, GlobalOwa, CustomOwa, PerPackageOwa
`PlsqlBeforeProcedure`	`before-procedure`	No	String
`PlsqlBindBucketLengths`	`bind-bucket-lengths`	Yes	Unsigned integer
`PlsqlBindBucketWidths`	`bind-bucket-widths`	Yes	Unsigned integer
`PlsqlCGIEnvironmentList`	`cgi-environment-list`	Yes	String
`PlsqlCompatibilityMode`	`compatibility-mode`	No	Unsigned integer
`PlsqlDatabaseUsername`	`database-username`	No	String
`PlsqlDefaultPage`	`default-page`	No	String
`PlsqlDocumentPath`	`document-path`	No	String
`PlsqlDocumentProcedure`	`document-procedure`	No	String
`PlsqlDocumentTablename`	`document-table-name`	No	String
`PlsqlErrorStyle`	`error-style`	No	Enumeration of ApacheStyle, ModplsqlStyle, DebugStyle
`PlsqlExclusionList`	`exclusion-list`	Yes	String
`PlsqlFetchBufferSize`	`fetch-buffer-size`	No	Unsigned integer

Table 10–2 (Cont.) Mapping Between mod_plsql and Embedded PL/SQL Gateway DAD Attributes

mod_plsql DAD Attribute	Embedded PL/SQL Gateway DAD Attribute	Multiple Occurr.	Legal Values
`PlsqlInfoLogging`	`info-logging`	No	Enumeration of InfoDebug
`PlsqlInputFilterEnable`	`input-filter-enable`	No	String
`PlsqlMaxRequestsPerSession`	`max-requests-per-session`	No	Unsigned integer
`PlsqlNLSLanguage`	`nls-language`	No	String
`PlsqlOWADebugEnable`	`owa-debug-enable`	No	Enumeration of On, Off
`PlsqlPathAlias`	`path-alias`	No	String
`PlsqlPathAliasProcedure`	`path-alias-procedure`	No	String
`PlsqlRequestValidationFuncti on`	`request-validation-functi on`	No	String
`PlsqlSessionCookieName`	`session-cookie-name`	No	String
`PlsqlSessionStateManagement`	`session-state-management`	No	Enumeration of StatelessWithResetPackageState, StatelessWithFastRestPackageState, StatelessWithPreservePackageState
`PlsqlTransferMode`	`transfer-mode`	No	Enumeration of Char, Raw
`PlsqlUploadAsLongRaw`	`upload-as-long-raw`	No	String

The embedded gateway assumes default values when the attributes are not set. The default values of the DAD attributes are sufficient for most users of the embedded gateway. `mod_plsql` users do not need the following attributes:

- `PlsqlDatabasePassword`
- `PlsqlDatabaseConnectString` (because the embedded gateway does not support logon to external databases)

Like the DAD attributes, the global configuration parameters are optional. Table 10–3 describes the `DBMS_EPG` global attributes and the corresponding `mod_plsql` global parameters.

Table 10–3 Mapping Between mod_plsql and Embedded PL/SQL Gateway Global Attributes

mod_plsql DAD Attribute	Embedded PL/SQL Gateway DAD Attribute	Multiple Occurr.	Legal Values
`PlsqlLogLevel`	`log-level`	No	Unsigned integer
`PlsqlMaxParameters`	`max-parameters`	No	Unsigned integer

See Also:

- *Oracle Fusion Middleware Administrator's Guide for Oracle HTTP Server* for detailed descriptions of the `mod_plsql` DAD attributes. See this documentation for default values and usage notes.
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_EPG` package
- *Oracle XML DB Developer's Guide* for an account of the `xdbconfig.xml` file

Configuring User Authentication for Embedded PL/SQL Gateway

Because it uses the XML DB authentication schemes, the embedded gateway handles database authentication differently from `mod_plsql`. In particular, it does not store database passwords in a DAD.

> **Note:** If you want to serve a PL/SQL Web application on the Internet but maintain Oracle Database behind a firewall, then you cannot use the embedded PL/SQL gateway to run the application. You must use `mod_plsql` instead.

Use the `DBMS_EPG` package to configure database authentication. This section contains the following topics:

- Configuring Static Authentication with DBMS_EPG
- Configuring Dynamic Authentication with DBMS_EPG
- Configuring Anonymous Authentication with DBMS_EPG
- Determining the Authentication Mode of a DAD
- Creating and Configuring DADs: Example
- Determining the Authentication Mode for a DAD: Example
- Determining the Authentication Mode for All DADs: Example
- Showing DAD Authorizations that Are Not in Effect: Example
- Examining Embedded PL/SQL Gateway Configuration

Configuring Static Authentication with DBMS_EPG Static authentication is for the `mod_plsql` user who stores database usernames and passwords in the DAD so that the browser user is not required to enter database authentication information.

To set up static authentication, follow these steps:

1. Log on to the database as an XML DB administrator, that is, a user with the `XDBADMIN` role assigned. For example:

   ```
   CONNECT SYSTEM/password
   ```

2. Create the DAD. For example, the following procedure creates a DAD invoked `HR_DAD` and maps the virtual path to `/hrweb/`:

   ```
   EXEC DBMS_EPG.CREATE_DAD('HR_DAD', '/hrweb/*');
   ```

3. For this step, you need the `ALTER ANY USER` system privilege. Set the DAD attribute `database-username` to the database account whose privileges must be used by the DAD. For example, the following procedure specifies that the DAD named `HR_DAD` has the privileges of the `HR` account:

   ```
   EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('HR_DAD', 'database-username', 'HR');
   ```

   Note that the DAD attribute `database-username` is case sensitive.

4. Assign the DAD the privileges of the database user specified in the previous step. This authorization enables end users to invoke procedures and access document tables through the embedded PL/SQL gateway with the privileges of the authorized account. For example:

   ```
   EXEC DBMS_EPG.AUTHORIZE_DAD('HR_DAD', 'HR');
   ```

Alternatively, you can log off as the user with XDBADMIN privileges, log on as the database user whose privileges must be used by the DAD, and then assign these privileges to the DAD. For example:

```
CONNECT HR/password
EXEC DBMS_EPG.AUTHORIZE_DAD('HR_DAD');
```

Note that multiple users can authorize the same DAD. The `database-username` attribute setting of the DAD determines which user's privileges to use.

Unlike `mod_plsql`, the embedded gateway connects to the database as the special user `ANONYMOUS`, but accesses database objects with the user privileges assigned to the DAD. The database rejects access if the browser user attempts to connect explicitly with the HTTP `Authorization` header.

Note: The account `ANONYMOUS` is locked after XML DB installation. If you want to use static authentication with the embedded PL/SQL gateway, then you must first unlock this account.

Configuring Dynamic Authentication with DBMS_EPG Dynamic authentication is for the `mod_plsql` user who does not store database usernames and passwords in the DAD.

In dynamic authentication, a database user does not have to authorize the embedded gateway to use its privileges to access database objects. Instead, browser users must supply the database authentication information through the HTTP Basic Authentication scheme.

The action of the embedded gateway depends on whether the `database-username` attribute is set for the DAD. If the attribute is not set, then the embedded gateway connects to the database as the user supplied by the browser client. If the attribute is set, then the database restricts access to the user specified in the `database-username` attribute.

To set up dynamic authentication, follow these steps:

1. Log on to the database as a an XML DB administrator, that is, a user with the `XDBADMIN` role. For example:

   ```
   CONNECT SYSTEM/password
   ```

2. Create the DAD. For example, the following procedure creates a DAD invoked `DYNAMIC_DAD` and maps the virtual path to `/hrweb/`:

   ```
   EXEC DBMS_EPG.CREATE_DAD('DYNAMIC_DAD', '/hrweb/*');
   ```

3. Optionally, set the DAD attribute `database-username` to the database account whose privileges must be used by the DAD. The browser user will be prompted to enter the username and password for this account when accessing the DAD. For example, the following procedure specifies that the DAD named `DYNAMIC_DAD` has the privileges of the `HR` account:

   ```
   EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('DYNAMIC_DAD', 'database-username', 'HR');
   ```

   Note that the attribute `database-username` is case sensitive.

WARNING: Passwords sent through the HTTP Basic Authentication scheme are not encrypted. Configure the embedded gateway to use the HTTPS protocol to protect the passwords sent by the browser clients.

Configuring Anonymous Authentication with DBMS_EPG Anonymous authentication is for the `mod_plsql` user who creates a special DAD database user for database logon, but stores the application procedures and document tables in a different schema and grants access to the procedures and document tables to PUBLIC.

To set up anonymous authentication, follow these steps:

1. Log on to the database as an XML DB administrator, that is, a user with the `XDBADMIN` role assigned. For example:

   ```
   CONNECT SYSTEM/password
   ```

2. Create the DAD. For example, the following procedure creates a DAD invoked `HR_DAD` and maps the virtual path to `/hrweb/`:

   ```
   EXEC DBMS_EPG.CREATE_DAD('HR_DAD', '/hrweb/*');
   ```

3. Set the DAD attribute `database-username` to `ANONYMOUS`. For example:

   ```
   EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('HR_DAD', 'database-username', 'ANONYMOUS');
   ```

   Note that both `database-username` and `ANONYMOUS` are case sensitive.

   There is no need to authorize the embedded gateway to use `ANONYMOUS` privileges to access database objects, because `ANONYMOUS` has no system privileges and owns no database objects.

Determining the Authentication Mode of a DAD If you know the name of a DAD, then the authentication mode for this DAD depends on the following factors:

- Does the DAD exist?
- Is the `database-username` attribute for the DAD set?
- Is the DAD authorized to use the privilege of the `database-username` user?
- Is the `database-username` attribute the same as one of the user authorized to use the DAD?

Authentication for the DAD is static only if each of the preceding questions is answered in the affirmative. Table 10–4 shows how the answers to the preceding questions determine the authentication mode.

Table 10–4 Authentication Possibilities for a DAD

DAD Exists?	database-username set?	User authorized?	Mode
Yes	Yes	Yes	Static
Yes	Yes	No	Dynamic restricted
Yes	No	Does not matter	Dynamic
Yes	Yes (to `ANONYMOUS`)	Does not matter	Anonymous
No			N/A

For example, assume that you create a DAD named `MY_DAD`. If the `database-username` attribute for `MY_DAD` is set to `HR`, but the `HR` user does not authorize `MY_DAD`, then the authentication mode for `MY_DAD` is dynamic and restricted. A browser user who attempts to execute a PL/SQL subprogram through `MY_DAD` is prompted to enter the `HR` database username and password.

The `DBA_EPG_DAD_AUTHORIZATION` view shows which users have authorized use of a DAD. The `DAD_NAME` column displays the name of the DAD; the `USERNAME` column displays the user whose privileges are assigned to the DAD. Note that the DAD authorized may or may not exist.

Creating and Configuring DADs: Example You can use the sample SQL script in Example 10–1 to create and configure different DADs for testing purposes. The script does the following:

- Creates a DAD invoked `Static_Auth_DAD` for database user `HR` and assigns it the privileges of the `HR` account.
- Creates a DAD invoked `Static_Auth_DAD_2` for database user `HR` but accidentally (for illustration) assigns `Static_Auth_DAD_Typo` the privileges of the `HR` account. Note that the `Static_Auth_DAD_Typo` DAD does not exist.
- Creates a DAD invoked `Dynamic_Auth_DAD` that is not restricted to any user.
- Creates a DAD invoked `Dynamic_Auth_DAD_Restricted` that is restricted to the `HR` account.
- Accidentally (for illustration) assigns `Static_Auth_DAD` the privileges of the database user `OE`, even though the `database-user` attribute for this DAD is set to `HR`.

Example 10–1 Configuring Authentication Modes

```
rem ---------------------------------------------------------------------
rem Create DAD with static auth
rem ---------------------------------------------------------------------

CONNECT SYSTEM/password
EXEC DBMS_EPG.CREATE_DAD('Static_Auth_DAD', '/static/*')
EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('Static_Auth_DAD', 'database-username', 'HR')

CONNECT HR/password
EXEC DBMS_EPG.AUTHORIZE_DAD('Static_Auth_DAD')

rem ---------------------------------------------------------------------
rem Create DAD with static auth typo
rem ---------------------------------------------------------------------

CONNECT SYSTEM/password
EXEC DBMS_EPG.CREATE_DAD('Static_Auth_DAD_2', '/static2/*')
EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('Static_Auth_DAD_2', 'database-username', 'HR')

CONNECT HR/password
EXEC DBMS_EPG.AUTHORIZE_DAD('Static_Auth_DAD_Typo')

rem ---------------------------------------------------------------------
rem Create DAD with dynamic auth
rem ---------------------------------------------------------------------

CONNECT SYSTEM/password
EXEC DBMS_EPG.CREATE_DAD('Dynamic_Auth_DAD', '/dynamic/*')
```

```
rem ---------------------------------------------------------------------
rem Create DAD with dynamic auth restricted
rem ---------------------------------------------------------------------

CONNECT SYSTEM/password
EXEC DBMS_EPG.CREATE_DAD('Dynamic_Auth_DAD_Restricted', '/dynamic/*')
EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('Dynamic_Auth_DAD_Restricted',
                                'database-username', 'HR')

rem ---------------------------------------------------------------------
rem Authorize DAD by a non-DAD user (DAD database-username is 'HR')
rem ---------------------------------------------------------------------

CONNECT OE/password
EXEC DBMS_EPG.AUTHORIZE_DAD('Static_Auth_DAD')
EXIT
```

Determining the Authentication Mode for a DAD: Example Example 10–2 creates a PL/SQL procedure named `show_dad_auth_status` that shows the privileges of a specified DAD. You execute the procedure by passing it the name of a DAD. Note that the procedure exits with an error if the specified DAD does not exist.

The procedure queries `DBA_EPG_DAD_AUTHORIZATION` to determine whether the specified DAD is set up for static authentication. If no row is found for the DAD, then authentication is dynamic. The procedure also executes the `DBMS_EPG.GET_DAD_ATTRIBUTE` procedure to determine whether the `database-username` attribute is set for the specified DAD. If it is set, and if the DAD uses dynamic authentication, then this DAD is restricted to the specified user. Otherwise, the DAD is set up for dynamic authentication for any user.

Example 10–2 Determining the Authentication Mode for a DAD

```
----------------------------------------------------------------------
-- This procedure shows the DAD authorization status for the given DAD
----------------------------------------------------------------------
CREATE OR REPLACE PROCEDURE show_dad_auth_status(p_dadname VARCHAR2)
IS
  v_daduser VARCHAR2(32);
  v_cnt     PLS_INTEGER;
BEGIN
  -- Get DAD's database-username attribute
  v_daduser := DBMS_EPG.GET_DAD_ATTRIBUTE(p_dadname, 'database-username');

  -- Determine whether DAD authorization exists for the DAD user
  SELECT COUNT(*)
    INTO v_cnt
  FROM   DBA_EPG_DAD_AUTHORIZATION da
  WHERE  da.DAD_NAME = p_dadname
  AND    da.USERNAME = v_daduser;

  -- If DAD authorization exists for the DAD user it is static authentication
  IF (v_cnt > 0) THEN
    DBMS_OUTPUT.PUT_LINE('''' || p_dadname ||
                         ''' is set up for static auth for user ''' ||
                         v_daduser || '''.');
    RETURN;
  END IF;

  -- Is dynamic authentication restricted to a particular user?
  IF (v_daduser IS NOT NULL) THEN
    DBMS_OUTPUT.PUT_LINE('''' || p_dadname ||
                         ''' is set up for dynamic auth for user ''' ||
                         v_daduser || ''' only.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('''' || p_dadname ||
                         ''' is set up for dynamic auth for any user.');
  END IF;
END;
/
```

Assume that you have run the script in Example 10–1 to create and configure various DADs. The following example queries the authorization status of `Static_Auth_DAD` and shows sample output:

```
SET SERVEROUTPUT ON
EXEC show_dad_auth_status('Static_Auth_DAD')

'Static_Auth_DAD' is set up for static auth for user 'HR'.
```

Determining the Authentication Mode for All DADs: Example Example 10–3 shows an anonymous PL/SQL block that determines the authentication mode for all registered DADs, that is, all DADs returned by executing the `DBMS_EPG.GET_DAD_LIST` procedure. The block executes the `show_dad_auth_status` procedure to find the authentication mode for each DAD.

Example 10–3 Showing the Authentication Mode for All DADs

```
DECLARE
  v_dad_names DBMS_EPG.VARCHAR2_TABLE;
BEGIN
  -- Show the DAD authorization status for all the DADs
  DBMS_OUTPUT.PUT_LINE('---------- authorization status for all DADs ----------');
  DBMS_EPG.GET_DAD_LIST(v_dad_names);

  -- loop through DAD names and display authorization status for each DAD
  FOR i IN 1..v_dad_names.count LOOP
    show_dad_auth_status(v_dad_names(i));
  END LOOP;
END;
/
```

If you ran the script in Example 10–1 to create and configure various DADs, the output is:

```
---------- authorization status for all DADs ----------
'Static_Auth_DAD' is set up for static auth for user 'HR'.
'Static_Auth_DAD_2' is set up for dynamic auth for user 'HR' only.
'Dynamic_Auth_DAD' is set up for dynamic auth for any user.
'Dynamic_Auth_DAD_Restricted' is set up for dynamic auth for user 'HR' only.
```

Showing DAD Authorizations that Are Not in Effect: Example Example 10–4 shows an anonymous PL/SQL block that shows DAD authorizations that are *not* in effect. This situation can occur in either of the following situations:

- A database user who authorizes a DAD is not the same user specified by the `database-username` attribute of the DAD
- A database user authorizes a DAD that does not exist

Example 10–4 Showing DAD Authorizations that Are Not in Effect

```
DECLARE
  v_dad_names  DBMS_EPG.VARCHAR2_TABLE;
  v_dad_user   VARCHAR2(32);
  v_dad_found  BOOLEAN;
BEGIN
  -- Show the DAD authorizations that are not in effect
  DBMS_OUTPUT.PUT_LINE('---------- DAD authorizations not in effect ----------');

  -- Get the DAD list for later use
  DBMS_EPG.GET_DAD_LIST(v_dad_names);

  -- Go through each DAD authorization and look for the DAD database-name
  -- attribute setting
  FOR r IN (SELECT * FROM DBA_EPG_DAD_AUTHORIZATION) LOOP
    v_dad_found := FALSE;

    FOR i IN 1..v_dad_names.count LOOP
      IF (r.DAD_NAME = v_dad_names(i)) THEN
        v_dad_user := DBMS_EPG.GET_DAD_ATTRIBUTE(r.DAD_NAME, 'database-username');

        -- Does the DAD database-username attribute match the user the
        -- the DAD is authorized for?
        IF (r.USERNAME <> v_dad_user) THEN
          DBMS_OUTPUT.PUT_LINE('DAD authorization of ''' || r.dad_name ||
                               ''' by user ''' || r.username || '''' ||
                               ' is not in effect because the DAD user is ' ||
                               '''' || v_dad_user || '''.');
        END IF;
        v_dad_found := TRUE;
        EXIT;
      END IF;
    END LOOP;

    -- Does the DAD exist?
    IF (NOT v_dad_found) THEN
      DBMS_OUTPUT.PUT_LINE('DAD authorization of ''' || r.dad_name ||
                           ''' by user ''' || r.username ||
                           ''' is not in effect because the DAD does not exist.');
    END IF;
  END LOOP;
END;
/
```

If you run the script in Example 10–1 to create and configure DADs, output (reformatted to fit on the page) is:

```
---------- DAD authorizations not in effect ----------
DAD authorization of 'Static_Auth_DAD' by user 'OE' is not in effect because the
  DAD user is 'HR'.
DAD authorization of 'Static_Auth_DAD_Typo' by user 'HR' is not in effect
  because the DAD does not exist.
```

Examining Embedded PL/SQL Gateway Configuration The following script helps you examine the configuration of the embedded PL/SQL gateway:

```
$ORACLE_HOME/rdbms/admin/epgstat.sql
```

Example 10–5 shows the output of the `epgstat.sql` script for Example 10–1.

Example 10–5 epgstat.sql Script Output for Example 10–1

```
SQL> @epgstat.sql
+--------------------------------------+
| XDB protocol ports:                  |
|  XDB is listening for the protocol   |
|  when the protocol port is nonzero.  |
+--------------------------------------+

HTTP Port FTP Port
--------- --------
     8080        0

1 row selected.

+---------------------------+
| DAD virtual-path mappings |
+---------------------------+

Virtual Path                     DAD Name
-------------------------------- --------------------------------
/dynamic/*                       Dynamic_Auth_DAD_Restricted
/static/*                        Static_Auth_DAD
/static2/*                       Static_Auth_DAD_2

3 rows selected.

+----------------+
| DAD attributes |
+----------------+

DAD Name         DAD Param                DAD Value
------------     --------------------- ----------------------------------------
Dynamic_Auth     database-username        HR
_DAD_Restric
ted

Static_Auth_     database-username        HR
DAD

Static_Auth_     database-username        HR
DAD2

3 rows selected.

+---------------------------------------------------+
| DAD authorization:                                |
|  To use static authentication of a user in a DAD, |
|  the DAD must be authorized for the user.         |
+---------------------------------------------------+

DAD Name                         User Name
-------------------------------- --------------------------------
Static_Auth_DAD                  HR
                                 OE
Static_Auth_DAD_Typo             HR

3 rows selected.

+----------------------------+
| DAD authentication schemes |
+----------------------------+

DAD Name             User Name                        Auth Scheme
-------------------- -------------------------------- ------------------
Dynamic_Auth_DAD                                      Dynamic
Dynamic_Auth_DAD_Res HR                               Dynamic Restricted
tricted

Static_Auth_DAD      HR                               Static
Static_Auth_DAD_2    HR                               Dynamic Restricted

4 rows selected.

+----------------------------------------------------------+
| ANONYMOUS user status:                                   |
|  To use static or anonymous authentication in any DAD,   |
|  the ANONYMOUS account must be unlocked.                 |
+----------------------------------------------------------+

Database User   Status
--------------- --------------------
ANONYMOUS       EXPIRED

1 row selected.

+---------------------------------------------------------------------+
| ANONYMOUS access to XDB repository:                                 |
|  To allow public access to XDB repository without authentication,   |
|  ANONYMOUS access to the repository must be allowed.                |
+---------------------------------------------------------------------+

Allow repository anonymous access?
----------------------------------
true

1 row selected.
```

Invoking PL/SQL Stored Subprograms Through Embedded PL/SQL Gateway

The basic steps for invoking PL/SQL subprograms through the embedded PL/SQL gateway are the same as for the `mod_plsql` gateway. See *Oracle HTTP Server mod_plsql User's Guide* for instructions. You need to adapt the `mod_plsql` instructions slightly for use with the embedded gateway. For example, invoke the embedded gateway in a browser by entering the URL in the following format:

```
protocol://hostname[:port]/virt-path/[[!][schema.][package.]proc_name[?query_str]]
```

The placeholder `virt-path` stands for the virtual path that you configured in `DBMS_EPG.CREATE_DAD`. The `mod_plsql` documentation uses `DAD_location` instead of `virt-path`.

The following topics documented in the first chapter of *Oracle HTTP Server mod_plsql User's Guide* apply equally to the embedded gateway:

- Transaction mode
- Supported data types
- Parameter-passing scheme
- File upload and download support

- Path-aliasing
- Common Gateway Interface (CGI) environment variables

Securing Application Access with Embedded PL/SQL Gateway

The embedded gateway shares the same protection mechanism with `mod_plsql`. See *Oracle HTTP Server mod_plsql User's Guide* for instructions.

Restrictions in Embedded PL/SQL Gateway

The `mod_plsql` restrictions documented in the first chapter of *Oracle HTTP Server mod_plsql User's Guide* apply equally to the embedded gateway. In addition, the embedded version of the gateway does not support the following features:

- Dynamic HTML caching
- System monitoring
- Single sign-on (SSO)

Using Embedded PL/SQL Gateway: Scenario

This section illustrates how to write a simple application that queries the `hr.employees` table and delivers HTML output to a Web browser through the PL/SQL gateway. It assumes that you have both XML DB and the sample schemas installed.

To write and execute the program follows these steps:

1. Log on to the database as a user with `ALTER USER` privileges and make sure that the database account `ANONYMOUS` is unlocked. The `ANONYMOUS` account, which is locked by default, is required for static authentication. If the account is locked, then use the following SQL statement to unlock it:

   ```
   ALTER USER anonymous ACCOUNT UNLOCK;
   ```

2. Log on to the database as an XML DB administrator, that is, a user with the `XDBADMIN` role. For example:

   ```
   CONNECT SYSTEM/password
   ```

   Note that you can determine which users and roles were granted the `XDADMIN` role by querying the data dictionary as follows:

   ```
   SELECT *
   FROM   DBA_ROLE_PRIVS
   WHERE  GRANTED_ROLE = 'XDBADMIN';
   ```

3. Create the DAD. For example, the following procedure creates a DAD invoked `HR_DAD` and maps the virtual path to `/hrweb/`:

   ```
   EXEC DBMS_EPG.CREATE_DAD('HR_DAD', '/plsql/*');
   ```

4. Set the DAD attribute `database-username` to the database user whose privileges must be used by the DAD. For example, the following procedure specifies that the DAD `HR_DAD` accesses database objects with the privileges of user `HR`:

   ```
   EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('HR_DAD', 'database-username', 'HR');
   ```

   Note that the attribute `database-username` is case sensitive.

5. Log off as the XML DB administrator and log on to the database as the database user whose privileges must be used by the DAD. For example:

   ```
   CONNECT HR/password
   ```

6. Authorize the embedded PL/SQL gateway to invoke procedures and access document tables through the DAD. For example:

   ```
   EXEC DBMS_EPG.AUTHORIZE_DAD('HR_DAD');
   ```

7. Create a sample PL/SQL stored procedure invoked `print_employees`. The following program creates an HTML page that includes the result set of a query of `hr.employees`:

   ```
   CREATE OR REPLACE PROCEDURE print_employees
   IS
     CURSOR emp_cursor IS
     SELECT last_name, first_name
     FROM hr.employees
     ORDER BY last_name;
   BEGIN
     HTP.PRINT('<html>');
     HTP.PRINT('<head>');
     HTP.PRINT('<meta http-equiv="Content-Type" content="text/html">');
     HTP.PRINT('<title>List of Employees</title>');
     HTP.PRINT('</head>');
     HTP.PRINT('<body TEXT="#000000" BGCOLOR="#FFFFFF">');
     HTP.PRINT('<h1>List of Employees</h1>');
     HTP.PRINT('<table width="40%" border="1">');
     HTP.PRINT('<tr>');
     HTP.PRINT('<th align="left">Last Name</th>');
     HTP.PRINT('<th align="left">First Name</th>');
     HTP.PRINT('</tr>');
     FOR emp_record IN emp_cursor LOOP
       HTP.PRINT('<tr>');
       HTP.PRINT('<td>' || emp_record.last_name  || '</td>');
       HTP.PRINT('<td>' || emp_record.first_name || '</td>');
     END LOOP;
     HTP.PRINT('</table>');
     HTP.PRINT('</body>');
     HTP.PRINT('</html>');
   END;
   /
   ```

8. Ensure that the Oracle Net listener can accept HTTP requests. You can determine the status of the listener on Linux and UNIX by running the following command at the system prompt:

   ```
   lsnrctl status | grep HTTP
   ```

   Output (reformatted from a single line to multiple lines due to page size constraints):

   ```
   (DESCRIPTION=
     (ADDRESS=(PROTOCOL=tcp)(HOST=test.com)(PORT=8080))
     (Presentation=HTTP)
     (Session=RAW)
   )
   ```

If you do not see the HTTP service started, then you can set following in you initialization parameter file (replacing *listener_name* with the name of your Oracle Net local listener), then restart the database and the listener:

```
dispatchers="(PROTOCOL=TCP)"
local_listener=listener_name
```

9. Run the `print_employees` program from your Web browser. For example, you can use the following URL, replacing *host* with the name of your host computer and *port* with the value of the `PORT` parameter in the previous step:

   ```
   http://host:port/plsql/print_employees
   ```

   For example, if your host is `test.com` and your HTTP port is `8080`, then enter:

   ```
   http://test.com:8080/plsql/print_employees
   ```

   The Web browser returns an HTML page with a table that includes the first and last name of every employee in the `hr.employees` table.

Generating HTML Output with PL/SQL

Traditionally, PL/SQL Web applications use function calls to generate each HTML tag for output. These functions are part of the PL/SQL Web Toolkit packages that come with Oracle Database. Example 10–6 illustrates how to generate a simple HTML page by calling the `HTP` functions that correspond to each HTML tag.

Example 10–6 Displaying HTML Tags with HTP Functions

```
CREATE OR REPLACE PROCEDURE html_page
IS
BEGIN
  HTP.HTMLOPEN;                            -- generates <HTML>
  HTP.HEADOPEN;                            -- generates <HEAD>
  HTP.TITLE('Title');                      -- generates <TITLE>Hello</TITLE>
  HTP.HEADCLOSE;                           -- generates </HTML>

  -- generates <BODY TEXT="#000000" BGCOLOR="#FFFFFF">
  HTP.BODYOPEN( cattributes => 'TEXT="#000000" BGCOLOR="#FFFFFF"');

  -- generates <H1>Heading in the HTML File</H1>
  HTP.HEADER(1, 'Heading in the HTML File');

  HTP.PARA;                                 -- generates <P>
  HTP.PRINT('Some text in the HTML file.');
  HTP.BODYCLOSE;                            -- generates </BODY>
  HTP.HTMLCLOSE;                            -- generates </HTML>
END;
```

An alternative to making function calls that correspond to each tag is to use the `HTP.PRINT` function to print the text and tags together. Example 10–7 illustrates this technique.

Example 10–7 Displaying HTML Tags with HTP.PRINT

```
CREATE OR REPLACE PROCEDURE html_page2
IS
BEGIN
  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>');
END;
```

Chapter 11, "Developing PL/SQL Server Pages" describes an additional method for delivering using PL/SQL to generate HTML content. PL/SQL server pages enables you to build on your knowledge of HTML tags and avoid 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 do the following:

- Simplify the processing involved in displaying tables
- Store persistent data (cookies)
- Work with CGI protocol internals

Passing Parameters to PL/SQL Web Applications

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, streamline the interaction so that users can specify these choices very simply, without excessive 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 subprogram when the user clicks the `Submit` button on the page.
- Hard-coded in the URL. The user clicks on a link, and a set of predefined parameters are transmitted to a stored subprogram. Typically, you include separate links on your Web page for all the choices that the user might want.

Topics:

- Passing List and Dropdown-List Parameters from an HTML Form
- Passing Radio Button and Checkbox Parameters from an HTML Form
- Passing Entry-Field Parameters from an HTML Form
- Passing Hidden Parameters from an HTML Form
- Uploading a File from an HTML Form
- Submitting a Completed HTML Form
- Handling Missing Input from an HTML Form
- Maintaining State Information Between Web Pages

Passing List and Dropdown-List Parameters from an HTML Form

List boxes and drop-down lists are implemented with the HTML tag `<SELECT>`.

Use a list box for a large number of choices 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 drop-down list in the following situations:

- There are a small number of choices
- Screen space is limited.
- Choices are in an unusual order.

The drop-down captures the attention of first-time users and makes them read the items. If you keep the choices and order consistent, then users can memorize the motion of selecting an item from the drop-down list, allowing them to make selections quickly as they gain experience. Example 10–8 shows a simple drop-down list.

Example 10–8 HTML Drop-Down List

```
<form>
<select name="seasons">
<option value="winter">Winter
<option value="spring">Spring
<option value="summer">Summer
<option value="fall">Fall
</select>
```

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 subprogram. 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 subprogram 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 subprogram receives a null value for the corresponding parameter.

If one checkbox in a group is checked, the stored subprogram receives a single `VARCHAR2` parameter.

If more than one checkbox in a group is checked, the stored subprogram 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 client-side Javascript, 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 subprogram 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 subprograms 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 invoked, the user sees a simple form prompting for the input values. When the user submits the information, the same procedure is invoked 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 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 invoke 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 subprograms, without requiring the user to specify the same choices each time, is to include hidden parameters in the form that invokes a stored subprogram. The first stored subprogram 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 subprogram, as if the user had entered it through a radio button or entry field.

Other techniques for passing information from one stored subprogram 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 subprograms can retrieve it. This technique involves some extra overhead on the database server, and you must still find a way to keep track of each user as multiple users access the server at the same time.

Uploading a File from an HTML Form

You can use an HTML form to choose a file on a client system, and transfer it to the server. A stored subprogram can insert the file into the database as a `CLOB`, `BLOB`, or other type that can hold large amounts of data.

The PL/SQL Web toolkit and the PL/SQL gateway have the notion of a "document table" that holds uploaded files.

See Also: *mod_plsql User's Guide*

Submitting a Completed HTML Form

By default, an HTML form must have a `Submit` button, which transmits the data from the form to a stored subprogram 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 drop-down 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 subprogram 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 subprograms to handle the possibility that some parameters are null:

- Use a `DEFAULT` clause in all parameter declarations, to prevent an exception when the stored subprogram is invoked 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 subprogram 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 subprogram again, directly from the results page. For example, include a link that invokes the same stored subprogram 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 subprogram parameters for you.

If the user has to make multiple selections, or one selection from many choices, or it is important to avoid an accidental selection, use an HTML form. After the user makes and reviews all the choices, they confirm the choices with the `Submit` button. Subsequent pages can use forms with hidden parameters (`<INPUT TYPE=HIDDEN>` tags) to pass these choices from one page to the next.

If the user is only considering one or two choices, or the decision points are scattered throughout the Web page, you can save the user from hunting around for the `Submit` button by representing actions as hyperlinks and including any necessary name-value pairs in the query string (the part following the `?` within a URL).

An alternative way to main state information is to use Oracle Application Server and its `mod_ose` module. This approach lets you store state information in package variables that remain available as a user moves around a Web site.

> **See Also:** The Oracle Application Server documentation set at `http://www.oracle.com/technology/documentation`

Performing Network Operations in PL/SQL Stored Subprograms

While built-in PL/SQL features are focused on traditional database operations and programming logic, Oracle Database provides packages that open up Internet computing to PL/SQL programmers.

This section contains the following topics:

- Sending E-Mail from PL/SQL
- Getting a Host Name or Address from PL/SQL
- Using TCP/IP Connections from PL/SQL
- Retrieving HTTP URL Contents from PL/SQL
- Using Tables, Image Maps, Cookies, and CGI Variables from PL/SQL

Sending E-Mail from PL/SQL

You can send e-mail from a PL/SQL program or stored subprogram with the `UTL_SMTP` package. You can read about this package in the *Oracle Database PL/SQL Packages and Types Reference*.

The following code example illustrates how the SMTP package might be used by an application to send e-mail. 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 message were in single string, open_data(), write_data(), and close_data()
--  could be in a single call to data().
    utl_smtp.open_data(mail_conn);
    utl_smtp.write_data(mail_conn, 'This is a test message.' || chr(13));
    utl_smtp.write_data(mail_conn, 'This is line 2.' || chr(13));
    utl_smtp.close_data(mail_conn);
    utl_smtp.quit(mail_conn);
    EXCEPTION
        WHEN OTHERS THEN
           -- Insert error-handling code here
           NULL;
END;
```

Getting a Host Name or Address from PL/SQL

You can determine the host name of the local system, or the IP address of a given host name from a PL/SQL program or stored subprogram using the UTL_INADDR package. You can find details about this package in the *Oracle Database PL/SQL Packages and Types Reference*. You use the results in calls to the UTL_TCP package.

Using TCP/IP Connections from PL/SQL

You can open TCP/IP connections to systems on the network, and read or write to the corresponding sockets, using the UTL_TCP package. You can find details about this package in the *Oracle Database PL/SQL Packages and Types Reference*.

Retrieving HTTP URL Contents 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 might be plain text, a JPEG image, or any sort of file that is downloadable from a Web server. You can find details about this package in the *Oracle Database PL/SQL Packages and Types Reference*.

The UTL_HTTP package lets you:

- Control the details of the HTTP session, including header lines, cookies, redirects, proxy servers, IDs and passwords for protected sites, and CGI parameters through the GET or POST methods.
- Speed up multiple accesses to the same Web site using HTTP 1.1 persistent connections.
- Construct and interpret URLs for use with UTL_HTTP through the ESCAPE and UNESCAPE functions in the UTL_URL package.

Typically, developers have used Java or Perl to perform these operations; this package lets you do them with PL/SQL.

```
CREATE OR REPLACE PROCEDURE show_url
(
    url      IN VARCHAR2,
    username IN VARCHAR2 DEFAULT NULL,
    password IN VARCHAR2 DEFAULT NULL
) AS
    req       utl_http.req;
    resp      utl_http.resp;
    name      VARCHAR2(256);
    value     VARCHAR2(1024);
    data      VARCHAR2(255);
    my_scheme VARCHAR2(256);
    my_realm  VARCHAR2(256);
    my_proxy  BOOLEAN;
BEGIN
-- When going through a firewall, pass requests through this host.
-- Specify sites inside the firewall that don't need the proxy host.
  utl_http.set_proxy('proxy.my-company.com', 'corp.my-company.com');

-- Ask UTL_HTTP not to raise an exception for 4xx and 5xx status codes,
-- rather than just returning the text of the error page.
  utl_http.set_response_error_check(FALSE);

-- Begin retrieving this Web page.
  req := utl_http.begin_request(url);
```

```
-- Identify ourselves. Some sites serve special pages for particular browsers.
  utl_http.set_header(req, 'User-Agent', 'Mozilla/4.0');

-- Specify a user ID and password for pages that require them.
  IF (username IS NOT NULL) THEN
    utl_http.set_authentication(req, username, password);
  END IF;

  BEGIN
-- Start receiving the HTML text.
    resp := utl_http.get_response(req);

-- Show the status codes and reason phrase of the response.
    dbms_output.put_line('HTTP response status code: ' || resp.status_code);
    dbms_output.put_line('HTTP response reason phrase: ' || resp.reason_phrase);

-- Look for client-side error and report it.
    IF (resp.status_code >= 400) AND (resp.status_code <= 499) THEN

-- Detect whether the page is password protected, and we didn't supply
-- the right authorization.
      IF (resp.status_code = utl_http.HTTP_UNAUTHORIZED) THEN
        utl_http.get_authentication(resp, my_scheme, my_realm, my_proxy);
        IF (my_proxy) THEN
          dbms_output.put_line('Web proxy server is protected.');
          dbms_output.put('Please supply the required ' || my_scheme ||
            ' authentication username/password for realm ' || my_realm ||
            ' for the proxy server.');
        ELSE
          dbms_output.put_line('Web page ' || url || ' is protected.');
          dbms_output.put('Please supplied the required ' || my_scheme ||
            ' authentication username/password for realm ' || my_realm ||
            ' for the Web page.');
        END IF;
      ELSE
        dbms_output.put_line('Check the URL.');
      END IF;

      utl_http.end_response(resp);
      RETURN;

-- Look for server-side error and report it.
    ELSIF (resp.status_code >= 500) AND (resp.status_code <= 599) THEN

      dbms_output.put_line('Check if the Web site is up.');
      utl_http.end_response(resp);
      RETURN;

    END IF;

-- The HTTP header lines contain information about cookies, character sets,
-- and other data that client and server can use to customize each session.
    FOR i IN 1..utl_http.get_header_count(resp) LOOP
      utl_http.get_header(resp, i, name, value);
      dbms_output.put_line(name || ': ' || value);
    END LOOP;

-- Keep reading lines until no more are left and an exception is raised.
    LOOP
      utl_http.read_line(resp, value);
      dbms_output.put_line(value);
    END LOOP;
  EXCEPTION
    WHEN utl_http.end_of_body THEN
    utl_http.end_response(resp);
  END;

END;
/
SET serveroutput ON
-- The following URLs illustrate the use of this procedure,
-- but these pages do not actually exist. To test, substitute
-- URLs from your own Web server.
exec show_url('http://www.oracle.com/no-such-page.html')
exec show_url('http://www.oracle.com/protected-page.html')
exec show_url('http://www.oracle.com/protected-page.html', 'scott', 'tiger')
```

Using Tables, Image Maps, Cookies, and CGI Variables from PL/SQL

Packages for all of these functions are supplied with Oracle8*i* and higher. You use these packages in combination with the `mod_plsql` plug-in of Oracle HTTP Server (OHS). You can format the results of a query in an HTML table, produce an image map, set and get HTTP cookies, check the values of CGI variables, and combine other typical Web operations with a PL/SQL program.

Documentation for these packages is not part of the database documentation library. The location of the documentation depends on the particular application server you are running. To get started with these packages, look at their subprogram names and parameters using the SQL*Plus `DESCRIBE` statement:

```
DESCRIBE HTP;
DESCRIBE HTF;
DESCRIBE OWA_UTIL;
```

11 Developing PL/SQL Server Pages

This chapter explains how to develop PL/SQL Server Pages (PSP), which let you include dynamic content in web pages.

Topics:

What Are PL/SQL Server Pages and Why Use Them?

PL/SQL Server Pages (PSP) are server-side scripts that include dynamic content, including the results of SQL queries, inside web pages. You can author the web pages in an HTML authoring tool and insert blocks of PL/SQL code.

Example 11–1 shows a simple PL/SQL server page called `simple.psp`.

Example 11–1 simple.psp

```
<%@ page language="PL/SQL" %>
<%@ page contentType="text/html" %>
<%@ plsql procedure="show_employees" %>
<%-- This example displays the last name and first name of every
     employee in the hr.employees table. --%>
<%!
  CURSOR emp_cursor IS
  SELECT last_name, first_name
  FROM hr.employees
  ORDER BY last_name;
%>
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html">
<title>List of Employees</title>
</head>
<body TEXT="#000000" BGCOLOR="#FFFFFF">
<h1>List of Employees</h1>
<table width="40%" border="1">
<tr>
<th align="left">Last Name</th>
<th align="left">First Name</th>
</tr>
<%  FOR emp_record IN emp_cursor LOOP %>
  <tr>
  <td> <%= emp_record.last_name %> </td>
  <td> <%= emp_record.first_name %> </td>
  </tr>
<%  END LOOP; %>
</table>
</body>
</html>
```

You can compile and load this script into an Oracle database with the `loadpsp` command-line utility. The following example loads this server page into the `hr` schema, replacing the `show_employees` procedure if it already exists:

```
loadpsp -replace -user hr/hr simple.psp
```

Browser users can execute the `show_employees` procedure through a URL. An HTML page that displays the last and first names of employees in the `hr.employees` table is returned to the browser through the PL/SQL gateway.

Deploying content through PL/SQL Server Pages has the following advantages:

- For developers familiar with PL/SQL, the server pages are the easiest way to create professional web pages that included database-generated content. You can develop web pages as normal and then embed PL/SQL code in the HTML.
- PL/SQL Server Pages can be more convenient than using the `HTP` and `HTF` packages to write out HTML content line by line.
- Because processing is performed on the database server, the client browser receives a plain HTML page with no special script tags. You can support all browsers and browser levels equally.
- Network traffic is efficient because use of PL/SQL Server Pages minimizes the number of database round-trips.
- You can 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 system.

Prerequisites for Developing and Deploying PL/SQL Server Pages

To develop and deploy PL/SQL server pages, you must meet the following prerequisites:

- To write a PL/SQL server page you need access to a text editor or HTML authoring tool for writing the script. No other development tool is required.
- To load a PL/SQL server page you need:
  - An account on an Oracle database in which to load the server pages.

  - Execution rights to the `loadpsp` command-line utility, which is located in `$ORACLE_HOME/bin`.

- To deploy the server pages you must use mod_plsql. As explained in "Using mod_plsql Gateway to Map Client Requests to a PL/SQL Web Application" on page 10-4, the gateway makes use of the PL/SQL Web Toolkit.

  See Also:

  - "Using mod_plsql Gateway to Map Client Requests to a PL/SQL Web Application" on page 10-4

PL/SQL Server Pages and the HTP Package

You can enable browser users to execute PL/SQL program units through HTTP in the following ways:

- By writing an HTML page with embedded PL/SQL code and compiling it as a PL/SQL server page. You might invoke subprograms from the PL/SQL Web Toolkit, but not to generate every line of HTML output.
- By writing a complete stored subprogram that produces HTML by invoking the `HTP` and `OWA_*` packages in the PL/SQL Web Toolkit. This technique is described in "Generating HTML Output with PL/SQL" on page 10-20.

Thus, you must choose which technique to use when writing your web application. 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 if you want to include dynamic content or make it the front end of a database application, then use PL/SQL Server Pages.
  - If you have a large body of PL/SQL code that produces formatted output, then you might find it more convenient to produce HTML tags by changing your print statements to invoke 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, then use PL/SQL Server Pages.
  - If you use authoring tools that produce PL/SQL code, then it might be less convenient to use PL/SQL Server Pages.

PL/SQL Server Pages and Other Scripting Solutions

Scripting solutions can be client-side or server-side. JavaScript is one of the most popular client-side scripting language. PL/SQL Server Pages fully support JavaScript. 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.

Java Server Pages (JSP) and Active Server Pages (ASP) are two of the most popular server-side scripting solutions. Compared to PL/SQL Server Pages:

- Java server pages are loosely analogous to PL/SQL Server Pages pages; Java servlets are analogous to PL/SQL packages. PL/SQL Server Pages use the same script tag syntax as JSP to make it easy to switch back and forth.

- PL/SQL Server Pages use syntax that is similar to ASP, although not identical. Typically, you must translate from VBScript or JScript to PL/SQL. The best candidates for migration are pages that use the Active Data Object (ADO) interface to perform database operations.

> **Note:** 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 PL/SQL Server Pages features.

Developing PL/SQL Server Pages

To develop a PL/SQL server page, you can start with an existing web page or with an existing stored subprogram. Either way, with a few additions and changes you can create dynamic web pages that perform database operations and display the results.

The file for a PL/SQL server page must have the extension `.psp`. It can contain whatever content you choose, with text and tags interspersed with PL/SQL Server Pages directives, declarations, and scriptlets. A server page can take the following forms:

- In the simplest case, it is an HTML file. Compiling it as a PL/SQL server page produces a stored subprogram that outputs exactly the same HTML file.
- In the most complex case, it is a PL/SQL subprogram that generates all the content of the web page, including the tags for title, body, and headings.
- In the typical case, it is a mixture of HTML (providing the static parts of the page) and PL/SQL (providing the dynamic content).

The order and placement of the PL/SQL Server Pages directives and declarations is usually not significant. It becomes significant only when another file is included. For ease of maintenance, it is recommended that you place the directives and declarations together near the beginning of the file.

Table 11–1 lists the PL/SQL Server Pages elements and directs you to the section that explains how to use them. The section "Quoting and Escaping Strings in a PSP Script" on page 11-11 describes how to quote strings that are used in various PL/SQL Server Pages elements.

Table 11–1 PSP Elements

PSP Element	Name	Specifies . . .	Section
`<%@ page` ... `%>`	Page Directive	Characteristics of the PL/SQL server page.	"Specifying Basic Server Page Characteristics" on page 11-5
`<%@ parameter` ... `%>`	Parameter Directive	The name, and optionally the type and default, for each parameter expected by the PSP stored procedure.	"Accepting User Input" on page 11-8
`<%@ plsql` ... `%>`	Procedure Directive	The name of the stored procedure produced by the PSP file.	"Naming the PL/SQL Stored Procedure" on page 11-8
`<%@ include` ... `%>`	Include Directive	The name of a file to be included at a specific point in the PSP file.	"Including the Contents of Other Files" on page 11-9
`<%!` ... `%>`	Declaration Block	The declaration for a set of PL/SQL variables that are visible throughout the page, not just within the next `BEGIN`/`END` block.	"Declaring Global Variables in a PSP Script" on page 11-9

Table 11–1 (Cont.) PSP Elements

PSP Element	Name	Specifies . . .	Section
`<% ... %>`	Code Block	A set of PL/SQL statements to be executed when the procedure is run.	"Specifying Executable Statements in a PSP Script" on page 11-10
`<%= ... %>`	Expression Block	A single PL/SQL expression, such as a string, arithmetic expression, function call, or combination of these.	"Substituting Expression Values in a PSP Script" on page 11-11
`<%-- ... --%>`	Comment	A comment in a PSP script.	"Including Comments in a PSP Script" on page 11-12

Note: If you are already familiar with dynamic HTML, you can go directly to"Examples of PL/SQL Server Pages" on page 11-15.

Topics:

- Specifying Basic Server Page Characteristics
- Accepting User Input
- Naming the PL/SQL Stored Procedure
- Including the Contents of Other Files
- Declaring Global Variables in a PSP Script
- Specifying Executable Statements in a PSP Script
- Substituting Expression Values in a PSP Script
- Quoting and Escaping Strings in a PSP Script
- Including Comments in a PSP Script

Specifying Basic Server Page Characteristics

Use the `<%@ page ... %>` directive to specify characteristics of the PL/SQL server page such as the following:

- What scripting language it uses.
- What type of information (MIME type) it produces.
- What code to run to handle all uncaught exceptions. This might be an HTML file with a friendly message, renamed to a `.psp` file. You must specify this same file name in the `loadpsp` command that compiles the main PSP file. You must specify exactly the same name in both the `errorPage` directive and in the `loadpsp` command, including any relative path name such as `../include/`.

The following code shows the syntax of the `page` directive (the attribute names `contentType` and `errorPage` are case-sensitive):

```
<%@ page [language="PL/SQL"] [contentType="content type string"] charset="encoding" [errorPage="file.psp"] %>
```

Topics:

- Specifying the Scripting Language
- Returning Data to the Client Browser
- Handling Script Errors

Specifying the Scripting Language

To identify a file as a PL/SQL server page, include the following directive somewhere in the file:

```
<%@ page language="PL/SQL" %>
```

This directive is for compatibility with other scripting environments. Example 11–1 shows an example of a simple PL/SQL server page that includes the language directive.

Returning Data to the Client Browser

Options:

- Returning HTML
- Returning XML, Text, and Other Document Types
- Returning Pages Containing Different Character Sets

Returning HTML The PL/SQL parts of a PL/SQL server page are enclosed within special delimiters. All other content is passed verbatim—including any whitespace—to the browser. To display text or HTML tags, write it as you would write a typical web page. You do not need to invoke any output functions. As illustration, the server page in Example 11–1 returns the HTML page shown in Example 11–2, except that it includes the table rows for the queried employees.

Example 11–2 Sample Returned HTML Page

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html">
<title>List of Employees</title>
</head>
<body TEXT="#000000" BGCOLOR="#FFFFFF">
<h1>List of Employees</h1>
<table width="40%" border="1">
<tr>
<th align="left">Last Name</th>
<th align="left">First Name</th>
</tr>

  <!-- result set of query of hr.employees inserted here -->

</table>
</body>
</html>
```

Sometimes you might want to display one line of output or another, or change the value of an attribute, based on a condition. You can include control structures and variable substitution inside the PSP delimiters, as shown in the following code fragment from Example 11–1:

```
<%  FOR emp_record IN emp_cursor LOOP %>
  <tr>
  <td> <%= emp_record.last_name %> </td>
  <td> <%= emp_record.first_name %> </td>
  </tr>
<%  END LOOP; %>
```

Returning XML, Text, and Other Document Types By default, the PL/SQL gateway transmits files as HTML documents so that the browser interprets the HTML tags. If you want the browser to interpret the document as XML, plain text (with no formatting), or some other document type, then include the following directive:

```
<%@ page contentType="MIMEtype" %>
```

The attribute name is case-sensitive, so be sure to capitalize it as `contentType`. Insert `text/html`, `text/xml`, `text/plain`, `image/jpeg`, or some other MIME type that the browser or other client program recognizes. Users might have to configure their browsers to recognize some MIME types. The following shows an example of a directive for an Excel spreadsheet:

```
<%@ page contentType="application/vnd.ms-excel" %>
```

Typically, a PL/SQL server page is intended to be displayed in a web browser. It can also be retrieved and interpreted by a program that can make HTTP requests, such as a a Java or Perl client.

Returning Pages Containing Different Character Sets By default, the PL/SQL gateway transmits files with the character set defined by the PL/SQL gateway. To convert the data to a different character set for browser display, include the following directive:

```
<%@ page charset="encoding" %>
```

Specify `Shift_JIS`, `Big5`, `UTF-8`, or another encoding that the client program recognizes.

You must also configure the character set setting in the database accessor descriptor (DAD) of the PL/SQL gateway. Users might 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, but the web browsers are configured to display `Shift_JIS` encoding.

Handling Script Errors

When writing PL/SQL server pages, be mindful of the following types of errors:

- HTML syntax errors. Any errors in HTML markup are handled by the browser. The `loadpsp` utility does not check for them.
- PL/SQL syntax errors. If you make a syntax error in the PL/SQL code, the `loadpsp` utility stops and displays the line number, column number, and a brief message. You must fix the error before continuing. Any previous version of the stored subprogram can be erased when you attempt to replace it with a script that contains a syntax error. You might want to use one database for prototyping and debugging, then load the final stored subprogram into a different database for production. You can switch databases using a command-line flag without changing any source code.
- Run-time errors. 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 PL/SQL server page. Use the `errorPage` attribute (the name is case-sensitive) of the `<%@ page` ... `%>` directive to specify the page name.

  The page for unhandled exceptions is a PL/SQL server page with extension `.psp`. The error subprogram does not receive any parameters, so to determine the cause of the error, it can invoke 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 subprogram.

The following example shows a directive that specifies `errors.psp` as the page to run when errors are encountered:

```
<%@ page language="PL/SQL" contentType="text/html" errorPage="errors.psp" %>
```

Accepting User Input

To set up parameter passing for a PL/SQL server page, include a directive with the following syntax:

```
<%@ plsql parameter="parameter name" [type="PL/SQL type"] [default="value"] %>
```

Example 11–9 shows an example of a script that includes the parameter directive.

By default, parameters are of type `VARCHAR2`. To use a different type, include a `type="`*`PL/SQL type`*`"` attribute within the directive, as in the following example:

```
<%@ plsql parameter="p_employee_id" type="NUMBER" %>
```

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`, as in the following example:

```
<%@ plsql parameter="p_last_name" default="null" %>
```

User input comes encoded in the URL that retrieves the HTML page. You can generate the URL by hard-coding it in an HTML link, or by invoking your page as the action of an HTML form. Your page receives the input as parameters to a PL/SQL stored subprogram. For example, assume that you change the first few lines of Example 11–1 to include a parameter directive as follows, and then load it into the database:

```
<%@ page language="PL/SQL" %>
<%@ page contentType="text/html" %>
<%@ plsql parameter="p_employee_id" default="null" type="NUMBER" %>
<%@ plsql procedure="show_employees" %>
<%!
  CURSOR emp_cursor IS
  SELECT last_name, first_name
  FROM hr.employees
  WHERE employee_id = p_employee_id
  ORDER BY last_name;
%>
```

If the PL/SQL gateway is configured so that you can execute procedures by invoking `http://www.host.com/pls/proc_name`, where `proc_name` is the name of a procedure, then you can pass `200` for parameter `p_employee_id` as follows:

```
http://www.host.com/pls/show_employees?p_employee_id=200
```

Naming the PL/SQL Stored Procedure

Each top-level PL/SQL server page corresponds to a stored procedure within the server. When you load the page with `loadpsp`, the utility creates a PL/SQL stored procedure. By default, the procedure is given the same name as the PSP script, except with the .psp extension removed. Thus, if your script is named `hello_world.psp`, then by default the utility creates a procedure named `hello_world`.

To give the procedure a name that is different from the script name, include the following directive, where *procname* is the name of a procedure:

```
<%@ plsql procedure="procname" %>
```

Example 11–1 includes the following directive, which gives the stored procedure the name `show_employees`:

```
<%@ plsql procedure="show_employees" %>
```

Thus, you can name the file `empnames.psp` or anything else that ends with `*.psp`, but the procedure is created as `show_employees`. It is the name of the procedure, not the name of the PSP script, that you include in the URL.

Including the Contents of Other Files

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. Insert the following directive at the point where the content of the other file is to appear, replacing *filename* with the name of the file to be included:

```
<%@ include file="filename" %>
```

The included file must have an extension other than `.psp`. You must specify exactly the same name in both the `include` directive and in the `loadpsp` command, including any relative path name such as `../include/`.

Because the files are processed when you load the stored procedure into the database, the substitution is performed only once, not whenever the page is served. Therefore, changes to the included files that occur after the page is loaded into the database are not displayed when the procedure is executed.

You can use the include feature to pull in libraries of code, such as a navigation banners, footers, tables of contents, and so forth into multiple files. Alternatively, you can use this feature as a macro capability to include the same section of script code in more than one place in a page. The following example includes an HTML footer:

```
<%@ include file="footer.htm" %>
```

The following characteristics of included files:

- You can use any names and extensions for the included files. For example, you can include a file called `products.txt`.
- If the included files contain PL/SQL scripting code, then 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 `loadpsp` utility, you must include the names of all included files also. Specify the names of included files before the names of any `.psp` files.

Declaring Global Variables in a PSP Script

You can use the `<%! ... %>` directive to define 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. The syntax for this directive is as follows:

```
<%! PL/SQL declaration;
    [ PL/SQL declaration; ] ... %>
```

The usual PL/SQL syntax is allowed within the block. The delimiters server as shorthand, enabling you to omit the DECLARE keyword. All declarations are available to the code later in the file. Example 11–1 includes the following cursor declaration:

```
<%!
  CURSOR emp_cursor IS
  SELECT last_name, first_name
  FROM hr.employees
  ORDER BY last_name;
%>
```

You can specify multiple declaration blocks; internally, they are all merged into a single block when the PSP file is created as a stored procedure.

You can also use explicit DECLARE blocks within the <% ... %> delimiters that are explained in "Specifying Executable Statements in a PSP Script" on page 11-10. These declarations are only visible to the following BEGIN/END block.

Note: To make things easier to maintain, keep all your directives and declarations together near the beginning of a PL/SQL server page.

Specifying Executable Statements in a PSP Script

You can use the <% ... %> code block directive to execute a set of PL/SQL statements when the stored procedure is run. The following code shows the syntax for executable statements:

```
<% PL/SQL statement;
   [ PL/SQL statement; ] ... %>
```

This element typically spans multiple lines, with individual PL/SQL statements ended by semicolons. The statements can include complete blocks, as in the following example, which invokes the OWA_UTIL.TABLEPRINT procedure:

```
<% OWA_UTIL.TABLEPRINT(CTABLE => 'hr.employees', CATTRIBUTES => 'border=2',
   CCOLUMNS => 'last_name,first_name', CCLAUSES => 'WHERE employee_id > 100'); %>
```

The statements can also be the bracketing parts of IF/THEN/ELSE or BEGIN/END blocks. When a code block is split into multiple directives, you can put HTML or other directives in the middle, and the middle pieces are conditionally executed when the stored procedure is run. The following code from Example 11–10 provides an illustration of this technique:

```
<% FOR ITEM IN (SELECT product_name, list_price, catalog_url
                FROM product_information
                WHERE list_price IS NOT NULL
                ORDER BY list_price DESC) LOOP
   IF item.list_price > p_minprice THEN
      v_color := '#CCCCFF';
   ELSE
      v_color := '#CCCCCC';
   END IF;
%>
<TR BGCOLOR="<%= v_color %>">
  <TD><A HREF="<%= item.catalog_url %>"><%= item.product_name %></A></TD>
  <TD><BIG><%= item.list_price %></BIG></TD>
</TR>
<% END LOOP; %>
```

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.

Note: To share procedures, constants, and types across different PL/SQL server pages, compile them into a package in the database by 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.

Substituting Expression Values in a PSP Script

An expression directive outputs a single PL/SQL expression, such as a string, arithmetic expression, function call, or combination of these things. The result is substituted as a string at that spot in the HTML page that is produced by the stored procedure. The expression result must be a string value or be able to be cast to a string. For any types that cannot be implicitly cast, such as DATE, pass the value to the PL/SQL TO_CHAR function.

The syntax of an expression directive is as follows, where the *expression* placeholder is replaced by the desired expression:

```
<%= expression %>
```

You do not need to end the PL/SQL expression with a semicolon.

Example 11–1 includes a directive to print the value of a variable in a row of a cursor:

```
<%= emp_record.last_name %>
```

Compare the preceding example to the equivalent htp.print call in the following example (note especially the semicolon that ends the statement):

```
<% HTP.PRN (emp_record.last_name); %>
```

The content within the <%= ... %> delimiters is processed by the HTP.PRN function, which trims leading or trailing whitespace and requires that you quote literal strings.

You can use concatenation by using the twin pipe symbol (||) as in PL/SQL. The following directive shows an example of concatenation:

```
<%= 'The employee last name is ' || emp_record.last_name %>
```

Quoting and Escaping Strings in a PSP Script

PSP attributes use double quotes to delimit data. When values specified in PSP attributes are used for PL/SQL operations, they are passed exactly as you specify them in the PSP file. Thus, if PL/SQL requires a single-quoted string, then you must specify the string with the single quotes around it—and surround the whole thing with double quotes.

For example, your PL/SQL procedure might use the string Babe Ruth as the default value for a variable. For the string to be used in PL/SQL, you must enclose it in single quotes as 'Babe Ruth'. If you specify this single-quoted string in the default attribute of a PSP directive, you must enclose it in double quotes as in the following example:

```
<%@ plsql parameter="in_players" default="'Babe Ruth'" %>
```

You can also nest single-quoted strings inside single quotes. In this case, you must *escape* the nested single quotes by specifying the sequence `\'`. For example:

```
<%@ plsql parameter="in_players" default="'Walter \'Big Train\' Johnson'" %>
```

You can include most characters and character sequences in a PSP file without having them changed by the PSP loader. To include the sequence `%>`, specify the escape sequence `%\>`. To include the sequence `<%`, specify the escape sequence `<\%`. For example:

```
<%= 'The %\> sequence is used in scripting language: ' || lang_name %>
<%= 'The <\% sequence is used in scripting language: ' || lang_name %>
```

Including Comments in a PSP Script

To put a comment in the HTML portion of a PL/SQL server page for the benefit of those reading the PSP source code, use the following syntax:

```
<%-- PSP comment text --%>
```

Comments in the preceding form do not appear in the HTML output from the PSP and also do not appear when you query the PL/SQL source code in `USER_OBJECTS`.

To create a comment that is visible in the HTML output and in the `USER_OBJECTS` source, place the comment in the HTML and use the normal HTML comment syntax:

```
<!-- HTML comment text -->
```

To include a comment inside a PL/SQL block within a PSP, and to make the comment invisible in the HTML output but visible in `USER_OBJECTS`, use the normal PL/SQL comment syntax, as in the following example:

```
-- Comment in PL/SQL code
```

Example 11–3 shows a fragment of a PSP file with the three types of comments.

Example 11–3 Sample Comments in a PSP File

```
<p>Today we introduce our new model XP-10.
<%--
  This is the project with code name "Secret Project".
  Users viewing the HTML page will not see this PSP script comment.
  The comment is not visible in the USER_OBJECTS source code.
--%>
<!--
  Some pictures of the XP-10.
  Users viewing the HTML page source will see this comment.
  The comment is also visible in the USER_OBJECTS source code.
-->
<%
FOR image_file IN (SELECT pathname, width, height, description
                   FROM image_library WHERE model_num = 'XP-10')
-- Comments interspersed with PL/SQL statements.
-- Users viewing the HTML page source will not see these PL/SQL comments.
-- These comments are visible in the USER_OBJECTS source code.
LOOP
%>
<img src="<%= image_file.pathname %>" width=<% image_file.width %>
height=<% image_file.height %> alt="<% image_file.description %>">
<br>
<% END LOOP; %>
```

Loading PL/SQL Server Pages into the Database

Use the `loadpsp` utility, which is located in `$ORACLE_HOME/bin`, to 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. The syntax of the `loadpsp` utility as follows:

```
loadpsp [ -replace ] -user username/password[@connect_string]
    [ include_file_name ... ] [ error_file_name ] psp_file_name ...
```

To create procedures with `CREATE OR REPLACE` syntax, use the `-replace` flag.

When you load a PSP file, the loader performs the following actions:

1. Logs on to the database with the specified user name, password, and net service name
2. Creates the stored procedures in the user schema

Include the names of all the include files before the names of the PL/SQL server pages. Also include the name of the file specified in the `errorPage` attribute of the `page` directive. These filenames on the `loadpsp` command line must match exactly the names specified within the PSP `include` and `page` directives, including any relative path name such as `../include/`. Example 11–4 shows a sample PSP load command.

Example 11–4 Loading PL/SQL Server Pages

```
loadpsp -replace -user hr/hr@orcl banner.inc error.psp display_order.psp
```

Example 11–4 has the following characteristics:

- The stored procedure is created in the database `orcl`. The database is accessed as user `hr` with password `hr`, 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 occurs when the PSP is loaded into the database, not when the stored procedure is executed.
- `error.psp` is a file containing code, text, or both that is processed when an unhandled exception occurs, to present a friendly page rather than an internal error message.
- `display_order.psp` contains the main code and text for the web page. By default, the corresponding stored procedure is named `display_order`.

Querying PL/SQL Server Pages Source Code

After you have loaded a PSP file, you can see the source code by querying the static data dictionary views `*_SOURCE`. For example, suppose that you load the script in Example 11–1 with the following command:

```
loadpsp -replace -user hr/hr simple.psp
```

If you log on to the database as user `hr`, then you can execute the following query in SQL*Plus to view the source code of the PSP:

```
SET HEADING OFF

SELECT TEXT
FROM   USER_SOURCE
WHERE  NAME = 'SHOW_EMPLOYEES'
```

```
ORDER BY LINE;
```

Sample output is shown in Example 11–5. The code generated by `loadpsp` is different from the code in the source file. The `loadpsp` utility has added extra code, mainly calls to the `HTP` package, to the PSP code. The `HTP` package generates the HTML tags for the web page.

Example 11–5 Output from Query of USER_SOURCE

```
PROCEDURE show_employees  AS

  CURSOR emp_cursor IS
  SELECT last_name, first_name
  FROM hr.employees
  ORDER BY last_name;

 BEGIN NULL;
owa_util.mime_header('text/html'); htp.prn('
');
htp.prn('
');
htp.prn('

');
htp.prn('
<html>
<head>
<meta http-equiv="Content-Type" content="text/html">
<title>List of Employees</title>
</head>
<body TEXT="#000000" BGCOLOR="#FFFFFF">
<h1>List of Employees</h1>
<table width="40%" border="1">
<tr>
<th align="left">Last Name</th>
<th align="left">First Name</th>

</tr>
');
  FOR emp_record IN emp_cursor LOOP
htp.prn('
  <tr>
  <td> ');
htp.prn( emp_record.last_name );
htp.prn(' </td>
  <td> ');
htp.prn( emp_record.first_name );
htp.prn(' </td>
  </tr>
');
  END LOOP;
htp.prn('
</table>
</body>
</html>
');
 END;
```

Executing PL/SQL Server Pages Through URLs

After the PL/SQL server page is turned into a stored procedure, you can run the procedure 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 PL/SQL 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 nonalphanumeric characters in encoded format (such as `%20` for a space). You can use the `GET` method to invoke a PSP page from an HTML form, or you can use a hard-coded HTML link to invoke the stored procedure with a given set of parameters.

Using `METHOD=GET`, the syntax of the URL looks something like the following:

```
http://sitename/schemaname/procname?parmname1=value1&parmname2=value2
```

For example, the following URL includes a `p_lname` and `p_fname` parameter:

```
http://www.host.com/pls/show_employees?p_lname=Ashdown&p_fname=Lance
```

Using `METHOD=POST`, the syntax of the URL does not show the parameters:

```
http://sitename/schemaname/procname
```

For example, the following URL specifies a procedure name but does not pass parameters:

```
http://www.host.com/pls/show_employees
```

The `METHOD=GET` format is more convenient for debugging and allows visitors to pass exactly the same parameters when they return to the page through a bookmark.

The `METHOD=POST` format allows a larger volume of parameter data, and is suitable for passing sensitive information that must 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 invoked 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 follow the instructions in "Loading PL/SQL Server Pages into the Database" on page 11-13 and "Executing PL/SQL Server Pages Through URLs" on page 11-15 to test the examples.

Topics:

- Setup for PL/SQL Server Pages Examples
- Printing the Sample Table with a Loop
- Allowing a User Selection
- Using an HTML Form to Invoke a PL/SQL Server Page
- Including JavaScript in a PSP File

Setup for PL/SQL Server Pages Examples

These examples use the `product_information` table in the `oe` schema, which is described as follows:

```
Table PRODUCT_INFORMATION
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PRODUCT_ID                                NOT NULL NUMBER(6)
 PRODUCT_NAME                                       VARCHAR2(50)
 PRODUCT_DESCRIPTION                                VARCHAR2(2000)
 CATEGORY_ID                                        NUMBER(2)
 WEIGHT_CLASS                                       NUMBER(1)
 WARRANTY_PERIOD                                    INTERVAL YEAR(2) TO MONTH
 SUPPLIER_ID                                        NUMBER(6)
 PRODUCT_STATUS                                     VARCHAR2(20)
 LIST_PRICE                                         NUMBER(8,2)
 MIN_PRICE                                          NUMBER(8,2)
 CATALOG_URL                                        VARCHAR2(50)
```

The examples assume the following:

- Youn have set up mod_plsql as described in "Using mod_plsql Gateway to Map Client Requests to a PL/SQL Web Application" on page 10-4.
- You have created a DAD for static authentication of the `oe` user.
- You can access PL/SQL stored procedures created in the `oe` schema through the following URL, where `proc_name` is the name of a stored procedure:`http://www.host.com/pls/proc_name`

For debugging purposes, you can display the complete contents of an SQL table. You can do this with a single call to `OWA_UTIL.TABLEPRINT` as illustrated in Example 11–6. In subsequent iterations, we use other techniques to gain more control over the presentation.

Example 11–6 show_prod_simple.psp

```
<%@ plsql procedure="show_prod_simple" %>
<HTML>
<HEAD><TITLE>Show Contents of product_information (Complete Dump)</TITLE></HEAD>
<BODY>
<%
DECLARE
  dummy BOOLEAN;
BEGIN
  dummy := OWA_UTIL.TABLEPRINT('oe.product_information','border');
END;
%>
</BODY>
</HTML>
```

Load the PSP in Example 11–6 at the command line as follows:

```
loadpsp -replace -user oe/oe show_prod_simple.psp
```

Access the PSP through the following URL:

```
http://www.host.com/pls/show_prod_simple
```

Printing the Sample Table with a Loop

Example 11–6 loops through the items in the `product_information` table and adjusts the `SELECT` statement to retrieve only a subset of the rows or columns. In this example, we pick a very simple presentation, a set of list items, to avoid any problems from mismatched or unclosed table tags.

Example 11–7 show_catalog_raw.psp

```
<%@ plsql procedure="show_prod_raw" %>
<HTML>
<HEAD><TITLE>Show Products (Raw Form)</TITLE></HEAD>
<BODY>
<UL>
<% FOR item IN (SELECT product_name, list_price, catalog_url
                FROM product_information
                WHERE list_price IS NOT NULL
                ORDER BY list_price DESC) LOOP %>
<LI>
Item = <%= item.product_name %><BR>
Price = <%= item.list_price %><BR>
URL = <%= item.catalog_url %><BR>
<% END LOOP; %>
</UL>
</BODY>
</HTML>
```

Example 11–8 shows a more sophisticated variation of Example 11–7 in which formatting is added to the HTML to improve the presentation.

Example 11–8 show_catalog_pretty.psp

```
<%@ plsql procedure="show_prod_pretty" %>
<HTML>
<HEAD><TITLE>Show Products (Better Form)</TITLE></HEAD>
<BODY>
<UL>
<% FOR item IN (SELECT product_name, list_price, catalog_url
                FROM product_information
                WHERE list_price IS NOT NULL
                ORDER BY list_price DESC) LOOP %>
<LI>
Item = <A HREF=<%= item.catalog_url %>><%= item.product_name %></A><BR>
Price = <BIG><%= item.list_price %></BIG><BR>
<% END LOOP; %>
</UL>
</BODY>
</HTML>
```

Allowing a User Selection

In the previous examples, the HTML page remains the same unless the `product_information` table is updated. Example 11–9 livens up the page by:

- Making it accept a minimum price, and present only the items that are more expensive. (Your customers' buying criteria might vary.)
- Setting the default minimum price to 100 units of the appropriate currency. Later, we see how to allow the user to pick a minimum price.

Example 11–9 show_product_partial.psp

```
<%@ plsql procedure="show_product_partial" %>
<%@ plsql parameter="p_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 <%= p_minprice %>.
<UL>
<% FOR ITEM IN (SELECT product_name, list_price, catalog_url
                FROM product_information
                WHERE list_price > p_minprice
                ORDER BY list_price DESC)
   LOOP %>
<LI>
Item = <A HREF="<%= item.catalog_url %>"><%= item.product_name %></A><BR>
Price = <BIG><%= item.list_price %></BIG><BR>
<% END LOOP; %>
</UL>
</BODY>
</HTML>
```

After loading Example 11–9 into the database, you can pass a parameter to the `show_product_partial` procedure through a URL. The following example specifies a minimum price of 250:

```
http://www.host.com/pls/show_product_partial?p_minprice=250
```

Filtering results is appropriate for applications such as search results, where users might be overwhelmed by choices. But in a retail situation, you might want to use the alternative technique illustrated in Example 11–10, so that customers can still choose to purchase other items:

- Instead of filtering the results through a `WHERE` clause, retrieve the entire result set and then take different actions for different returned rows.
- Change the HTML to highlight the output that meets their criteria. Example 11–10 uses the background color for an HTML table row. You can also insert a special icon, increase the font size, or use another technique to call attention to the most important rows.
- Present the results in an HTML table.

Example 11–10 show_product_highlighed.psp

```
<%@ plsql procedure="show_product_highlighted" %>
<%@ plsql parameter="p_minprice" default="100" %>
<%! v_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 <%= p_minprice %>.
<P>
<TABLE BORDER>
  <TR>
    <TH>Product</TH>
    <TH>Price</TH>
  </TR>
  <% FOR ITEM IN (SELECT product_name, list_price, catalog_url
                  FROM product_information
                  WHERE list_price IS NOT NULL
                  ORDER BY list_price DESC) LOOP
     IF item.list_price > p_minprice THEN
        v_color := '#CCCCFF';
     ELSE
        v_color := '#CCCCCC';
     END IF;
  %>
  <TR BGCOLOR="<%= v_color %>">
    <TD><A HREF="<%= item.catalog_url %>"><%= item.product_name %></A></TD>
    <TD><BIG><%= item.list_price %></BIG></TD>
  </TR>
  <% END LOOP; %>
</TABLE>
</BODY>
</HTML>
```

Using an HTML Form to Invoke a PL/SQL Server Page

Example 11–11 shows a bare-bones HTML form that allows the user to enter a price. The form invokes the `show_product_partial` stored procedure illustrated in Example 11–9 and passes it the entered value as the `p_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 resides in the same directory as the PSP file that it invokes. Even though this HTML file contains no PL/SQL code, we can give it a `.psp` extension and load it as a stored procedure into the database. When the `product_form` stored procedure is executed through a URL, it displays the HTML exactly as it appears in the file.

Example 11–11 product_form.psp

```
<HTML>
<BODY>
<FORM method="POST" action="show_product_partial">
  <P>Enter the minimum price you want to pay:
  <INPUT type="text" name="p_minprice">
  <INPUT type="submit" value="Submit">
</FORM>
</BODY>
</HTML>
```

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.

Example 11–12 shows a version of Example 11–9 that uses JavaScript to display the order status in the browser status bar when the user moves his or her mouse over the product URL.

Example 11–12 show_product_javascript.psp

```
<%@ plsql procedure="show_product_javascript" %>
<%@ plsql parameter="p_minprice" default="100" %>
<HTML>
<HEAD>
  <TITLE>Show Items Greater Than Specified Price</TITLE>
```

```
<SCRIPT language="JavaScript">
<!--hide

var text=" ";

function overlink (text)
{
  window.status=text;
}
function offlink (text)
{
  window.status=text;
}

//-->
</SCRIPT>

</HEAD>
<BODY>
<P>This report shows the items whose price is greater than <%= p_minprice %>.
<P>
<UL>
<% FOR ITEM IN (SELECT product_name, list_price, catalog_url, product_status
                FROM product_information
                WHERE list_price > p_minprice
                ORDER BY list_price DESC)
   LOOP %>
<LI>
Item =
  <A HREF="<%= item.catalog_url %>"
  onMouseover="overlink('PRODUCT STATUS: <%= item.product_status %>');return true"
  onMouseout="offlink(' ');return true">
    <%= item.product_name %>
  </A>
<BR>
Price = <BIG><%= item.list_price %></BIG><BR>
<% END LOOP; %>
</UL>
</BODY>
</HTML>
```

Debugging PL/SQL Server Pages

As you begin experimenting with PL/SQL Server Pages, 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.
  - 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.

- When using a URL to request a PSP, you might get an error that the file is not found. In this case:
  - Be sure you are requesting the right virtual path, depending on the way the web gateway is configured. Typically, the path includes the host name, optionally a port number, the schema name, and the name of the stored procedure (with no `.psp` extension).
  - 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, 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.
  - When you get one file-not-found error, request the latest version of the page the next time. The error page might be cached by the browser. You might need to force a page reload in the browser to bypass the 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 difficult part is to configure the interface between different HTML forms, scripts, and CGI programs so that the right values are passed into your page. The page might return an error because of a parameter mismatch.

  Guidelines:
  - To determine exactly what is being passed to your page, use `METHOD=GET` in the invoking form so that the parameters are visible in the URL.
  - Ensure that the form or CGI program that invokes 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, then the PSP file must declare equivalent parameters.
  - Ensure 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`.
  - Ensure that the query string of the URL consists of name-value pairs, separated by equals signs, especially if you are passing parameters by constructing a hard-coded link to the page.
  - If you are passing a lot of parameter data, such as large strings, you might exceed the volume that can be passed with `METHOD=GET`. You can switch to `METHOD=POST` in the invoking form without changing your PSP file.
  - Although the `loadpsp` command reports line numbers correctly when there is a syntax error in your source file, line numbers reported for run-time errors refer to a transformed version of the source and do not match the line numbers in the original source. When you encounter errors that produce an error trace instead of the expected web page, you must locate the error through exception handlers and by printing debug output.

Putting PL/SQL Server Pages into Production

Before putting your PSP application into production, consider 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.
  - If you set one color (such as the foreground text color), 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 an alternative technique. For example, you might put an icon next to special items in a table. Some users might see your page on a monochrome screen or on browsers that cannot represent different colors.
- 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 call the database.
- Browsers tend to be lenient when displaying incorrect HTML. What looks OK in one browser might look bad or might not display at all in another browser.

  Guidelines:
  - 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 must 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.

12 Using Continuous Query Notification

Continuous Query Notification (CQN) allows an application to register queries with the database for either object change notification (the default) or query result change notification. An object referenced by a registered query is a **registered object**.

If a query is registered for **object change notification (OCN)**, the database notifies the application whenever a transaction changes an object that the query references and commits, whether or not the query result changed.

If a query is registered for **query result change notification (QRCN)**, the database notifies the application whenever a transaction changes the result of the query and commits.

A **CQN registration** associates a list of one or more queries with a notification type (OCN or QRCN) and a notification handler. To create a CQN registration, you can use either the PL/SQL interface or the OCI interface. If you use the PL/SQL interface, the notification handler is a server-side PL/SQL stored procedure; if you use the OCI interface, the notification handler is a client-side C callback procedure.

This chapter explains general CQN concepts and explains how to use the PL/SQL CQN interface. For information about using OCI for CQN, see *Oracle Call Interface Programmer's Guide*.

Topics:

Note: The terms **OCN** and **QRCN** refer to both the notification type and the notification itself: An application registers a query *for OCN*, and the database sends the application *an OCN*; an application registers a query *for QRCN*, and the database sends the application *a QRCN*.

Object Change Notification (OCN)

If an application registers a query for object change notification (OCN), the database sends the application an OCN whenever a transaction changes an object associated with the query and commits, whether or not the result of the query changed.

For example, if an application registers the query in Example 12–1 for OCN, and a user commits a transaction that changes the EMPLOYEES table, the database sends the application an OCN, even if the changed row or rows did not satisfy the query predicate (for example, if DEPARTMENT_ID = 5).

Example 12–1 Query to be Registered for Change Notification

```
SELECT EMPLOYEE_ID, SALARY FROM EMPLOYEES
   WHERE DEPARTMENT_ID = 10;
```

Query Result Change Notification (QRCN)

Note: For QRCN support, the COMPATIBLE initialization parameter of the database must be at least 11.0.0, and Automatic Undo Management (AUM) must be enabled (as it is by default).

For information about the COMPATIBLE initialization parameter, see *Oracle Database Administrator's Guide*.

For information about AUM, see *Oracle Database Administrator's Guide*.

If an application registers a query for query result change notification (QRCN), the database sends the application a QRCN whenever a transaction changes the result of the query and commits.

For example, if an application registers the query in Example 12–1 for QRCN, the database sends the application a QRCN only if the query result set changes; that is, if one of the following DML statements commits:

- An INSERT or DELETE of a row that satisfies the query predicate (DEPARTMENT_ID = 10).
- An UPDATE to the EMPLOYEE_ID or SALARY column of a row that already satisfied the query predicate (DEPARTMENT_ID = 10).
- An UPDATE to the DEPARTMENT_ID column of a row that changed its value from 10 to a value other than 10, causing the row to be deleted from the result set.
- An UPDATE to the DEPARTMENT_ID column of a row that changed its value to 10 from a value other than 10, causing the row to be added to the result set.

The default notification type is OCN. For QRCN, specify QOS_QUERY in the QOSFLAGS attribute of the CQ_NOTIFICATION$_REG_INFO object.

With QRCN, you have a choice of guaranteed mode (the default) or best-effort mode.

Topics:

- Guaranteed Mode
- Best-Effort Mode

Guaranteed Mode

In guaranteed mode, there are no false positives: the database sends the application a QRCN only when the query result set is guaranteed to have changed.

For example, suppose that an application registered the query in Example 12–1 for QRCN, that employee 201 is in department 10, and that the following statements are executed:

```
UPDATE EMPLOYEES SET SALARY = SALARY + 10 WHERE EMPLOYEE_ID = 201;
UPDATE EMPLOYEES SET SALARY = SALARY - 10 WHERE EMPLOYEE_ID = 201;
COMMIT;
```

Each `UPDATE` statement in the preceding transaction changes the query result set, but together they have no effect on the query result set; therefore, the database does not send the application a QRCN for the transaction.

For guaranteed mode, specify `QOS_QUERY`, but not `QOS_BEST_EFFORT`, in the `QOSFLAGS` attribute of the `CQ_NOTIFICATION$_REG_INFO` object.

Some queries are too complex for QRCN in guaranteed mode. For the characteristics of queries that can be registered in guaranteed mode, see "Queries that Can Be Registered for QRCN in Guaranteed Mode" on page 12-15.

Best-Effort Mode

Some queries that are too complex for guaranteed mode can be registered for QRCN in best-effort mode, in which CQN creates and registers simpler versions of them.

For example, the query in Example 12–2 is too complex for QRCN in guaranteed mode because it contains the aggregate function `SUM`.

Example 12–2 Query Too Complex for QRCN in Guaranteed Mode

```
SELECT SUM(SALARY) FROM EMPLOYEES
  WHERE DEPARTMENT_ID = 20;
```

In best-effort mode, CQN registers the following simpler version of the query in Example 12–2:

```
SELECT SALARY FROM EMPLOYEES
  WHERE DEPARTMENT_ID = 20;
```

Whenever the result of the original query changes, the result of its simpler version also changes; therefore, no notifications are lost due to the simplification. However, the simplification might cause false positives, because the result of the simpler version can change when the result of the original query does not.

In best-effort mode, the database does the following:

- Minimizes the OLTP response overhead that is due to notification-related processing, as follows:
  - For a single-table query, the database determines whether the query result has changed by which columns changed and which predicates the changed rows satisfied.

  – For a multiple-table query (a join), the database uses the primary-key/foreign-key constraint relationships between the tables to determine whether the query result has changed.

- Sends the application a QRCN whenever a DML statement changes the query result set, even if a subsequent DML statement nullifies the change made by the first DML statement.

As a result of its overhead minimization, best-effort mode infrequently causes false positives, even for queries that CQN does not simplify. For example, consider the query in Example 12–1 and the transaction in "Guaranteed Mode" on page 12-3. In best-effort mode, CQN does not simplify the query, but the transaction generates a false positive.

Some types of queries are so simplified that invalidations are generated at object level; that is, whenever any object referenced in those queries changes. Examples of such queries are those that use unsupported column types or include subqueries. The solution to this problem is to rewrite the original queries.

For example, the query in Example 12–3 is too complex for QRCN in guaranteed mode because it includes a subquery.

Example 12–3 Query Whose Simplified Version Invalidates Objects

```
SELECT SALARY FROM EMPLOYEES
  WHERE DEPARTMENT_ID IN
    (SELECT DEPARTMENT_ID FROM DEPARTMENTS
       WHERE LOCATION_ID = 1700
    );
```

In best-effort mode, CQN simplifies the query in Example 12–3 to this:

```
SELECT * FROM EMPLOYEES, DEPARTMENTS;
```

The simplified query can cause objects to be invalidated. However, if you rewrite the original query as follows, you can register it in either guaranteed mode or best-effort mode:

```
SELECT SALARY FROM EMPLOYEES, DEPARTMENTS
  WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID
    AND DEPARTMENTS.LOCATION_ID = 1700;
```

Queries that can be registered only in best-effort mode are described in "Queries that Can Be Registered for QRCN Only in Best-Effort Mode" on page 12-16.

The default for QRCN mode is guaranteed mode. For best-effort mode, specify `QOS_BEST_EFFORT` in the `QOSFLAGS` attribute of the `CQ_NOTIFICATION$_REG_INFO` object.

Events that Generate Notifications

The following events generate notifications:

- Committed DML Transactions
- Committed DDL Statements
- Deregistration
- Global Events

Committed DML Transactions

When the notification type is OCN, any DML transaction that changes one or more registered objects generates one notification for each object when it commits.

When the notification type is QRCN, any DML transaction that changes the result of one or more registered queries generates a notification when it commits. The notification includes the query IDs of the queries whose results changed.

For either notification type, the notification includes:

- Name of each changed table
- Operation type (`INSERT`, `UPDATE`, or `DELETE`)
- `ROWID` of each changed row, if the registration was created with the `ROWID` option and the number of modified rows was not too large. For more information, see "ROWID Option" on page 12-12.

Committed DDL Statements

For both OCN and QRCN, the following DDL statements, when committed, generate notifications:

- `ALTER TABLE`
- `TRUNCATE TABLE`
- `FLASHBACK TABLE`
- `DROP TABLE`

Note: When the notification type is OCN, a committed `DROP TABLE` statement generates a `DROP NOTIFICATION`.

Any OCN registrations of queries on the dropped table become disassociated from that table (which no longer exists), but the registrations themselves continue to exist. If any of these registrations are associated with objects other than the dropped table, committed changes to those other objects continue to generate notifications. Registrations associated only with the dropped table also continue to exist, and their creator can add queries (and their referenced objects) to them.

An OCN registration is based on the version and definition of an object at the time the query was registered. If an object is dropped, registrations on that object are disassociated from it forever. If a new object is created with the same name, and in the same schema, as the dropped object, the new object is not associated with OCN registrations that were associated with the dropped object.

When the notification type is QRCN:

- The notification includes the following:
  - Query IDs of the queries whose results have changed
  - Name of the modified table
  - Type of DDL operation

- Some DDL operations that invalidate registered queries can cause those queries to be deregisted.

  For example, suppose that the following query is registered for QRCN:

  ```
  SELECT COL1 FROM TEST_TABLE
    WHERE COL2 = 1;
  ```

  Suppose that `TEST_TABLE` has the following schema:

  ```
  (COL1 NUMBER, COL2 NUMBER, COL3 NUMBER)
  ```

  The following DDL statement, when committed, invalidates the query and causes it to be removed from the registration:

  ```
  ALTER TABLE DROP COLUMN COL2;
  ```

Deregistration

For both OCN and QRCN, deregistration—removal of a registration from the database—generates a notification. The reasons that the database removes a registration are:

- Timeout

  If `TIMEOUT` is specified with a nonzero value when the queries are registered, the database purges the registration after the specified time interval.

  If `QOS_DEREG_NFY` is specified when the queries are registered, the database purges the registration after it generates its first notification.

- Loss of privileges

  If privileges are lost on an object associated with a registered query, and the notification type is OCN, the database purges the registration. (When the notification type is QRCN, the database removes that query from the registration, but does not purge the registration.)

  For privileges needed to register queries, see "Prerequisites for Creating CQN Registrations" on page 12-14.

A notification is not generated when a client application performs an explicit deregistration.

Global Events

The global events `EVENT_STARTUP` and `EVENT_SHUTDOWN` generate notifications.

In an Oracle RAC environment, the following events generate notifications:

- `EVENT_STARTUP` when the first instance of the database starts up
- `EVENT_SHUTDOWN` when the last instance of the database shuts down
- `EVENT_SHUTDOWN_ANY` when any instance of the database shuts down

The preceding global events are constants defined in the `DBMS_CQ_NOTIFICATION` package.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_CQ_NOTIFICATION` package

Notification Contents

A notification contains some or all of the following information:

- Type of event, which is one of the following:
  - Startup
  - Object change
  - Query result change
  - Deregistration
  - Shutdown
- Registration ID of affected registration
- Names of changed objects
- If `ROWID` option was specified, `ROWID`s of changed rows
- If the notification type is QRCN: Query IDs of queries whose results changed
- If notification resulted from a DML or DDL statement:
  - Array of names of modified tables
  - Operation type (for example, `INSERT` or `UPDATE`)

A notification does not contain the changed data itself. For example, the notification does not say that a monthly salary increased from 5000 to 6000. To obtain more recent values for the changed objects or rows or query results, the application must query the database.

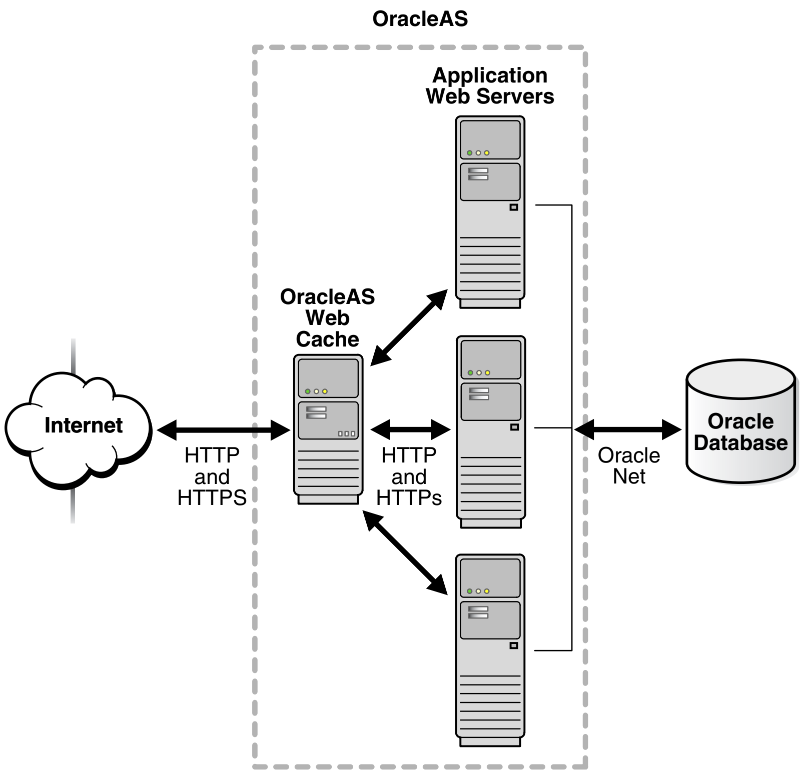
Good Candidates for CQN

Good candidates for CQN are applications that cache the result sets of queries on infrequently changed objects in the middle tier, to avoid network round trips to the database. These applications can use CQN to register the queries to be cached. When such an application receives a notification, it can refresh its cache by re-executing the registered queries.

An example of such an application is a web forum. Because its users do not need to view new content as soon as it is inserted into the database, this application can cache information in the middle tier and have CQN tell it when it when to refresh the cache.

Figure 12–1 illustrates a typical scenario in which an Oracle Database serves data that is cached in the middle tier and then accessed over the Internet.

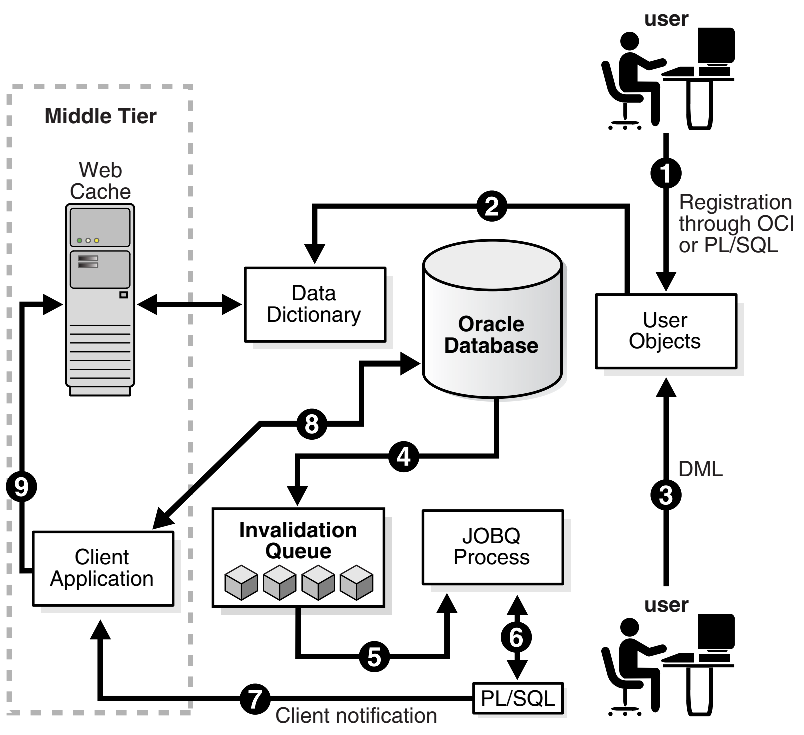
Figure 12–1 Middle-Tier Caching



Applications in the middle tier require rapid access to cached copies of database objects while keeping the cache as current as possible in relation to the database. Cached data becomes obsolete when a transaction modifies the data and commits, thereby putting the application at risk of accessing incorrect results. If the application uses CQN, Oracle Database can publish a notification when a change occurs to registered objects with details on what changed. In response to the notification, the application can refresh cached data by fetching it from the back-end database.

Figure 12–2 illustrates the process by which middle-tier Web clients receive and process notifications.

Figure 12–2 Basic Process of Continuous Query Notification (CQN)



Explanation of steps in Figure 12–2 (assuming that registrations are created using PL/SQL and that the application has cached the result set of a query on `HR.EMPLOYEES`):

1. The developer uses PL/SQL to create a CQN registration for the query, which consists of creating a stored PL/SQL procedure to process notifications and then using the PL/SQL CQN interface to create a registration for the query, specifying the PL/SQL procedure as the notification handler.
2. The database populates the registration information in the data dictionary.
3. A user updates a row in the `HR.EMPLOYEES` table in the back-end database and commits the update, causing the query result to change. The data for `HR.EMPLOYEES` cached in the middle tier is now outdated.
4. Oracle Database adds a message that describes the change to an internal queue.
5. Oracle Database notifies a `JOBQ` background process of a new notification message.
6. The `JOBQ` process executes the stored procedure specified by the client application. In this example, `JOBQ` passes the data to a server-side PL/SQL procedure. The implementation of the PL/SQL notification handler determines how the notification is handled.
7. Inside the server-side PL/SQL procedure, the developer can implement logic to notify the middle-tier client application of the changes to the registered objects. For example, it notifies the application of the `ROWID` of the changed row in `HR.EMPLOYEES`.
8. The client application in the middle tier queries the back-end database to retrieve the data in the changed row.
9. The client application updates the cache with the new data.

Creating CQN Registrations

A **CQN registration** associates a list of one or more queries with a notification type and a notification handler.

The notification type is either OCN or QRCN. For information about these types, see "Object Change Notification (OCN)" on page 12-2 and "Query Result Change Notification (QRCN)" on page 12-2.

To create a CQN registration, you can use either the PL/SQL interface or the OCI interface. If you use the PL/SQL interface, the notification handler is a server-side PL/SQL stored procedure; if you use the OCI interface, the notification handler is a client-side C callback procedure. (This topic explains only the PL/SQL interface. For information about the OCI interface, see *Oracle Call Interface Programmer's Guide*.)

Once created, a registration is stored in the Oracle Database. In an Oracle RAC environment, it is visible to all database instances. Transactions that change the query results in any database instance generate notifications.

By default, a registration survives until the application that created it explicitly deregisters it or until the database implicitly purges it (due to loss of privileges, for example).

Topics:

- PL/SQL CQN Registration Interface
- CQN Registration Options
- Prerequisites for Creating CQN Registrations
- Queries that Can Be Registered for Object Change Notification (OCN)
- Queries that Can Be Registered for Query Result Change Notification (QRCN)
- Using PL/SQL to Register Queries for CQN
- Best Practices for CQN Registrations
- Troubleshooting CQN Registrations

PL/SQL CQN Registration Interface

The PL/SQL CQN registration interface is implemented with the `DBMS_CQ_NOTIFICATION` package. You use the `DBMS_CQ_NOTIFICATION.NEW_REG_START` function to open a registration block. You specify the registration details, including the notification type and notification handler, as part of the `CQ_NOTIFICATION$_REG_INFO` object, which is passed as an argument to the `NEW_REG_START` procedure. Every query that you execute while the registration block is open is registered with CQN. If you specified notification type QRCN, the database assigns a query ID to each query. You can retrieve these query IDs with the `DBMS_CQ_NOTIFICATION.CQ_NOTIFICATION_QUERYID` function. To close the registration block, you use the `DBMS_CQ_NOTIFICATION.REG_END` function.

For step-by-step instructions, see "Using PL/SQL to Register Queries for CQN" on page 12-18.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_CQ_NOTIFICATION` package

CQN Registration Options

You can change the CQN registration defaults with the options summarized in Table 12–1.

Table 12–1 Continuous Query Notification Registration Options

Option	Description
Notification Type	Specifies QRCN (the default is OCN).
QRCN Mode[1]	Specifies best-effort mode (the default is guaranteed mode).
`ROWID`	Includes the `ROWID` of each changed row in the notification.
Operations Filter[2]	Publishes the notification only if the operation type matches the specified filter condition.
Transaction Lag[2]	Deprecated. Use Notification Grouping instead.
Notification Grouping	Specifies how notifications are grouped.
Reliable	Stores notifications in a persistent database queue (instead of in shared memory, the default).
Purge on Notify	Purges the registration after the first notification.
Timeout	Purges the registration after a specified time interval.

[1] Applies only when notification type is QRCN.
[2] Applies only when notification type is OCN.

Topics:

- Notification Type Option
- QRCN Mode (QRCN Notification Type Only)
- ROWID Option
- Operations Filter Option (OCN Notification Type Only)
- Transaction Lag Option (OCN Notification Type Only)
- Notification Grouping Options
- Reliable Option
- Purge-on-Notify and Timeout Options

Notification Type Option

The notification types are OCN (described in "Object Change Notification (OCN)" on page 12-2) and QRCN (described in "Query Result Change Notification (QRCN)" on page 12-2).

QRCN Mode (QRCN Notification Type Only)

The QRCN mode option applies only when the notification type is QRCN. Instructions for setting the notification type to QRCN are in "Notification Type Option" on page 12-11.

The QRCN modes are guaranteed (described in "Guaranteed Mode" on page 12-3) and best-effort (described in "Best-Effort Mode" on page 12-3).

The default is guaranteed mode. For best-effort mode, specify `QOS_BEST_EFFORT` in the `QOSFLAGS` attribute of the `CQ_NOTIFICATION$_REG_INFO` object.

ROWID Option

To include the ROWID option of each changed row in the notification, specify QOS_ROWIDS in the QOSFLAGS attribute of the CQ_NOTIFICATION$_REG_INFO object.

From the ROWID information in the notification, the application can retrieve the contents of the changed rows by performing queries of the following form:

```
SELECT * FROM table_name_from_notification
  WHERE ROWID = rowid_from_notification;
```

ROWIDs are published in the external string format. For a regular heap table, the length of a ROWID is 18 character bytes. For an Index Organized Table (IOT), the length of the ROWID depends on the size of the primary key, and might exceed 18 bytes.

If the server does not have enough memory for the ROWIDs, the notification might be "rolled up" into a FULL-TABLE-NOTIFICATION, indicated by a special flag in the notification descriptor. Possible reasons for a FULL-TABLE-NOTIFICATION are:

- Total shared memory consumption due to ROWIDs exceeds 1% of the dynamic shared pool size.
- Too many rows were changed in a single registered object within a transaction (the upper limit is approximately 80).
- Total length of the logical ROWIDs of modified rows for an IOT is too large (the upper limit is approximately 1800 bytes).
- You specified the Notification Grouping option NTFN_GROUPING_TYPE with the value DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_SUMMARY, described in "Notification Grouping Options" on page 12-13.

Because a FULL-TABLE-NOTIFICATION does not include ROWIDs, the application that receives it must assume that the entire table (that is, all rows) might have changed.

Operations Filter Option (OCN Notification Type Only)

The Operations Filter option applies only when the notification type is OCN.

The Operations Filter option allows you to specify the types of operations that generate notifications.

The default is all operations. To specify that only some operations generate notifications, use the OPERATIONS_FILTER attribute of the CQ_NOTIFICATION$_REG_INFO object. With the OPERATIONS_FILTER attribute, specify the type of operation with the constant that represents it, which is defined in the DBMS_CQ_NOTIFICATIONS package, as follows:

Operation	Constant
INSERT	DBMS_CQ_NOTIFICATIONS.INSERTOP
UPDATE	DBMS_CQ_NOTIFICATIONS.UPDATEOP
DELETE	DBMS_CQ_NOTIFICATIONS.DELETEOP
ALTEROP	DBMS_CQ_NOTIFICATIONS.ALTEROP
DROPOP	DBMS_CQ_NOTIFICATIONS.DROPOP
UNKNOWNOP	DBMS_CQ_NOTIFICATIONS.UNKNOWNOP
All (default)	DBMS_CQ_NOTIFICATIONS.ALL_OPERATIONS

To specify multiple operations, use bitwise OR. For example:

```
DBMS_CQ_NOTIFICATIONS.INSERTOP + DBMS_CQ_NOTIFICATIONS.DELETEOP
```

`OPERATIONS_FILTER` has no effect if you also specify `QOS_QUERY` in the `QOSFLAGS` attribute, because `QOS_QUERY` specifies notification type QRCN.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_CQ_NOTIFICATION` package

Transaction Lag Option (OCN Notification Type Only)

The Transaction Lag option applies only when the notification type is OCN.

Note: This option is deprecated. To implement flow-of-control notifications, use "Notification Grouping Options" on page 12-13.

The Transaction Lag option specifies the number of transactions by which the client application can lag behind the database. If the number is 0, every transaction that changes a registered object results in a notification. If the number is 5, every fifth transaction that changes a registered object results in a notification. The database tracks intervening changes at object granularity and includes them in the notification, so that the client does not lose them.

A transaction lag greater than 0 is useful only if an application implements flow-of-control notifications. Ensure that the application generates notifications frequently enough to satisfy the lag, so that they are not deferred indefinitely.

If you specify `TRANSACTION_LAG`, then notifications do not include `ROWID`s, even if you also specified `QOS_ROWIDS`.

Notification Grouping Options

By default, notifications are generated immediately after the event that causes them.

Notification Grouping options, which are attributes of the `CQ_NOTIFICATION$_REG_INFO` object, are the following:

Attribute	Description
`NTFN_GROUPING_CLASS`	Specifies the class by which to group notifications. Currently, the only allowed values are `DBMS_CQ_NOTIFICATION.NTFN_GROUPING_CLASS_TIME`, which groups notifications by time, and zero, which is the default (notifications are generated immediately after the event that causes them).
`NTFN_GROUPING_VALUE`	Specifies the time interval that defines the group, in seconds. For example, if this value is 900, notifications generated in the same 15-minute interval are grouped together.
`NTFN_GROUPING_TYPE`	Specifies the type of grouping, which is either of the following: ■ `DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_SUMMARY`: All notifications in the group are summarized into a single notification. **Note:** The single notification does not include `ROWID`s, even if you specified the `ROWID` option. ■ `DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_LAST`: Only the last notification in the group is published and the earlier ones discarded.

Attribute	Description
NTFN_GROUPING_START_TIME	Specifies when to start generating notifications. If specified as NULL, it defaults to the current system-generated time.
NTFN_GROUPING_REPEAT_COUNT	Specifies how many times to repeat the notification. Set to DBMS_CQ_NOTIFICATION.NTFN_GROUPING_FOREVER to receive notifications for the life of the registration. To receive at most *n* notifications during the life of the registration, set to *n*.

Note: Notifications generated by timeouts, loss of privileges, and global events might be published before the specified grouping interval expires. If they are, any pending grouped notifications are also published before the interval expires.

Reliable Option

By default, a CQN registration is stored in shared memory. To store it in a persistent database queue instead—that is, to generate **reliable notifications**—specify QOS_RELIABLE in the QOSFLAGS attribute of the CQ_NOTIFICATION$_REG_INFO object.

The advantage of reliable notifications is that if the database fails after generating them, it can still deliver them after it restarts. In an Oracle RAC environment, a surviving database instance can deliver them.

The disadvantage of reliable notifications is that they have higher CPU and I/O costs than default notifications do.

Purge-on-Notify and Timeout Options

By default, a CQN registration survives until the application that created it explicitly unregisters it or until the database implicitly purges it (due to loss of privileges, for example).

To purge the registration after it generates its first notification, specify QOS_DEREG_NFY in the QOSFLAGS attribute of the CQ_NOTIFICATION$_REG_INFO object.

To purge the registration after *n* seconds, specify *n* in the TIMEOUT attribute of the CQ_NOTIFICATION$_REG_INFO object.

You can use the Purge-on-Notify and Timeout options together.

Prerequisites for Creating CQN Registrations

The following are prerequistes for creating CQN registrations:

- You must have the following privileges:
  - EXECUTE privilege on the DBMS_CQ_NOTIFICATION package, whose subprograms you use to create a registration
  - CHANGE NOTIFICATION system privilege
  - SELECT privileges on all objects to be registered

  Loss of privileges on an object associated with a registered query generates a notification—see "Deregistration" on page 12-6.
- You must be connected as a non-SYS user.

- You must not be in the middle of an uncommitted transaction.
- The `dml_locks init.ora` parameter must have a nonzero value (as its default value does).

  (This is also a prerequisite for receiving notifications.)

> **Note:** For QRCN support, the `COMPATIBLE` setting of the database must be at least 11.0.0.

Queries that Can Be Registered for Object Change Notification (OCN)

Most queries can be registered for OCN, including those executed as part of stored procedures and `REF` cursors.

Queries that cannot be registered for OCN are the following:

- Queries on fixed tables or fixed views
- Queries on user views
- Queries that contain database links (dblinks)
- Queries over materialized views

> **Note:** You can use synonyms in OCN registrations, but not in QRCN registrations.

Queries that Can Be Registered for Query Result Change Notification (QRCN)

Some queries can be registered for QRCN in guaranteed mode, some can be registered for QRCN only in best-effort mode, and some cannot be registered for QRCN in either mode. (For information about modes, see "Guaranteed Mode" on page 12-3 and "Best-Effort Mode" on page 12-3.)

Topics:

- Queries that Can Be Registered for QRCN in Guaranteed Mode
- Queries that Can Be Registered for QRCN Only in Best-Effort Mode
- Queries that Cannot Be Registered for QRCN in Either Mode

Queries that Can Be Registered for QRCN in Guaranteed Mode

To be registered for QRCN in guaranteed mode, a query must conform to the following rules:

- Every column that it references is either a `NUMBER` data type or a `VARCHAR2` datatype.
- Arithmetic operators in column expressions are limited to the following binary operators, and their operands are columns with numeric datatypes:
  - + (addition)
  - - (subtraction, not unary minus)
  - * (multiplication)
  - / (division)
- Comparison operators in the predicate are limited to the following:

  - < (less than)
  - <= (less than or equal to)
  - = (equal to)
  - >= (greater than or equal to)
  - > (greater than)
  - <> or `!=` (not equal to)
  - `IS NULL`
  - `IS NOT NULL`
- Boolean operators in the predicate are limited to `AND`, `OR`, and `NOT`.
- The query contains no aggregate functions (such as `SUM`, `COUNT`, `AVERAGE`, `MIN`, and `MAX`).

  For a list of built-in SQL aggregate functions, see *Oracle Database SQL Language Reference*.

Guaranteed mode supports most queries on single tables and some inner equijoins, such as:

```
SELECT SALARY FROM EMPLOYEES, DEPARTMENTS
  WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID
    AND DEPARTMENTS.LOCATION_ID = 1700;
```

Notes:

- Sometimes the query optimizer uses an execution plan that makes a query incompatible for guaranteed mode (for example, `OR`-expansion). For information about the query optimizer, see *Oracle Database Performance Tuning Guide*.
- Queries that can be registered in guaranteed mode can also be registered in best-effort mode, but results might differ, because best-effort mode can cause false positives even for queries that CQN does not simplify. For details, see "Best-Effort Mode" on page 12-3.

Queries that Can Be Registered for QRCN Only in Best-Effort Mode

A query that does any of the following can be registered for QRCN only in best-effort mode, and its simplified version will generated notifications at object granularity:

- Refers to columns that have encryption enabled
- Has more than 10 items of the same type in the `SELECT` list
- Has expressions that include any of the following:
  - String functions (such as `SUBSTR`, `LTRIM`, and `RTRIM`)
  - Arithmetic functions (such as `TRUNC`, `ABS`, and `SQRT`)

    For a list of built-in SQL functions, see *Oracle Database SQL Language Reference*.
  - Pattern-matching conditions `LIKE` and `REGEXP_LIKE`
  - `EXISTS` or `NOT EXISTS` condition

- Has disjunctions involving predicates defined on columns from different tables. For example:

  ```
  SELECT EMPLOYEE_ID, DEPARTMENT_ID
    FROM EMPLOYEES, DEPARTMENTS
      WHERE EMPLOYEES.EMPLOYEE_ID = 10
        OR DEPARTMENTS.DEPARTMENT_ID = 'IT';
  ```

- Has user rowid access. For example:

  ```
  SELECT DEPARTMENT_ID
    FROM DEPARTMENTS
      WHERE ROWID = 'AAANkdAABAAALinAAF';
  ```

- Has any join other than an inner join
- Has an execution plan that involves any of the following:
  - Bitmap join, domain, or functional indexes
  - `UNION ALL` or `CONCATENATION`

    (Either in the query itself, or as the result of an `OR`-expansion execution plan chosen by the query optimizer.)
  - `ORDER BY` or `GROUP BY`

    (Either in the query itself, or as the result of a `SORT` operation with an `ORDER BY` option in the execution plan chosen by the query optimizer.)
  - Partitioned index-organized table (IOT) with overflow segment
  - Clustered objects
  - Parallel execution

Queries that Cannot Be Registered for QRCN in Either Mode

A query that refers to any of the following cannot be registered for QRCN in either guaranteed or best-effort mode:

- Views
- Tables that are fixed, remote, or have Virtual Private Database (VPD) policies enabled
- `DUAL` (in the `SELECT` list)
- Synonyms
- Calls to user-defined PL/SQL subprograms
- Operators not listed in "Queries that Can Be Registered for QRCN in Guaranteed Mode" on page 12-15
- The aggregate function `COUNT`

  (Other aggregate functions are allowed in best-effort mode, but not in guaranteed mode.)
- Application contexts; for example:

  ```
  SELECT SALARY FROM EMPLOYEES
    WHERE USER = SYS_CONTEXT('USERENV', 'SESSION_USER');
  ```

- `SYSDATE`, `SYSTIMESTAMP`, or `CURRENT TIMESTAMP`

Also, a query that the query optimizer has rewritten using a materialized view cannot be registered for QRCN. For information about the query optimizer, see *Oracle Database Performance Tuning Guide*.

Using PL/SQL to Register Queries for CQN

To use PL/SQL to create a CQN registration, follow these steps:

1. Create a stored PL/SQL procedure to serve as the notification handler.
2. Create a `CQ_NOTIFICATION$_REG_INFO` object that specifies the name of the notification handler, the notification type, and other attributes of the registration.
3. In your client application, use the `DBMS_CQ_NOTIFICATION.NEW_REG_START` function to open a registration block.
4. Execute the queries that you want to register. (Do not execute DML or DDL operations.)
5. Close the registration block, using the `DBMS_CQ_NOTIFICATION.REG_END` function.

Topics:

- Creating a PL/SQL Notification Handler
- Creating a CQ_NOTIFICATION$_REG_INFO Object
- Identifying Individual Queries in a Notification
- Adding Queries to an Existing Registration

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about the `CQ_NOTIFICATION$_REG_INFO` object and the functions `NEW_REG_START` and `REG_END`, all of which are defined in the `DBMS_CQ_NOTIFICATION` package

Creating a PL/SQL Notification Handler

The PL/SQL stored procedure that you create to serve as the notification handler must have the following signature:

```
PROCEDURE schema_name.proc_name(ntfnds IN CQ_NOTIFICATION$_DESCRIPTOR)
```

In the preceding signature, *`schema_name`* is the name of the database schema, *`proc_name`* is the name of the stored procedure, and *`ntfnds`* is the notification descriptor.

The notification descriptor is a `CQ_NOTIFICATION$_DESCRIPTOR` object, whose attributes describe the details of the change (transaction ID, type of change, queries affected, tables modified, and so on).

The `JOBQ` process passes the notification descriptor, *`ntfnds`*, to the notification handler, *`proc_name`*, which handles the notification according to its application requirements. (This is step 6 in Figure 12–2.)

Note: The notification handler executes inside a job queue process. The `JOB_QUEUE_PROCESSES` initialization parameter specifies the maximum number of processes that can be created for the execution of jobs. You must set `JOB_QUEUE_PROCESSES` to a nonzero value to receive PL/SQL notifications.

Creating a CQ_NOTIFICATION$_REG_INFO Object

An object of type CQ_NOTIFICATION$_REG_INFO specifies the notification handler that the database executes when a registered objects changes. In SQL*Plus, you can view its type attributes by executing the following statement:

```
DESC CQ_NOTIFICATION$_REG_INFO
```

Table 12–2 describes the attributes of SYS.CQ_NOTIFICATION$_REG_INFO.

Table 12–2 Attributes of CQ_NOTIFICATION$_REG_INFO

Attribute	Description
CALLBACK	Specifies the name of the PL/SQL procedure to be executed when a notification is generated (a notification handler). You must specify the name in the form *schema_name.procedure_name*, for example, hr.dcn_callback.
QOSFLAGS	Specifies one or more quality-of-service flags, which are constants in the DBMS_CQ_NOTIFICATION package. For their names and descriptions, see Table 12–3. To specify more than one quality-of-service flag, use bitwise OR. For example: DBMS_CQ_NOTIFICATION.QOS_RELIABLE + DBMS_CQ_NOTIFICATION.QOS_ROWIDS
TIMEOUT	Specifies the timeout period for registrations. If set to a nonzero value, it specifies the time in seconds after which the database purges the registration. If 0 or NULL, then the registration persists until the client explicitly unregisters it. Can be combined with the QOSFLAGS attribute with its QOS_DEREG_NFY flag.
OPERATIONS_FILTER	Applies only to OCN (described in "Object Change Notification (OCN)" on page 12-2). Has no effect if you specify the QOS_FLAGS attribute with its QOS_QUERY flag. Filters messages based on types of SQL statement. You can specify the following constants in the DBMS_CQ_NOTIFICATION package: ■ ALL_OPERATIONS notifies on all changes ■ INSERTOP notifies on inserts ■ UPDATEOP notifies on updates ■ DELETEOP notifies on deletes ■ ALTEROP notifies on ALTER TABLE operations ■ DROPOP notifies on DROP TABLE operations ■ UNKNOWNOP notifies on unknown operations You can specify a combination of operations with a bitwise OR. For example: DBMS_CQ_NOTIFICATION.INSERTOP + DBMS_CQ_NOTIFICATION.DELETEOP.

Table 12–2 (Cont.) Attributes of CQ_NOTIFICATION$_REG_INFO

Attribute	Description
TRANSACTION_LAG	**Deprecated**. To implement flow-of-control notifications, use the NTFN_GROUPING_* attributes. Applies only to OCN (described in "Object Change Notification (OCN)" on page 12-2). Has no effect if you specify the QOS_FLAGS attribute with its QOS_QUERY flag. Specifies the number of transactions or database changes by which the client can lag behind the database. If 0, then the client receives an invalidation message as soon as it is generated. If 5, then every fifth transaction that changes a registered object results in a notification. Oracle Database tracks intervening changes at an object granularity and bundles the changes along with the notification. Thus, the client does not lose intervening changes. Most applications that must be notified of changes to an object on transaction commit without further deferral are expected to chose 0 transaction lag. A nonzero transaction lag is useful only if an application implements flow control on notifications. When using nonzero transaction lag, it is recommended that the application workload has the property that notifications are generated at a reasonable frequency. Otherwise, notifications might be deferred indefinitely till the lag is satisfied. If you specify TRANSACTION_LAG, then the ROWID level granularity is not available in the notification messages even if you specified QOS_ROWIDS during registration.
NTFN_GROUPING_CLASS	Specifies the class by which to group notifications. Currently, the only allowed value is DBMS_CQ_NOTIFICATION.NTFN_GROUPING_CLASS_TIME, which groups notifications by time.
NTFN_GROUPING_VALUE	Specifies the time interval that defines the group, in seconds. For example, if this value is 900, notifications generated in the same 15-minute interval are grouped together.
NTFN_GROUPING_TYPE	Specifies either of the following types of grouping: ▪ DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_SUMMARY: All notifications in the group are summarized into a single notification. ▪ DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_LAST: Only the last notification in the group is published and the earlier ones discarded.
NTFN_GROUPING_START_TIME	Specifies when to start generating notifications. If specified as NULL, it defaults to the current system-generated time.
NTFN_GROUPING_REPEAT_COUNT	Specifies how many times to repeat the notification. Set to DBMS_CQ_NOTIFICATION.NTFN_GROUPING_FOREVER to receive notifications for the life of the registration. To receive at most *n* notifications during the life of the registration, set to *n*.

The quality-of-service flags in Table 12–3 are constants in the DBMS_CQ_NOTIFICATION package. You can specify them with the QOS_FLAGS attribute of CQ_NOTIFICATION$_REG_INFO (see Table 12–2).

Table 12–3 Quality-of-Service Flags

Flag	Description
QOS_DEREG_NFY	Purges the registration after the first notification.
QOS_RELIABLE	Stores notifications in a persistent database queue. In an Oracle RAC environment, if a database instance fails, surviving database instances can deliver any queued notification messages. **Default:** Notifications are stored in shared memory, which performs more efficiently.
QOS_ROWIDS	Includes the ROWID of each changed row in the notification.
QOS_QUERY	Registers queries for QRCN, described in "Query Result Change Notification (QRCN)" on page 12-2. If a query cannot be registered for QRCN, an error is generated at registration time, unless you also specify QOS_BEST_EFFORT. **Default:** Queries are registered for OCN, described in "Object Change Notification (OCN)" on page 12-2
QOS_BEST_EFFORT	Used with QOS_QUERY. Registers simplified versions of queries that are too complex for query result change evaluation; in other words, registers queries for QRCN in best-effort mode, described in "Best-Effort Mode" on page 12-3. To see which queries were simplified, query the static data dictionary view DBA_CQ_NOTIFICATION_QUERIES or USER_CQ_NOTIFICATION_QUERIES. These views give the QUERYID and the text of each registered query. **Default:** Queries are registered for QRCN in guaranteed mode, described in "Guaranteed Mode" on page 12-3

Suppose that you want to invoke the procedure HR.dcn_callback whenever a registered object changes. In Example 12–4, you create a CQ_NOTIFICATION$_REG_INFO object that specifies that HR.dcn_callback receives notifications. To create the object you must have EXECUTE privileges on the DBMS_CQ_NOTIFICATION package.

Example 12–4 Creating a CQ_NOTIFICATION$_REG_INFO Object

```
DECLARE
  v_cn_addr CQ_NOTIFICATION$_REG_INFO;

BEGIN
  -- Create object:

  v_cn_addr := CQ_NOTIFICATION$_REG_INFO (
    'HR.dcn_callback',                 -- PL/SQL notification handler
    DBMS_CQ_NOTIFICATION.QOS_QUERY     -- notification type QRCN
    + DBMS_CQ_NOTIFICATION.QOS_ROWIDS, -- include rowids of changed objects
    0,                             -- registration persists until unregistered
    0,                             -- notify on all operations
    0                              -- notify immediately
    );

  -- Register queries:
  ...
END;
/
```

Identifying Individual Queries in a Notification

Any query in a registered list of queries can cause a continuous query notification. If you want to know when a certain query causes a notification, use the DBMS_CQ_NOTIFICATION.CQ_NOTIFICATION_QUERYID function in the SELECT list of that query. For example:

```
SELECT EMPLOYEE_ID, SALARY, DBMS_CQ_NOTIFICATION.CQ_NOTIFICATION_QUERYID
  FROM EMPLOYEES
    WHERE DEPARTMENT_ID = 10;
```

When that query causes a notification, the notification includes the query ID.

Adding Queries to an Existing Registration

To add queries to an existing registration, follow these steps:

1. Retrieve the registration ID of the existing registration.

   You can retrieve it from either saved SQL*Plus output or a query of *_CHANGE_NOTIFICATION_REGS.

2. Open the existing registration by calling the procedure DBMS_CQ_NOTIFICATION.ENABLE_REG with the registration ID as the parameter.

3. Execute the queries that you want to register. (Do not execute DML or DDL operations.)

4. Close the registration, using the DBMS_CQ_NOTIFICATION.REG_END function.

Example 12–5 adds a query to an existing registration whose registration ID is 21.

Example 12–5 Adding a Query to an Existing Registration

```
DECLARE
  v_cursor SYS_REFCURSOR;

BEGIN
  -- Open existing registration
  DBMS_CQ_NOTIFICATION.ENABLE_REG(21);
  OPEN v_cursor FOR
    -- Execute query to be registered
    SELECT DEPARTMENT_ID
      FROM HR.DEPARTMENTS;  -- register this query
  CLOSE v_cursor;
  -- Close registration
  DBMS_CQ_NOTIFICATION.REG_END;
END;
/
```

Best Practices for CQN Registrations

For best CQN performance, follow these registration guidelines:

- Register few queries—preferably those that reference objects that rarely change.

  Extremely volatile registered objects cause numerous notifications, whose overhead slows OLTP throughput.

- Minimize the number of duplicate registrations of any given object, in order to avoid replicating a notification message for multiple recipients.

Troubleshooting CQN Registrations

If you are unable to create a registration, or if you have created a registration but are not receiving the notifications that you expected, the problem might be one of the following:

- The `JOB_QUEUE_PROCESSES` parameter is not set to a nonzero value.

  This prevents you from receiving PL/SQL notifications through the notification handler.

- You were connected as a SYS user when you created the registrations.

  You must be connected as a non-SYS user in order to create CQN registrations.

- You changed a registered object, but did not commit the transaction.

  Notifications are generated only when the transaction commits.

- The registrations were not successfully created in the database.

  To check, query the static data dictionary view `*_CHANGE_NOTIFICATION_REGS`. For example, the following statement displays all registrations and registered objects for the current user:

  ```
  SELECT REGID, TABLE_NAME FROM USER_CHANGE_NOTIFICATION_REGS;
  ```

- Run-time errors occurred during the execution of the notification handler.

  If so, they were logged to the trace file of the `JOBQ` process that tried to execute the procedure. The name of the trace file usually has the following form:

  ```
  ORACLE_SID_jnumber_PID.trc
  ```

  For example, if the ORACLE_SID is `dbs1` and the process ID (PID) of the `JOBQ` process is 12483, the name of the trace file is usually `dbs1_j000_12483.trc`.

  Suppose that a registration is created with `'chnf_callback`' as the notification handler and registration ID 100. Suppose that `'chnf_callback`' was not defined in the database. Then the `JOBQ` trace file might contain a message of the form:

  ```
  ****************************************************************************
     Run-time error during execution of PL/SQL cbk chnf_callback for reg CHNF100.
     Error in PLSQL notification of msgid:
     Queue :
     Consumer Name :
     PLSQL function :chnf_callback
     Exception Occured, Error msg:
     ORA-00604: error occurred at recursive SQL level 2
     ORA-06550: line 1, column 7:
     PLS-00201: identifier 'CHNF_CALLBACK' must be declared
     ORA-06550: line 1, column 7:
     PL/SQL: Statement ignored
  ****************************************************************************
  ```

  If run-time errors occurred during the execution of the notification handler, create a very simple version of the notification handler to verify that you are actually receiving notifications, and then gradually add application logic.

  An example of a very simple notification handler is:

  ```
  REM Create table in HR schema to hold count of notifications received.
  CREATE TABLE nfcount(cnt NUMBER);
  INSERT INTO nfcount VALUES(0);
  COMMIT;
  ```

```
CREATE OR REPLACE PROCEDURE chnf_callback
  (ntfnds IN CQ_NOTIFICATION$_DESCRIPTOR)
IS
BEGIN
  UPDATE nfcount SET cnt = cnt+1;
  COMMIT;
END;
/
```

- There is a time lag between the commit of a transaction and the notification received by the end user.

Querying CQN Registrations

To see top-level information about all registrations, including their QOS options, query one of the static data dictionary views `*_CHANGE_NOTIFICATION_REGS`.

For example, you can obtain the registration ID for a client and the list of objects for which it receives notifications. To view registration IDs and table names for `HR`, you can do the following from SQL*Plus:

```
CONNECT HR/password;
SELECT regid, table_name FROM USER_CHANGE_NOTIFICATION_REGS;
```

To see which queries are registered for QRCN, query the static data dictionary view `USER_CQ_NOTIFICATION_QUERIES` or `DBA_CQ_NOTIFICATION_QUERIES`. These views include information about any bind values that the queries use. In these views, bind values in the original query are included in the query text as constants. The query text is equivalent, but maybe not identical, to the original query that was registered.

> **See Also:** *Oracle Database Reference* for more information about the static data dictionary views `USER_CHANGE_NOTIFICATION_REGS` and `DBA_CQ_NOTIFICATION_QUERIES`

Interpreting Notifications

When a transaction commits, Oracle Database determines whether registered objects were modified in the transaction. If so, it executes the notification handler specified in the registration.

Topics:

- Interpreting a CQ_NOTIFICATION$_DESCRIPTOR Object
- Interpreting a CQ_NOTIFICATION$_TABLE Object
- Interpreting a CQ_NOTIFICATION$_QUERY Object
- Interpreting a CQ_NOTIFICATION$_ROW Object

Interpreting a CQ_NOTIFICATION$_DESCRIPTOR Object

When a CQN registration generates a notification, Oracle Database passes a `CQ_NOTIFICATION$_DESCRIPTOR` object to the notification handler. The notification handler can find the details of the database change in the attributes of the `CQ_NOTIFICATION$_DESCRIPTOR` object.

In SQL*Plus, you can list these attributes by connecting as `SYS` and executing the following statement:

```
DESC CQ_NOTIFICATION$_DESCRIPTOR
```

Table 12–4 summarizes the attributes of CQ_NOTIFICATION$_DESCRIPTOR.

Table 12–4 Attributes of CQ_NOTIFICATION$_DESCRIPTOR

Attribute	Description
REGISTRATION_ID	The registration ID that was returned during registration.
TRANSACTION_ID	The ID for the transaction that made the change.
DBNAME	The name of the database in which the notification was generated.
EVENT_TYPE	The database event that triggers a notification. For example, the attribute can contain the following constants, which correspond to different database events: ■ EVENT_NONE ■ EVENT_STARTUP (Instance startup) ■ EVENT_SHUTDOWN (Instance shutdown - last instance shutdown in the case of Oracle RAC) ■ EVENT_SHUTDOWN_ANY (Any instance shutdown in the case of Oracle RAC) ■ EVENT_DEREG (Registration was removed) ■ EVENT_OBJCHANGE (Change to a registered table) ■ EVENT_QUERYCHANGE (Change to a registered result set)
NUMTABLES	The number of tables that were modified.
TABLE_DESC_ARRAY	This field is present only for OCN registrations. For QRCN registrations, it is NULL. If EVENT_TYPE is EVENT_OBJCHANGE]: a VARRAY of table change descriptors of type CQ_NOTIFICATION$_TABLE, each of which corresponds to a changed table. For attributes of CQ_NOTIFICATION$_TABLE, see Table 12–5. Otherwise: NULL.
QUERY_DESC_ARRAY	This field is present only for QRCN registrations. For OCN registrations, it is NULL. If EVENT_TYPE is EVENT_QUERYCHANGE]: a VARRAY of result set change descriptors of type CQ_NOTIFICATION$_QUERY, each of which corresponds to a changed result set. For attributes of CQ_NOTIFICATION$_QUERY, see Table 12–6. Otherwise: NULL.

Interpreting a CQ_NOTIFICATION$_TABLE Object

The CQ_NOTIFICATION$_DESCRIPTOR type contains an attribute called TABLE_DESC_ARRAY, which holds a VARRAY of table descriptors of type CQ_NOTIFICATION$_TABLE.

In SQL*Plus, you can list these attributes by connecting as SYS and executing the following statement:

```
DESC CQ_NOTIFICATION$_TABLE
```

Table 12–5 summarizes the attributes of CQ_NOTIFICATION$_TABLE.

Table 12–5 Attributes of CQ_NOTIFICATION$_TABLE

Attribute	Specifies . . .
`OPFLAGS`	The type of operation performed on the modified table. For example, the attribute can contain the following constants, which correspond to different database operations: ■ `ALL_ROWS` signifies that either the entire table is modified, as in a `DELETE *`, or row-level granularity of information is not requested or not available in the notification, and the recipient must assume that the entire table has changed ■ `UPDATEOP` signifies an update ■ `DELETEOP` signifies a deletion ■ `ALTEROP` signifies an `ALTER TABLE` ■ `DROPOP` signifies a `DROP TABLE` ■ `UNKNOWNOP` signifies an unknown operation
`TABLE_NAME`	The name of the modified table.
`NUMROWS`	The number of modified rows.
`ROW_DESC_ARRAY`	A `VARRAY` of row descriptors of type `CQ_NOTIFICATION$_ROW`, which is described in Table 12–7. If `ALL_ROWS` was set in the `opflags`, then the `ROW_DESC_ARRAY` member is `NULL`.

Interpreting a CQ_NOTIFICATION$_QUERY Object

The `CQ_NOTIFICATION$_DESCRIPTOR` type contains an attribute called `QUERY_DESC_ARRAY`, which holds a `VARRAY` of result set change descriptors of type `CQ_NOTIFICATION$_QUERY`.

In SQL*Plus, you can list these attributes by connecting as `SYS` and executing the following statement:

```
DESC CQ_NOTIFICATION$_QUERY
```

Table 12–6 summarizes the attributes of `CQ_NOTIFICATION$_QUERY`.

Table 12–6 Attributes of CQ_NOTIFICATION$_QUERY

Attribute	Specifies . . .
`QUERYID`	Query ID of the changed query.
`QUERYOP`	Operation that changed the query (either `EVENT_QUERYCHANGE` or `EVENT_DEREG`).
`TABLE_DESC_ARRAY`	A `VARRAY` of table change descriptors of type `CQ_NOTIFICATION$_TABLE`, each of which corresponds to a changed table that caused a change in the result set. For attributes of `CQ_NOTIFICATION$_TABLE`, see Table 12–5.

Interpreting a CQ_NOTIFICATION$_ROW Object

If the `ROWID` option was specified during registration, the `CQ_NOTIFICATION$_TABLE` type has a `ROW_DESC_ARRAY` attribute, a `VARRAY` of type `CQ_NOTIFICATION$_ROW` that contains the `ROWID`s for the changed rows. If `ALL_ROWS` was set in the `OPFLAGS` field of the `CQ_NOTIFICATION$_TABLE` object, then `ROWID` information is not available.

Table 12–7 summarizes the attributes of `CQ_NOTIFICATION$_ROW`.

Table 12–7 Attributes of CQ_NOTIFICATION$_ROW

Attribute	Specifies . . .
OPFLAGS	The type of operation performed on the modified table. See the description of OPFLAGS in Table 12–5.
ROW_ID	The ROWID of the changed row.

Deleting Registrations

To delete a registration, call the procedure DBMS_CQ_NOTIFICATION.DEREGISTER with the registration ID as the parameter. For example, the following statement deregisters the registration whose registration ID is 21:

```
DBMS_CQ_NOTIFICATION.DEREGISTER(21);
```

Only the user who created the registration or the SYS user can deregister it.

Configuring CQN: Scenario

In this scenario, you are a developer who manages a Web application that provides employee data: name, location, phone number, and so on. The application, which runs on Oracle Application Server, is heavily used and processes frequent queries of the HR.EMPLOYEES and HR.DEPARTMENTS tables in the back-end database. Because these tables change relatively infrequently, the application can improve performance by caching the query results. Caching avoids a round trip to the back-end database as well as server-side execution latency.

You can use the DBMS_CQ_NOTIFICATION package to register queries based on HR.EMPLOYEES and HR.DEPARTMENTS tables. To configure CQN, you follow these steps:

1. Create a server-side PL/SQL stored procedure to process the notifications, as instructed in "Creating a PL/SQL Notification Handler" on page 12-28.
2. Register the queries on the HR.EMPLOYEES and HR.DEPARTMENTS tables for QRCN, as instructed in "Registering the Queries" on page 12-30.

After you complete these steps, any committed change to the result of a query registered in step 2 causes the notification handler created in step 1 to notify the Web application of the change, whereupon the Web application refreshes the cache by querying the back-end database.

Topics:

- Creating a PL/SQL Notification Handler
- Registering the Queries

Creating a PL/SQL Notification Handler

You create a a server-side stored PL/SQL procedure to process notifications as follows:

1. You connect to the database as a user with DBA privileges:

   ```
   CONNECT / AS SYSDBA;
   ```

2. You grant the required privileges to HR:

   ```
   GRANT EXECUTE ON DBMS_CQ_NOTIFICATION TO HR;
   GRANT CHANGE NOTIFICATION TO HR;
   ```

3. You enable the `JOB_QUEUE_PROCESSES` parameter to receive notifications:

   ```
   ALTER SYSTEM SET "JOB_QUEUE_PROCESSES"=4;
   ```

4. You connect to the database as a non-SYS user:

   ```
   CONNECT HR/password;
   ```

5. You create database tables to hold records of notification events received:

   ```
   REM Create a table to record notification events.
   CREATE TABLE nfevents (
     regid      NUMBER,
     event_type NUMBER   );

   REM Create a table to record notification queries.
   CREATE TABLE nfqueries (
     qid NUMBER,
     qop NUMBER           );

   REM Create a table to record changes to registered tables.
   CREATE TABLE nftablechanges (
     qid             NUMBER,
     table_name      VARCHAR2(100),
     table_operation NUMBER    );

   REM Create a table to record ROWIDs of changed rows.
   CREATE TABLE nfrowchanges (
     qid        NUMBER,
     table_name VARCHAR2(100),
     row_id     VARCHAR2(2000));
   ```

6. You create the procedure `HR.chnf_callback`, as shown in Example 12–6.

Example 12–6 Creating Server-Side PL/SQL Notification Handler

```
CREATE OR REPLACE PROCEDURE chnf_callback (
  ntfnds IN CQ_NOTIFICATION$_DESCRIPTOR   )
IS
  regid           NUMBER;
  tbname          VARCHAR2(60);
  event_type      NUMBER;
  numtables       NUMBER;
  operation_type  NUMBER;
  numrows         NUMBER;
  row_id          VARCHAR2(2000);
  numqueries      NUMBER;
  qid NUMBER;
  qop NUMBER;

BEGIN
  regid := ntfnds.registration_id;
  event_type := ntfnds.event_type;
  INSERT INTO nfevents VALUES(regid, event_type);
  numqueries :=0;

  IF (event_type = DBMS_CQ_NOTIFICATION.EVENT_QUERYCHANGE) THEN
    numqueries := ntfnds.query_desc_array.count;
    FOR i IN 1..numqueries LOOP
      qid := ntfnds.query_desc_array(i).queryid;
```

```
      qop := ntfnds.query_desc_array(i).queryop;
      INSERT INTO nfqueries VALUES(qid, qop);
      numtables := 0;
      numtables := ntfnds.query_desc_array(i).table_desc_array.count;
      FOR j IN 1..numtables LOOP
        tbname :=
          ntfnds.query_desc_array(i).table_desc_array(j).table_name;
        operation_type :=
          ntfnds.query_desc_array(i).table_desc_array(j).Opflags;
        INSERT INTO nftablechanges VALUES(qid, tbname, operation_type);
        IF (bitand(operation_type, DBMS_CQ_NOTIFICATION.ALL_ROWS) = 0)
        THEN
          numrows := ntfnds.query_desc_array(i).table_desc_array(j).numrows;
        ELSE
          numrows :=0;  -- ROWID info not available
        END IF;

        /* Body of loop does not execute when numrows is zero */
        FOR k IN 1..numrows LOOP
          Row_id :=
 ntfnds.query_desc_array(i).table_desc_array(j).row_desc_array(k).row_id;
          INSERT INTO nfrowchanges VALUES(qid, tbname, Row_id);
        END LOOP;  -- loop over rows
      END LOOP;  -- loop over tables
    END LOOP;  -- loop over queries
  END IF;
  COMMIT;
END;
/
```

Registering the Queries

After creating the notification handler, you register the queries for which you want to receive notifications, specifying `HR.chnf_callback` as the notification handler, as in Example 12–7.

Example 12–7 Registering a Query

```
DECLARE
  reginfo  CQ_NOTIFICATION$_REG_INFO;
  mgr_id   NUMBER;
  dept_id  NUMBER;
  v_cursor SYS_REFCURSOR;
  regid    NUMBER;

BEGIN
  /* Register two queries for QRNC: */
  /* 1. Construct registration information.
        chnf_callback is name of notification handler.
        QOS_QUERY specifies result-set-change notifications. */

  reginfo := cq_notification$_reg_info (
    'chnf_callback',
    DBMS_CQ_NOTIFICATION.QOS_QUERY,
    0, 0, 0                            );

  /* 2. Create registration. */

  regid := DBMS_CQ_NOTIFICATION.new_reg_start(reginfo);
  OPEN v_cursor FOR
    SELECT dbms_cq_notification.CQ_NOTIFICATION_QUERYID, manager_id
      FROM HR.EMPLOYEES
        WHERE employee_id = 7902;
  CLOSE v_cursor;

  OPEN v_cursor FOR
    SELECT dbms_cq_notification.CQ_NOTIFICATION_QUERYID, department_id
      FROM HR.departments
        WHERE department_name = 'IT';
  CLOSE v_cursor;
END;
/
```

You can view the newly created registration by issuing the following query:

```
SELECT queryid, regid, TO_CHAR(querytext)
   FROM user_cq_notification_queries;
```

The result of the preceding query has the following information:

```
QUERYID REGID                               TO_CHAR(QUERYTEXT)
------- ----- ------------------------------------------------
     22    41 SELECT HR.DEPARTMENTS.DEPARTMENT_ID
                FROM HR.DEPARTMENTS
                  WHERE HR.DEPARTMENTS.DEPARTMENT_NAME  = 'IT'

     21    41 SELECT HR.EMPLOYEES.MANAGER_ID
                FROM HR.EMPLOYEES
                  WHERE HR.EMPLOYEES.EMPLOYEE_ID  = 7902
```

Execute the following transaction, which changes the result of the query with `QUERYID` 22:

```
UPDATE DEPARTMENTS SET DEPARTMENT_NAME = 'FINANCE'
  WHERE department_name = 'IT';
COMMIT;
```

The notification procedure `chnf_callback` (which you created in Example 12–6) executes.

Now query the table in which notification events are recorded:

```
SQL> SELECT * FROM nfevents;
```

The result of the preceding query has the following information:

```
REGID EVENT_TYPE
----- ----------
   61          7
```

`EVENT_TYPE` 7 corresponds to `EVENT_QUERYCHANGE` (query result change).

Query the table in which changes to registered tables are recorded:

```
SELECT * FROM nftablechanges;
```

The result of the preceding query has the following information:

```
REGID     TABLE_NAME TABLE_OPERATION
----- -------------- ---------------
   42 HR.DEPARTMENTS               4
```

TABLE_OPERATION 4 corresponds to UPDATEOP (update operation).

Query the table in which ROWIDs of changed rows are recorded:

```
SELECT * FROM nfrowchanges;
```

The result of the preceding query has the following information:

```
REGID     TABLE_NAME              ROWID
----- -------------- ------------------
   61 HR.DEPARTMENTS AAANkdAABAAALinAAF
```

TABLE_OPERATION 4 corresponds to UPDATEOP (update operation).

Part III

Advanced Topics for Application Developers

This part presents application development information that either involves sophisticated technology or is used by a small minority of developers.

Chapters:

See Also: *Oracle Database Performance Tuning Guide* for performance issues to consider when developing applications

13

Using Flashback Technology

This chapter explains how to use Flashback Technology in database applications.

Topics:

Overview of Flashback Technology

Flashback Technology is a group of Oracle Database features that that let you view past states of database objects or to return database objects to a previous state without using point-in-time media recovery.

With flashback features, you can do the following:

- Perform queries that return past data
- Perform queries that return metadata that shows a detailed history of changes to the database
- Recover tables or rows to a previous point in time
- Automatically track and archive transactional data changes
- Roll back a transaction and its dependent transactions while the database remains online

Flashback features use the Automatic Undo Management (AUM) system to obtain metadata and historical data for transactions. They rely on **undo data**, which are records of the effects of individual transactions. For example, if a user executes an

UPDATE statement to change a salary from 1000 to 1100, then Oracle Database stores the value 1000 in the undo data.

Undo data is persistent and survives a database shutdown. By using flashback features, you can employ undo data to query past data or recover from logical corruptions. Besides using it in flashback features, Oracle Database uses undo data to perform the following actions:

- Roll back active transactions
- Recover terminated transactions by using database or process recovery
- Provide read consistency for SQL queries

Topics:

- Application Development Features
- Database Administration Features

For additional general information about flashback features, see *Oracle Database Concepts*

Application Development Features

In application development, you can use the following flashback features to report historical data or undo erroneous changes. (You can also use these features interactively as a database user or administrator.)

Flashback Query

Use this feature to retrieve data for a time in the past that you specify with the AS OF clause of the SELECT statement. For more information, see "Using Flashback Query (SELECT AS OF)" on page 13-5.

Flashback Version Query

Use this feature to retrieve metadata and historical data for a specific time interval (for example, to view all the rows of a table that ever existed during a given time interval). Metadata for each row version includes start and end time, type of change operation, and identity of the transaction that created the row version. To create a Flashback Version Query, use the VERSIONS BETWEEN clause of the SELECT statement. For more information, see "Using Flashback Version Query" on page 13-7.

Flashback Transaction Query

Use this feature to retrieve metadata and historical data for a given transaction or for all transactions in a given time interval. You can also obtain the SQL code to undo the changes to particular rows affected by a transaction. To perform a Flashback Transaction Query, select from the static data dictionary view FLASHBACK_TRANSACTION_QUERY. For more information, see "Using Flashback Transaction Query" on page 13-9.

Typically, you use Flashback Transaction Query in conjunction with a Flashback Version Query that provides the transaction IDs for the rows of interest (see "Using Flashback Transaction Query with Flashback Version Query" on page 13-9).

DBMS_FLASHBACK Package

Use this feature to set the internal Oracle Database clock to a time in the past so that you can examine data that was current at that time, or to roll back a transaction and its dependent transactions while the database remains online (see "Flashback Transaction

Backout"). For more information, see "Using DBMS_FLASHBACK Package" on page 13-12.

Flashback Transaction Backout

Use Flashback Transaction Backout to roll back a transaction and its dependent transactions while the database remains online. This recovery operation uses undo data to create and execute the corresponding compensating transactions that return the affected data to its original state. (Flashback Transaction Backout is part of `DBMS_FLASHBACK` package.) For more information, see "Using DBMS_FLASHBACK Package" on page 13-12.

Flashback Data Archives

Use Flashback Data Archives to automatically track and archive both regular queries and Flashback Queries, ensuring SQL-level access to the versions of database objects without getting a snapshot-too-old error. For more information, see "Using Flashback Data Archives" on page 13-15.

Database Administration Features

The following flashback features are primarily for data recovery. Typically, you use these features only as a database administrator.

This chapter focuses on the "Application Development Features" on on page 13-2. For more information about the database administration features, see *Oracle Database Administrator's Guide* and the *Oracle Database Backup and Recovery User's Guide*.

Flashback Table

Use this feature to restore a table to its state at a previous point in time. You can restore a table while the database is on line, undoing changes to only the specified table.

Flashback Drop

Use this feature to recover a dropped table. This feature reverses the effects of a `DROP TABLE` statement.

Flashback Database

Use this feature to quickly return the database to an earlier point in time, by undoing all of the changes that have taken place since then. This is fast, because you do not have to restore database backups.

Configuring Your Database for Flashback Technology

Before you can use flashback features in your application, you or your database administrator must perform the configuration tasks described in the following topics:

- Configuring Your Database for Automatic Undo Management
- Configuring Your Database for Flashback Transaction Query
- Configuring Your Database for Flashback Transaction Backout
- Enabling Flashback Operations on Specific LOB Columns
- Granting Necessary Privileges

Configuring Your Database for Automatic Undo Management

To configure your database for Automatic Undo Management (AUM), you or your database administrator must do the following:

- Create an undo tablespace with enough space to keep the required data for flashback operations.

  The more often users update the data, the more space is required. The database administrator usually calculates the space requirement.

- Enable AUM, as explained in *Oracle Database Administrator's Guide*. Set the following database initialization parameters:

  - `UNDO_MANAGEMENT`
  - `UNDO_TABLESPACE`
  - `UNDO_RETENTION`

  For a fixed-size undo tablespace, Oracle Database automatically tunes the system to give the undo tablespace the best possible undo retention.

  For an automatically extensible undo tablespace, Oracle Database retains undo data longer than the longest query duration as well as the low threshold of undo retention specified by the `UNDO_RETENTION` parameter.

  Note: You can query `V$UNDOSTAT.TUNED_UNDORETENTION` to determine the amount of time for which undo is retained for the current undo tablespace.

  Setting `UNDO_RETENTION` does not guarantee that unexpired undo data is not discarded. If the system needs more space, Oracle Database can overwrite unexpired undo with more recently generated undo data.

- Specify the `RETENTION GUARANTEE` clause for the undo tablespace to ensure that unexpired undo data is not discarded.

  See Also: *Oracle Database Administrator's Guide* for more information about creating an undo tablespace and enabling AUM

Configuring Your Database for Flashback Transaction Query

To configure your database for the Flashback Transaction Query feature, you or your database administrator must do the following:

- Ensure that Oracle Database is running with version 10.0 compatibility.
- Enable supplemental logging:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

Configuring Your Database for Flashback Transaction Backout

To configure your database for the Flashback Transaction Backout feature, you or your database administrator must do the following:

- With the database mounted but not open, enable `ARCHIVELOG`:

```
ALTER DATABASE ARCHIVELOG;
```

- Open at least one archive log:

```
ALTER SYSTEM ARCHIVE LOG CURRENT;
```

- If not done already, enable supplemental logging:

  ```
  ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
  ```

Enabling Flashback Operations on Specific LOB Columns

To enable flashback operations on specific LOB columns of a table, use the ALTER TABLE statement with the RETENTION option.

Because undo data for LOB columns can be voluminous, you must define which LOB columns to use with flashback operations.

> **See Also:** *Oracle Database SecureFiles and Large Objects Developer's Guide* to learn about LOB storage and the RETENTION parameter

Granting Necessary Privileges

You or your database administrator must grant privileges to users, roles, or applications that need to use the following flashback features. For information about the GRANT statement, see *Oracle Database SQL Language Reference*.

For Flashback Query and Flashback Version Query

Do either of the following:

- To allow access to specific objects during queries, grant FLASHBACK and SELECT privileges on those objects.
- To allow queries on all tables, grant the FLASHBACK ANY TABLE privilege.

For Flashback Transaction Query

Grant the SELECT ANY TRANSACTION privilege.

To allow execution of undo SQL code retrieved by a Flashback Transaction Query, grant SELECT, UPDATE, DELETE, and INSERT privileges for specific tables.

For DBMS_FLASHBACK Package

To allow access to the features in the DBMS_FLASHBACK package, grant the EXECUTE privilege on DBMS_FLASHBACK.

For Flashback Data Archives

To allow a specific user to use a specific Flashback Data Archive, grant the FLASHBACK ARCHIVE object privilege on that Flashback Data Archive to that user. The user can then enable Flashback Archive on tables, using that Flashback Data Archive.

To allow execution of the following statements, grant the ARCHIVE ADMINISTER system privilege:

- CREATE FLASHBACK ARCHIVE
- ALTER FLASHBACK ARCHIVE
- DROP FLASHBACK ARCHIVE

Using Flashback Query (SELECT AS OF)

To use Flashback Query, use a SELECT statement with an AS OF clause. Flashback Query retrieves data as it existed at some time in the past. The query explicitly references a past time through a timestamp or System Change Number (SCN). It returns committed data that was current at that point in time.

Uses of Flashback Query include:

- Recovering lost data or undoing incorrect, committed changes.

  For example, if you mistakenly delete or update rows, and then commit them, you can immediately undo the mistake.
- Comparing current data with the corresponding data at some time in the past.

  For example, you can run a daily report that shows the change in data from yesterday. You can compare individual rows of table data or find intersections or unions of sets of rows.
- Checking the state of transactional data at a particular time.

  For example, you can verify the account balance of a certain day.
- Simplifying application design by removing the need to store some kinds of temporal data.

  Flashback Query lets you retrieve past data directly from the database.
- Applying packaged applications, such as report generation tools, to past data.
- Providing self-service error correction for an application, thereby enabling users to undo and correct their errors.

Topics:

- Example of Examining and Restoring Past Data
- Guidelines for Flashback Query

For more information about the `SELECT AS OF` statement, see *Oracle Database SQL Language Reference*.

Example of Examining and Restoring Past Data

Suppose that you discover at 12:30 PM that the row for employee Chung was deleted from the `employees` table, and you know that at 9:30AM the data for Chung was correctly stored in the database. You can use Flashback Query to examine the contents of the table at 9:30 AM to find out what data was lost. If appropriate, you can restore the lost data.

Example 13–1 retrieves the state of the record for `Chung` at 9:30AM, April 4, 2004:

Example 13–1 Retrieving a Lost Row with Flashback Query

```
SELECT * FROM employees AS OF TIMESTAMP
   TO_TIMESTAMP('2004-04-04 09:30:00', 'YYYY-MM-DD HH:MI:SS')
   WHERE last_name = 'Chung';
```

Example 13–2 restores Chung's information to the `employees` table:

Example 13–2 Restoring a Lost Row After Flashback Query

```
INSERT INTO employees
    (SELECT * FROM employees AS OF TIMESTAMP
     TO_TIMESTAMP('2004-04-04 09:30:00', 'YYYY-MM-DD HH:MI:SS')
     WHERE last_name = 'Chung');
```

Guidelines for Flashback Query

- You can specify or omit the AS OF clause for each table and specify different times for different tables.
- You can use the AS OF clause in queries to perform DDL operations (such as creating and truncating tables) or DML operations (such as inserting and deleting) in the same session as Flashback Query.
- To use the result of Flashback Query in a DDL or DML statement that affects the current state of the database, use an AS OF clause inside an INSERT or CREATE TABLE AS SELECT statement.
- If a possible 3-second error (maximum) is important to Flashback Query in your application, use an SCN instead of a timestamp. See "General Guidelines for Flashback Technology" on page 13-21.
- You can create a view that refers to past data by using the AS OF clause in the SELECT statement that defines the view.

  If you specify a relative time by subtracting from the current time on the database host, the past time is recalculated for each query. For example:

  ```
  CREATE VIEW hour_ago AS
    SELECT * FROM employees AS OF
      TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE);
  -- SYSTIMESTAMP refers to the time zone of the database host environment
  ```

- You can use the AS OF clause in self-joins, or in set operations such as INTERSECT and MINUS, to extract or compare data from two different times.

  You can store the results by preceding Flashback Query with a CREATE TABLE AS SELECT or INSERT INTO TABLE SELECT statement. For example, the following query reinserts into table employees the rows that existed an hour ago:

  ```
  INSERT INTO employees
    (SELECT * FROM employees AS OF
       TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE))
  -- SYSTIMESTAMP refers to the time zone of the database host environment
    MINUS SELECT * FROM employees);
  ```

Using Flashback Version Query

Use Flashback Version Query to retrieve the different versions of specific rows that existed during a given time interval. A new row version is created whenever a COMMIT statement is executed.

Specify Flashback Version Query using the VERSIONS BETWEEN clause of the SELECT statement. The syntax is:

```
VERSIONS {BETWEEN {SCN | TIMESTAMP} start AND end}
```

where *start* and *end* are expressions representing the start and end, respectively, of the time interval to be queried. The time interval includes (*start* and *end*).

Flashback Version Query returns a table with a row for each version of the row that existed at any time during the specified time interval. Each row in the table includes pseudocolumns of metadata about the row version, described in Table 13–1. This information can reveal when and how a particular change (perhaps erroneous) occurred to your database.

Table 13–1 Flashback Version Query Row Data Pseudocolumns

Pseudocolumn Name	Description
VERSIONS_STARTSCN VERSIONS_STARTTIME	Starting System Change Number (SCN) or TIMESTAMP when the row version was created. This pseudocolumn identifies the time when the data first had the values reflected in the row version. Use this pseudocolumn to identify the past target time for Flashback Table or Flashback Query. If this pseudocolumn is NULL, then the row version was created before *start*.
VERSIONS_ENDSCN VERSIONS_ENDTIME	SCN or TIMESTAMP when the row version expired. If this pseudocolumn is NULL, then either the row version was current at the time of the query or the row corresponds to a DELETE operation.
VERSIONS_XID	Identifier of the transaction that created the row version.
VERSIONS_OPERATION	Operation performed by the transaction: I for insertion, D for deletion, or U for update. The version is that of the row that was inserted, deleted, or updated; that is, the row after an INSERT operation, the row before a DELETE operation, or the row affected by an UPDATE operation. For user updates of an index key, Flashback Version Query might treat an UPDATE operation as two operations, DELETE plus INSERT, represented as two version rows with a Dfollowed by an I VERSIONS_OPERATION.

A given row version is valid starting at its time VERSIONS_START* up to, but not including, its time VERSIONS_END*. That is, it is valid for any time *t* such that VERSIONS_START* <= *t* < VERSIONS_END*. For example, the following output indicates that the salary was 10243 from September 9, 2002, included, to November 25, 2003, excluded.

```
VERSIONS_START_TIME     VERSIONS_END_TIME     SALARY
-------------------     -----------------     ------
09-SEP-2003             25-NOV-2003           10243
```

Here is a typical use of Flashback Version Query:

```
SELECT versions_startscn, versions_starttime,
       versions_endscn, versions_endtime,
       versions_xid, versions_operation,
       name, salary
  FROM employees
  VERSIONS BETWEEN TIMESTAMP
      TO_TIMESTAMP('2003-07-18 14:00:00', 'YYYY-MM-DD HH24:MI:SS')
  AND TO_TIMESTAMP('2003-07-18 17:00:00', 'YYYY-MM-DD HH24:MI:SS')
  WHERE name = 'JOE';
```

You can use VERSIONS_XID with Flashback Transaction Query to locate this transaction's metadata, including the SQL required to undo the row change and the user responsible for the change—see "Using Flashback Transaction Query" on page 13-9.

> **See Also:** *Oracle Database SQL Language Reference* for information on Flashback Version Query pseudocolumns and the syntax of the VERSIONS clause.

Using Flashback Transaction Query

Flashback Transaction Query queries the static data dictionary view FLASHBACK_TRANSACTION_QUERY. Use Flashback Transaction Query to obtain transaction information, including SQL code that you can use to undo each change that the transaction made.

See Also:

- *Oracle Database Backup and Recovery User's Guide.* for information on how a database administrator can use Flashback Table to restore an entire table, rather than individual rows
- *Oracle Database Administrator's Guide* for information on how a database administrator can use Flashback Table to restore an entire table, rather than individual rows
- *Oracle Database Reference* for more information about the static data dictionary view FLASHBACK_TRANSACTION_QUERY

The following statement queries the FLASHBACK_TRANSACTION_QUERY view for transaction information, including the transaction ID, the operation, the operation start and end SCNs, the user responsible for the operation, and the SQL code to undo the operation:

```
SELECT xid, operation, start_scn,commit_scn, logon_user, undo_sql
     FROM flashback_transaction_query
     WHERE xid = HEXTORAW('000200030000002D');
```

The following statement uses Flashback Version Query as a subquery to associate each row version with the LOGON_USER responsible for the row data change.

```
SELECT xid, logon_user FROM flashback_transaction_query
     WHERE xid IN
       (SELECT versions_xid FROM employees VERSIONS BETWEEN TIMESTAMP
        TO_TIMESTAMP('2003-07-18 14:00:00', 'YYYY-MM-DD HH24:MI:SS') AND
        TO_TIMESTAMP('2003-07-18 17:00:00', 'YYYY-MM-DD HH24:MI:SS'));
```

Using Flashback Transaction Query with Flashback Version Query

This example uses simple versions of the employees and departments tables in the sample HR schema.

In this example, a database administrator uses SQL*Plus to do the following:

```
CONNECT HR/password
CREATE TABLE emp
   (empno   NUMBER PRIMARY KEY,
    empname VARCHAR2(16)
    salary  NUMBER);
INSERT INTO emp VALUES (111, 'Mike', 555);
COMMIT;

CREATE TABLE dept
   (deptno   NUMBER,
    deptname VARCHAR2(32));
INSERT INTO dept VALUES (10, 'Accounting');
COMMIT;
```

Now `emp` and `dept` have one row each. In terms of row versions, each table has one version of one row. Suppose that an erroneous transaction deletes `empno 111` from table `emp`:

```
UPDATE emp SET salary = salary + 100 WHERE empno = 111;
INSERT INTO dept VALUES (20, 'Finance');
DELETE FROM emp WHERE empno = 111;
COMMIT;
```

Next, a transaction reinserts `empno 111` into the `emp` table with a new employee name:

```
INSERT INTO emp VALUES (111, 'Tom', 777);
UPDATE emp SET salary = salary + 100 WHERE empno = 111;
UPDATE emp SET salary = salary + 50 WHERE empno = 111;
COMMIT;
```

The database administrator detects the application error and needs to diagnose the problem. The database administrator issues the following query to retrieve versions of the rows in the `emp` table that correspond to empno `111`. The query uses Flashback Version Query pseudocolumns:

```
CONNECT dba_name/password
SELECT versions_xid XID, versions_startscn START_SCN,
  versions_endscn END_SCN, versions_operation OPERATION,
  empname, salary FROM hr.emp
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
  where empno = 111;

XID              START_SCN  END_SCN   OPERATION  EMPNAME    SALARY
---------------- ---------- --------- ---------- ---------- ----------
0004000700000058 113855               I          Tom        927
000200030000002D 113564               D          Mike       555
000200030000002E 112670     113564    I          Mike       555
3 rows selected
```

The results table rows are in descending chronological order. The third row corresponds to the version of the row in the table `emp` that was inserted in the table when the table was created. The second row corresponds to the row in `emp` that the erroneous transaction deleted. The first row corresponds to the version of the row in `emp` that was reinserted with a new employee name.

The database administrator identifies transaction `000200030000002D` as the erroneous transaction and uses Flashback Transaction Query to audit all changes made by this transaction:

```
SELECT  xid, start_scn START, commit_scn COMMIT,
        operation OP, logon_user USER,
        undo_sql FROM flashback_transaction_query
        WHERE xid = HEXTORAW('000200030000002D');

XID               START   COMMIT  OP       USER   UNDO_SQL
----------------  -----   ------  --       ----   ---------------------------
000200030000002D  195243  195244  DELETE   HR     insert into "HR"."EMP"
("EMPNO","EMPNAME","SALARY") values ('111','Mike','655');

000200030000002D  195243  195244  INSERT   HR     delete from "HR"."DEPT"
where ROWID = 'AAAKD4AABAAAJ3BAAB';

000200030000002D  195243  195244  UPDATE   HR     update "HR"."EMP"
set "SALARY" = '555' where ROWID = 'AAAKD2AABAAAJ29AAA';
```

```
000200030000002D  195243  113565  BEGIN  HR

4 rows selected
```

The (`UNDO_SQL`) column contains the SQL code that the database administrator can execute to undo the changes made by that transaction. The `USER` column (`logon_user`) shows the user responsible for the transaction.

To see the details of the erroneous transaction and all subsequent transactions, the database administrator performs the following query:

```
SELECT xid, start_scn, commit_scn, operation, table_name, table_owner
  FROM flashback_transaction_query
  WHERE table_owner = 'HR' AND
        start_timestamp >=
          TO_TIMESTAMP ('2002-04-16 11:00:00','YYYY-MM-DD HH:MI:SS');

XID               START_SCN  COMMIT_SCN  OPERATION  TABLE_NAME  TABLE_OWNER
----------------  ---------  ----------  ---------  ----------  -----------
0004000700000058  195245     195246      UPDATE     EMP         HR
0004000700000058  195245     195246      UPDATE     EMP         HR
0004000700000058  195245     195246      INSERT     EMP         HR
000200030000002D  195243     195244      DELETE     EMP         HR
000200030000002D  195243     195244      INSERT     DEPT        HR
000200030000002D  195243     195244      UPDATE     EMP         HR

6 rows selected
```

Using ORA_ROWSCN

`ORA_ROWSCN` is a pseudocolumn of any table that is not fixed or external. It represents the SCN of the most recent change to a given row; that is, the latest `COMMIT` operation for the row. For example:

```
SELECT ora_rowscn, last_name, salary
  FROM employees
  WHERE employee_id = 7788;

ORA_ROWSCN    NAME    SALARY
----------    ----    ------
    202553    Fudd      3000
```

The latest `COMMIT` operation for the row took place at approximately SCN `202553`. To convert an SCN to the corresponding `TIMESTAMP` value, use the function `SCN_TO_TIMESTAMP`.

`ORA_ROWSCN` is a conservative upper bound of the latest commit time—the actual commit SCN can be somewhat earlier. `ORA_ROWSCN` is more precise (closer to the actual commit SCN) for a row-dependent table (created using `CREATE TABLE` with the `ROWDEPENDENCIES` clause).

Uses of `ORA_ROWSCN` in application development include concurrency control and client cache invalidation.

Scenario: Concurrency Control

Your application examines a row of data and records the corresponding `ORA_ROWSCN` as `202553`. Later, the application needs to update the row, but only if the row has not changed. The operation is made conditional on the `ORA_ROWSCN` being still `202553`. An equivalent interactive statement is:

```
UPDATE employees
  SET salary = salary + 100
  WHERE employee_id = 7788
  AND ora_rowscn = 202553;

0 rows updated.
```

The conditional update fails in this case, because the ORA_ROWSCN is no longer 202553. This means that a user or another application changed the row and performed a COMMIT more recently than the recorded ORA_ROWSCN.

Your application queries again to obtain the new row data and ORA_ROWSCN. Suppose that the ORA_ROWSCN is now 415639. The application tries the conditional update again, using the new ORA_ROWSCN. This time, the update succeeds, and it is committed. An interactive equivalent is:

```
SQL> UPDATE employees SET salary = salary + 100
     WHERE empno = 7788 AND ora_rowscn = 415639;

1 row updated.

SQL> COMMIT;

Commit complete.

SQL> SELECT ora_rowscn, name, salary FROM employees WHERE empno = 7788;

ORA_ROWSCN    NAME    SALARY
----------    ----    ------
    465461    Fudd      3100
```

The SCN corresponding to the new COMMIT is 465461.

Besides using ORA_ROWSCN in an UPDATE statement WHERE clause, you can use it in a DELETE statement WHERE clause or the AS OF clause of Flashback Query.

Using DBMS_FLASHBACK Package

The DBMS_FLASHBACK package provides the same functionality as Flashback Query, but Flashback Query is sometimes more convenient.

The DBMS_FLASHBACK package acts as a time machine: you can turn back the clock, carry out normal queries as if you were at that time in the past, and then return to the present. Because you can use the DBMS_FLASHBACK package to perform queries on past data without special clauses such as AS OF or VERSIONS BETWEEN, you can reuse existing PL/SQL code to query the database at times in the past.

You must have the EXECUTE privilege on the DBMS_FLASHBACKpackage.

To use the DBMS_FLASHBACK package in your PL/SQL code:

1. Specify a past time by invoking either DBMS_FLASHBACK.ENABLE_AT_TIME or DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER.
2. Perform regular queries (that is, queries without special flashback-feature syntax such as AS OF). Do not perform DDL or DML operations.

   The database is queried at the specified past time.
3. Return to the present time by invoking DBMS_FLASHBACK.DISABLE.

You must invoke DBMS_FLASHBACK.DISABLE before invoking DBMS_FLASHBACK.ENABLE_AT_TIME or DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER again. You cannot nest enable/disable pairs.

You can use a cursor to store the results of queries. To do this, open the cursor before invoking DBMS_FLASHBACK.DISABLE. After storing the results and invoking DBMS_FLASHBACK.DISABLE, you can do the following:

- Perform INSERT or UPDATE operations to modify the current database state by using the stored results from the past.
- Compare current data with the past data. After invoking DBMS_FLASHBACK.DISABLE, open a second cursor. Fetch from the first cursor to retrieve past data; fetch from the second cursor to retrieve current data. You can store the past data in a temporary table and then use set operators such as MINUS or UNION to contrast or combine the past and current data.

You can invoke DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER at any time to get the current System Change Number (SCN). DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER always returns the current SCN regardless of previous invocations of DBMS_FLASHBACK.ENABLE.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for details of the DBMS_FLASHBACK package

Using Flashback Transaction Backout

The DBMS_FLASHBACK.TRANSACTION_BACKOUT procedure ("TRANSACTION_BACKOUT") rolls back a transaction and its dependent transactions while the database remains online. This recovery operation uses undo data to create and execute the **compensating transactions** that return the affected data to its original state.

Topics:

- TRANSACTION_BACKOUT Parameters
- TRANSACTION_BACKOUT Reports

TRANSACTION_BACKOUT Parameters

The parameters of the TRANSACTION_BACKOUT procedure are:

- Number of transactions to be backed out
- List of transactions to be backed out, identified either by name or by XID
- Time hint, if you identify transactions by name

  Specify a time that is earlier than any transaction started.
- Backout option from Table 13–2

For the syntax of the TRANSACTION_BACKOUT procedure and detailed parameter descriptions, see *Oracle Database PL/SQL Packages and Types Reference*.

Table 13–2 Flashback TRANSACTION_BACKOUT Options

Option	Description
CASCADE	Backs out specified transactions and all dependent transactions in a post-order fashion (that is, children are backed out before parents are backed out). Without CASCADE, if any dependent transaction is not specified, an error occurs.
NOCASCADE	Default. Backs out specified transactions, which are expected to have no dependent transactions. First dependent transactions causes an error and appears in *_FLASHBACK_TRANSACTION_REPORT.
NOCASCADE_FORCE	Backs out specified transactions, ignoring dependent transactions. Server executes undo SQL statements for specified transactions in reverse order of commit times. If no constraints break and you are satisfied with the result, you can commit the changes; otherwise, you can roll them back.
NONCONFLICT_ONLY	Backs out changes to nonconflicting rows of the specified transactions. Database remains consistent, but transaction atomicity is lost.

TRANSACTION_BACKOUT analyzes the transactional dependencies, performs DML operations, and generates reports. TRANSACTION_BACKOUT does not commit the DML operations that it performs as part of transaction backout, but it holds all the required locks on rows and tables in the right form, preventing other dependencies from entering the system. To make the transaction backout permanent, you must explicitly commit the transaction.

TRANSACTION_BACKOUT Reports

To see the reports that TRANSACTION_BACKOUT generates, query the static data dictionary views *_FLASHBACK_TXN_STATE and *_FLASHBACK_TXN_REPORT.

*_FLASHBACK_TXN_STATE

The static data dictionary view *_FLASHBACK_TXN_STATE shows whether a transaction is active or backed out. If a transaction appears in this view, it is backed out.

*_FLASHBACK_TXN_STATE is maintained atomically with respect to compensating transactions. If a compensating transaction is backed out, all changes that it made are also backed out, and *_FLASHBACK_TXN_STATE reflects this. For example, if compensating transaction ct backs out transactions t1 and t2, then t1 and t2 appear in *_FLASHBACK_TXN_STATE. If ct itself is later backed out, the effects of t1 and t2 are reinstated, and t1 and t2 disappear from *_FLASHBACK_TXN_STATE.

See Also: *Oracle Database Reference* for more information about *_FLASHBACK_TXN_STATE.

*_FLASHBACK_TXN_REPORT

The static data dictionary view *_FLASHBACK_TXN_REPORT provides a detailed report for each backed-out transaction.

See Also: *Oracle Database Reference* for more information about *_FLASHBACK_TXN_STATE.

Using Flashback Data Archives

A Flashback Data Archive provides the ability to track and store all transactional changes to a table over its lifetime. It is no longer necessary to build this intelligence into your application. A Flashback Data Archive is useful for compliance with record stage policies and audit reports.

A Flashback Data Archive consists of one or more tablespaces or parts thereof. You can have multiple Flashback Data Archives. You can specify a default Flashback Data Archive for the system. A Flashback Data Archive is configured with retention time. Data archived in the Flashback Data Archive is retained for the retention time.

By default, flashback archiving is off for any table. You can enable flashback archiving (and then disable it again) for a table. While flashback archiving is enabled for a table, some DDL statements are not allowed on that table.

When choosing a Flashback Data Archive for a specific table, consider the data retention requirements for the table and the retention times of the Flashback Data Archives on which you have the `FLASHBACK ARCHIVE` object privilege.

Topics:

Creating a Flashback Data Archive

Create a Flashback Data Archive with the `CREATE FLASHBACK ARCHIVE` statement, specifying the following:

- (Optional) This is the default Flashback Data Archive for the system.

  If you omit this option, you can still make this Flashback Data Archive the default later (see "Specifying the Default Flashback Data Archive" on page 13-17).
- Name of the Flashback Data Archive
- Name of the first tablespace of the Flashback Data Archive
- (Optional) Maximum amount of space that the Flashback Data Archive can use in the first tablespace

  The default is unlimited. Unless your space quota on the first tablespace is also unlimited, you must specify this value; otherwise, you will get error ORA-55621.
- Retention time (number of days that Flashback Data Archive data for the table is guaranteed to be stored)

Examples

- Create a default Flashback Data Archive named `fla1` that uses up to 10 G of tablespace `tbs1`, whose data will be retained for one year:

```
CREATE FLASHBACK ARCHIVE DEFAULT fla1 TABLESPACE tbs1
  QUOTA 10G RETENTION 1 YEAR;
```

- Create a Flashback Data Archive named `fla2` that uses tablespace `tbs2`, whose data will be retained for two years:

```
CREATE FLASHBACK ARCHIVE fla2 TABLESPACE tbs2 RETENTION 2 YEAR;
```

For more information about the `CREATE FLASHBACK ARCHIVE` statement, see *Oracle Database SQL Language Reference*.

Altering a Flashback Data Archive

With the `ALTER FLASHBACK ARCHIVE` statement, you can:

- Make a specific Flashback Data Archive the default Flashback Data Archive
- Change the retention time of a Flashback Data Archive
- Purge some or all of its data
- Add, modify, and remove tablespaces

Note: Removing all tablespaces of a Flashback Data Archive causes an error.

Examples

- Make Flashback Data Archive `fla1` the default Flashback Data Archive:

```
ALTER FLASHBACK ARCHIVE fla1 SET DEFAULT;
```

- To Flashback Data Archive `fla1`, add up to 5 G of tablespace `tbs3`:

```
ALTER FLASHBACK ARCHIVE fla1 ADD TABLESPACE tbs3 QUOTA 5G;
```

- To Flashback Data Archive `fla1`, add as much of tablespace `tbs4` as needed:

```
ALTER FLASHBACK ARCHIVE fla1 ADD TABLESPACE tbs4;
```

- Change the maximum space that Flashback Data Archive `fla1` can use in tablespace `tbs3` to 20 G:

```
ALTER FLASHBACK ARCHIVE fla1 MODIFY TABLESPACE tbs3 QUOTA 20G;
```

- Allow Flashback Data Archive `fla1` to use as much of tablespace `tbs1` as needed:

```
ALTER FLASHBACK ARCHIVE fla1 MODIFY TABLESPACE tbs1;
```

- Change the retention time for Flashback Data Archive `fla1` to two years:

```
ALTER FLASHBACK ARCHIVE fla1 MODIFY RETENTION 2 YEAR;
```

- Remove tablespace `tbs2` from Flashback Data Archive `fla1`:

```
ALTER FLASHBACK ARCHIVE fla1 REMOVE TABLESPACE tbs2;
```

  (Tablespace `tbs2` is not dropped.)

- Purge all historical data from Flashback Data Archive `fla1`:

```
ALTER FLASHBACK ARCHIVE fla1 PURGE ALL;
```

- Purge all historical data older than one day from Flashback Data Archive `fla1`:

```
ALTER FLASHBACK ARCHIVE fla1
  PURGE BEFORE TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY);
```

- Purge all historical data older than SCN 728969 from Flashback Data Archive `fla1`:

  ```
  ALTER FLASHBACK ARCHIVE fla1 PURGE BEFORE SCN 728969;
  ```

For more information about the `ALTER FLASHBACK ARCHIVE` statement, see *Oracle Database SQL Language Reference*.

Dropping a Flashback Data Archive

Drop a Flashback Data Archive with the `DROP FLASHBACK ARCHIVE` statement. Dropping a Flashback Data Archive deletes its historical data, but does not drop its tablespaces.

Example

Remove Flashback Data Archive `fla1` and all its historical data, but not its tablespaces:

```
DROP FLASHBACK ARCHIVE fla1;
```

For more information about the `DROP FLASHBACK ARCHIVE` statement, see *Oracle Database SQL Language Reference*.

Specifying the Default Flashback Data Archive

By default, the system has no default Flashback Data Archive. You can specify one in one of the following ways:

- Specify the name of an existing Flashback Data Archive in the `SET DEFAULT` clause of the `ALTER FLASHBACK ARCHIVE` statement. For example:

  ```
  ALTER FLASHBACK ARCHIVE fla1 SET DEFAULT;
  ```

  If `fla1` does not exist, an error occurs.

- Include `DEFAULT` in the `CREATE FLASHBACK ARCHIVE` statement when you create a Flashback Data Archive. For example:

  ```
  CREATE FLASHBACK ARCHIVE DEFAULT fla2 TABLESPACE tbs1
    QUOTA 10G RETENTION 1 YEAR;
  ```

The default Flashback Data Archive for the system is the default Flashback Data Archive for every user who does not have his or her own default Flashback Data Archive.

See Also:

- *Oracle Database SQL Language Reference* for more information about the `CREATE FLASHBACK ARCHIVE` statement
- *Oracle Database SQL Language Reference* for more information about the `ALTER DATABASE` statement

Enabling and Disabling Flashback Data Archive

By default, flashback archiving is disabled. At any time, you can enable flashback archiving for a table. However, if you enable flashback archiving for a table, but AUM is disabled, you will get error ORA-55614 when you try to modify the table.

To enable flashback archiving for a table, include the `FLASHBACK ARCHIVE` clause in either the `CREATE TABLE` or `ALTER TABLE` statement.

In the `FLASHBACK ARCHIVE` clause, you can specify the Flashback Data Archive where the historical data for the table will be stored. The default is the default Flashback Data Archive for the system. If you specify a nonexistent Flashback Data Archive, an error occurs.

If a table already has flashback archiving enabled, and you try to enable it again with a different Flashback Data Archive, an error occurs.

To disable flashback archiving for a table, specify `NO FLASHBACK ARCHIVE` in the `ALTER TABLE` statement. (It is unnecessary to specify `NO FLASHBACK ARCHIVE` in the `CREATE TABLE` statement, because that is the default.)

Examples

- Create table `employee` and store the historical data in the default Flashback Data Archive:

  ```
  CREATE TABLE employee (EMPNO NUMBER(4) NOT NULL, ENAME VARCHAR2(10),
    JOB VARCHAR2(9), MGR NUMBER(4)) FLASHBACK ARCHIVE;
  ```

- Create table `employee` and store the historical data in the Flashback Data Archive `fla1`:

  ```
  CREATE TABLE employee (EMPNO NUMBER(4) NOT NULL, ENAME VARCHAR2(10),
    JOB VARCHAR2(9), MGR NUMBER(4)) FLASHBACK ARCHIVE fla1;
  ```

- Enable flashback archiving for the table `employee` and store the historical data in the default Flashback Data Archive:

  ```
  ALTER TABLE employee FLASHBACK ARCHIVE;
  ```

- Enable flashback archiving for the table `employee` and store the historical data in the Flashback Data Archive `fla1`:

  ```
  ALTER TABLE employee FLASHBACK ARCHIVE fla1;
  ```

- Disable flashback archiving for the table `employee`:

  ```
  ALTER TABLE employee NO FLASHBACK ARCHIVE;
  ```

See Also:

- *Oracle Database SQL Language Reference* for more information about the `CREATE TABLE` statement
- *Oracle Database SQL Language Reference* for more information about the `ALTER TABLE` statement

DDL Statements Not Allowed on Tables Enabled for Flashback Data Archive

Using any of the following DDL statements on a table enabled for Flashback Data Archive causes error ORA-55610:

- `ALTER TABLE` statement that does any of the following:

- Drops, renames, or modifies a column
- Performs partition or subpartition operations
- Converts a LONG column to a LOB column
- Includes an UPGRADE TABLE clause, with or without an INCLUDING DATA clause

■ DROP TABLE statement

■ RENAME TABLE statement

■ TRUNCATE TABLE statement

See Also: *Oracle Database SQL Language Reference* for information about these DDL statements

Viewing Flashback Data Archive Data

Table 13–3 lists and briefly describes the static data dictionary views that you can query for information about Flashback Data Archives. For detailed information about these views, see *Oracle Database Reference*.

Table 13–3 Static Data Dictionary Views for Flashback Data Archives

View	Description
`*_FLASHBACK_ARCHIVE`	Displays information about Flashback Data Archives.
`*_FLASHBACK_ARCHIVE_TS`	Displays tablespaces of Flashback Data Archives.
`*_FLASHBACK_ARCHIVE_TABLES`	Displays information about tables that are enabled for flashback archiving.

Flashback Data Archive Scenarios

■ Scenario: Using Flashback Data Archive to Enforce Digital Shredding

■ Scenario: Using Flashback Data Archive to Access Historical Data

■ Scenario: Using Flashback Data Archive to Generate Reports

■ Scenario: Using Flashback Data Archive for Auditing

■ Scenario: Using Flashback Data Archive to Recover Data

Scenario: Using Flashback Data Archive to Enforce Digital Shredding

Your company wants to "shred" (delete) historical data changes to the Taxes table after ten years. When you create the Flashback Data Archive for Taxes, you specify a retention time of ten years:

```
CREATE FLASHBACK ARCHIVE taxes_archive TABLESPACE tbs1 RETENTION 10 YEAR;
```

When history data from transactions on Taxes exceeds the age of ten years, it is purged. (The Taxes table itself, and history data from transactions less than ten years old, are not purged.)

Scenario: Using Flashback Data Archive to Access Historical Data

You want to be able to retrieve the inventory of all items at the beginning of the year from the table inventory, and to be able to retrieve the stock price for each symbol in your portfolio at the close of business on any specified day of the year from the table stock_data.

Create a default Flashback Data Archive named `fla1` that uses up to 10 G of tablespace `tbs1`, whose data will be retained for five years:

```
CREATE FLASHBACK ARCHIVE DEFAULT fla1 TABLESPACE tbs1
  QUOTA 10G RETENTION 5 YEAR;
```

Enable Flashback Data Archive for the tables `inventory` and `stock_data`, and store the historical data in the default Flashback Data Archive:

```
ALTER TABLE inventory FLASHBACK ARCHIVE;
ALTER TABLE stock_data FLASHBACK ARCHIVE;
```

To retrieve the inventory of all items at the beginning of the year 2007, use the following query:

```
SELECT product_number, product_name, count FROM inventory AS OF
  TIMESTAMP TO_TIMESTAMP ('2007-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS');
```

To retrieve the stock price for each symbol in your portfolio at the close of business on July 23, 2007, use the following query:

```
SELECT symbol, stock_price FROM stock_data AS OF
  TIMESTAMP TO_TIMESTAMP ('2007-07-23 16:00:00', 'YYYY-MM-DD HH24:MI:SS')
  WHERE symbol IN my_portfolio;
```

Scenario: Using Flashback Data Archive to Generate Reports

You want users to be able to generate reports from the table `investments`, for data stored in the past five years.

Create a default Flashback Data Archive named `fla2` that uses up to 20 G of tablespace `tbs1`, whose data will be retained for five years:

```
CREATE FLASHBACK ARCHIVE DEFAULT fla2 TABLESPACE tbs1
  QUOTA 20G RETENTION 5 YEAR;
```

Enable Flashback Data Archive for the table `investments`, and store the historical data in the default Flashback Data Archive:

```
ALTER TABLE investments FLASHBACK ARCHIVE;
```

Lisa wants a report on the performance of her investments at the close of business on December 31, 2006. She uses the following query:

```
SELECT * FROM investments AS OF
  TIMESTAMP TO_TIMESTAMP ('2006-12-31 16:00:00', 'YYYY-MM-DD HH24:MI:SS')
  WHERE name = 'LISA';
```

Scenario: Using Flashback Data Archive for Auditing

A medical insurance company needs to audit a medical clinic. The medical insurance company has its claims in the table `Billings`, and creates a default Flashback Data Archive named `fla4` that uses up to 100 G of tablespace `tbs1`, whose data will be retained for 10 years:

```
CREATE FLASHBACK ARCHIVE DEFAULT fla4 TABLESPACE tbs1
  QUOTA 100G RETENTION 10 YEAR;
```

The company enables Flashback Data Archive for the table `Billings`, and stores the historical data in the default Flashback Data Archive:

```
ALTER TABLE Billings FLASHBACK ARCHIVE;
```

On May 1, 2007, clients were charged the wrong amounts for some diagnoses and tests. To see the records as of May 1, 2007, the company uses the following query:

```
SELECT date_billed, amount_billed, patient_name, claim_Id,
  test_costs, diagnosis FROM Billings AS OF
  TO_TIMESTAMP('2007-05-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS');
```

Scenario: Using Flashback Data Archive to Recover Data

An end user recovers from erroneous transactions that were previously committed in the database. The undo data for the erroneous transactions is no longer available, but because the required historical information is available in the Flashback Data Archive, Flashback Query works seamlessly.

Lisa manages a software development group whose product sales are doing well. On November 3, 2007, she decides to give all her level-three employees who have more than two years of experience a salary increase of 10% and a promotion to level four. Lisa asks her HR representative, Bob, to make the changes.

Using the HR web application, Bob updates the `employee` table to give Lisa's level-three employees a 10% raise and a promotion to level four. Then Bob finishes his work for the day and leaves for home, unaware that he omitted the requirement of two years of experience in his transaction. A few days later, Lisa checks to see if Bob has done the updates and finds that everyone in the group was given a raise! She calls Bob immediately and asks him to correct the error.

First, he verifies that no other transaction modified the `employee` table after his: The commit timestamp from the transaction query corresponds to Bob's transaction, two days ago.

Next, Bob uses the following statement to return the `employee` table to the state it had before his erroneous change:

```
FLASHBACK TABLE employee TO TIMESTAMP (SYSTIMESTAMP - INTERVAL '2' DAY);
```

After Bob fixes the error, at 5 PM on November 5, Lisa decides to give her star performer, Joe, an additional raise of 5%. She asks Bob to do the update. Bob finds that the record for Joe is missing.

At first, Bob thinks he cannot find Joe's record without going to the backups. Then he remembers that the `employee` table has Flashback Data Archive enabled.

Bob knows that Joe's record was present at 1 PM on November 3, 2007. He recovers Joe's record with the following Flashback Query:

```
INSERT INTO employee SELECT * FROM employee AS OF TIMESTAMP
  TO_TIMESTAMP('2007-11-2 23:00:00', 'YYYY-MM-DD HH24:MI:SS')
  WHERE name = 'JOE';
```

Bob then re-executes the two updates that Lisa requested.

General Guidelines for Flashback Technology

- Use the `DBMS_FLASHBACK.ENABLE` and `DBMS_FLASHBACK.DISABLE` procedures around SQL code that you do not control, or when you want to use the same past time for several consecutive queries.
- Use Flashback Query, Flashback Version Query, or Flashback Transaction Query for SQL code that you write, for convenience. A Flashback Query, for example, is flexible enough to do comparisons and store results in a single query.

- To obtain an SCN to use later with a flashback feature, use `DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER`.
- To compute or retrieve a past time to use in a query, use a function return value as a timestamp or SCN argument. For example, add or subtract an `INTERVAL` value to the value of the `SYSTIMESTAMP` function.
- Use Flashback Query, Flashback Version Query, and Flashback Transaction Query locally or remotely. An example of a remote Flashback Query is:

```
(SELECT * FROM employees@some_remote_host AS OF
    TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE);
```

- To ensure database consistency, always perform a `COMMIT` or `ROLLBACK` operation before querying past data.
- Remember that all flashback processing uses the current session settings, such as national language and character set, not the settings that were in effect at the time being queried.
- Remember that DDLs that alter the structure of a table (such as drop/modify column, move table, drop partition, truncate table/partition, and add constraint) invalidate any existing undo data for the table. If you try to retrieve data from a time before such a DDL executed, you will get error ORA-1466. DDL operations that alter the storage attributes of a table (such as `PCTFREE`, `INITRANS`, and `MAXTRANS`) do not invalidate undo data.
- To query past data at a precise time, use an SCN. If you use a timestamp, the actual time queried might be up to 3 seconds earlier than the time you specify. Oracle Database uses SCNs internally and maps them to timestamps at a granularity of 3 seconds.

  For example, suppose that the SCN values 1000 and 1005 are mapped to the timestamps 8:41 AM and 8:46 AM, respectively. A query for a time between 8:41:00 and 8:45:59 AM is mapped to SCN 1000; a Flashback Query for 8:46 AM is mapped to SCN 1005.

  Due to this time-to-SCN mapping, if you specify a time that is slightly after a DDL operation (such as a table creation) Oracle Database might use an SCN that is just before the DDL operation, causing error ORA-1466.
- You cannot retrieve past data from a dynamic performance (`V$`) view. A query on such a view always returns current data.
- You can perform queries on past data in static data dictionary views, such as `*_TABLES`.

Performance Guidelines for Flashback Technology

- Use the `DBMS_STATS` package to generate statistics for all tables involved in a Flashback Query. Keep the statistics current. Flashback Query uses the cost-based optimizer, which relies on these statistics.
- Minimize the amount of undo data that must be accessed. Use queries to select small sets of past data using indexes, not to scan entire tables. If you must scan a full table, add a parallel hint to the query.

  The performance cost in I/O is the cost of paging in data and undo blocks that are not already in the buffer cache. The performance cost in CPU use is the cost of applying undo information to affected data blocks. When operating on changes in the recent past, flashback features essentially CPU bound.

- For Flashback Version Query, use index structures. Oracle Database keeps undo data for index changes as well as data changes. Performance of index lookup-based Flashback Version Query is an order of magnitude faster than the full table scans that are otherwise needed.
- In a Flashback Transaction Query, the `xid` column is of the type `RAW(8)`. To take advantage of the index built on the `xid` column, use the `HEXTORAW` conversion function: `HEXTORAW(xid)`.
- A Flashback Query against a materialized view does not take advantage of query rewrite optimization.

14 Developing Applications Using Multiple Programming Languages

This chapter explains how you can develop database applications that call external procedures written in other programming languages.

Topics:

Overview of Multilanguage Programs

Oracle Database lets you work in different languages:

- PL/SQL, as described in the *Oracle Database PL/SQL Language Reference*
- C, through the Oracle Call Interface (OCI), as described in the *Oracle Call Interface Programmer's Guide*
- C or C++, through the Pro*C/C++ precompiler, as described in the *Pro*C/C++ Programmer's Guide*
- COBOL, through the Pro*COBOL precompiler, as described in the *Pro*COBOL Programmer's Guide*
- Visual Basic, through Oracle Objects for OLE (OO4O), as described in *Oracle Objects for OLE Developer's Guide*.

- Java, through the JDBC Application Programmers Interface (API). See *Oracle Database Java Developer's Guide*.

How can 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 might narrow depending on how your application needs to work with Oracle Database:

- PL/SQL is a powerful development tool, specialized for SQL transaction processing.
- Some computation-intensive tasks are executed most efficiently in a lower level language, such as C.
- The need for portability, together with the need for security, might influence you to select Java.

Most significantly, from the point of view of performance, 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 system but outside the address space of the database server. Pro*COBOL and Pro*C/C++ are precompilers, and Visual Basic accesses Oracle Database through the OCI, which is implemented in C.

Taking all these factors into account suggests that there might be a number of situations in which you might need to implement your application in more than one language. For example, the introduction of Java running within the address space of the server suggest that you might want to import existing Java applications into the database, and then leverage this technology by calling Java functions from PL/SQL and SQL.

PL/SQL external procedures enable you to write C procedure calls as PL/SQL bodies. These C procedures are callable directly from PL/SQL, and from SQL through PL/SQL procedure calls. The database provides a special-purpose interface, the call specification, that lets you call external procedures from other languages. While this service is designed for intercommunication between SQL, PL/SQL, C, and Java, it is accessible from any base language that can call these languages. For example, your procedure can be written in a language other than Java or C and still be usable by SQL or PL/SQL, as long as your procedure is callable by C. Therefore, if you have a candidate C++ procedure, 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** 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 example, when you work in PL/SQL, the language loads the library dynamically at run time, and then calls the procedure as if it were a PL/SQL procedure. These procedures participate fully in the current transaction and can call back to the database to perform SQL operations.

The procedures are loaded only when necessary, so memory is conserved. Because the decoupling of the call specification from its implementation body means that the procedures can be enhanced without affecting the calling programs.

External procedures let you:

- Isolate execution of client applications and processes from the database instance to ensure that any problems on the client side do not adversely impact the database.
- Move computation-bound programs from client to server where they execute faster (because they avoid the round-trips of network communication)
- Interface the database server with external systems and data sources
- Extend the functionality of the database server itself

Overview of Call Specification for External Procedures

You publish external procedures through **call specifications**, which provide a superset of the `AS EXTERNAL` function through the `AS LANGUAGE` clause. `AS LANGUAGE` call specifications allow the publishing of external C procedures, but also Java class methods.

Note: To support legacy applications, call specifications also enable you to publish with the `AS EXTERNAL` clause. For new application development, however, using the `AS LANGUAGE` clause is recommended.

In general, call specifications enable:

- Dispatching the appropriate C or Java target procedure
- Datatype conversions
- Parameter mode mappings
- Automatic memory allocation and cleanup
- Purity constraints to be specified, where necessary, for packaged functions called from SQL.
- Calling Java methods or C procedures from database triggers
- Location flexibility: you can put `AS LANGUAGE` call specifications in package or type specifications, or package (or type) bodies to optimize performance and hide implementation details

To use an already-existing program as an external procedure, load, publish, and then call it.

Loading External Procedures

To make your external C procedures or Java methods available to PL/SQL, you must first load them. The manner of doing this depends upon whether the procedure is written in C or Java.

Topics:

- Loading Java Class Methods
- Loading External C Procedures

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 called by the CREATE JAVA statement, the Java Virtual Machine (JVM)] library manager loads Java binaries (.class files) and resources from local BFILEs or LOB columns into RDBMS libunits.

Suppose a compiled Java class is stored in the following operating system file:

```
/home/java/bin/Agent.class
```

Creating a class libunit in schema scott from file Agent.class requires two steps: First, create a directory object on the server's file system. The name of the directory object is an alias for the directory path leading to Agent.class.

To create the directory object, you must grant user scott the CREATE ANY DIRECTORY privilege, then execute the CREATE DIRECTORY statement, as follows:

```
CONNECT SYSTEM/password
GRANT CREATE ANY DIRECTORY TO Scott IDENTIFIED BY Tiger;
CONNECT SCOTT/password
CREATE DIRECTORY Bfile_dir AS '/home/java/bin';
```

You are ready to create the class libunit, as follows:

```
CREATE JAVA CLASS USING BFILE (Bfile_dir, 'Agent.class');
```

The name of the libunit is derived from the name of the class.

Alternatively, you can use the command-line utility LoadJava. This uploads Java binaries and resources into a system-generated database table, then uses the CREATE JAVA statement to load the Java files into RDBMS libunits. You can upload Java files from file systems, Java IDEs, intranets, or the Internet.

Loading External C Procedures

Note: You can load external C procedures only on platforms that support either DLLs or dynamically loadable shared libraries (such as Solaris .so libraries).

When an application calls an external C procedure, Oracle Database or Oracle Listener starts the external procedure agent, extproc. Using the network connection established by Oracle Database or Oracle Listener, the application passes the following information to extproc:

- Name of DLL or shared library
- Name of external procedure
- Any parameters for the external procedure

Then extproc loads the DLL or the shared library, runs the external procedure, and passes any values that the external procedure returns back to the application. The application and extproc must reside on the same computer.

extproc can call procedures in any library that complies with the calling standard used. For more information about the calling standard, see "CALLING STANDARD" on page 14-10.

Note: The default configuration for external procedures no longer requires a network listener to work with Oracle Database and `extproc`. Oracle Database now spawns `extproc` directly, eliminating the risk that Oracle Listener might spawn `extproc` unexpectedly. This default configuration is recommended for maximum security.

You must change this default configuration, so that Oracle Listener spawns `extproc`, if you use any of the following:

- A multithreaded `extproc` agent
- Oracle Database in MTS mode on Windows
- An `AGENT` clause in the `LIBRARY` specification or an `AGENT IN` clause in the `PROCEDURE` specification that redirects external procedures to a different `extproc` agent

Changing the default configuration requires additional network configuration steps.

To configure your database to use external procedures that are written in C, or that can be called from C applications, you or your database administrator must take the following steps:

1. Define the C Procedures
2. Set Up the Environment
3. Identify the DLL
4. Publish the External Procedures

Define the C Procedures

Define the C procedures using one of the following prototypes:

- Kernighan & Ritchie style prototypes; for example:

```
void C_findRoot(x)
 float x;
...
```

- ISO/ANSI prototypes other than numeric datatypes that are less than full width (such as `float`, `short`, `char`); for example:

```
void C_findRoot(double x)
...
```

- Other datatypes that do not change size under default argument promotions.

  The following example changes size under default argument promotions:

```
void C_findRoot(float x)
...
```

Set Up the Environment

When you use the default configuration for external procedures, Oracle Database spawns `extproc` directly. You do not need to make configuration changes for `listener.ora` and `tnsnames.ora`. Define the environment variables to be used by external procedures in the file `extproc.ora` (located at `$ORACLE_HOME/hs/admin` on UNIX operating sytems and at `ORACLE_HOME\hs\admin` on Windows), using the following syntax:

```
SET name=value (environment_variable_name value)
```

Set the `EXTPROC_DLLS` environment variable, which restricts the DLLs that `extproc` can load, to one of the following values:

- `NULL`; for example:

  ```
  SET EXTPROC_DLLS=
  ```

  This setting, the default, allows `extproc` to load only the DLLs that are in directory `$ORACLE_HOME/bin` or `$ORACLE_HOME/lib`.

- `ONLY` followed by a colon-separated list of DLLs; for example:

  ```
  SET EXTPROC_DLLS=ONLY:DLL1:DLL2
  ```

  This setting allows `extproc` to load only the DLLs named DLL1 and DLL2. , This setting provides maximum security.

- A colon-separated list of DLLs; for example:

  ```
  SET EXTPROC_DLLS=DLL1:DLL2
  ```

  This setting allows `extproc` to load the DLLs named DLL1 and DLL2 and the DLLs that are in directory `$ORACLE_HOME/bin` or `$ORACLE_HOME/lib`.

- `ANY`; for example:

  ```
  SET EXTPROC_DLLS=ANY
  ```

  This setting allows `extproc` to load any DLL.

To change the default configuration for external procedures and have your `extproc` agent spawned by Oracle Listener, configure your database to use external procedures that are written in C, or can be called from C applications, as follows:

1. Set configuration parameters for the agent, named `extproc` by default, in the configuration files `tnsnames.ora` and `listener.ora`. This establishes the connection for the external procedure agent, `extproc`, when the database is started.
2. Start a listener process exclusively for external procedures.

   The Listener sets a few required environment variables (such as `ORACLE_HOME`, `ORACLE_SID`, and `LD_LIBRARY_PATH`) for `extproc`. It can also define specific environment variables in the `ENVS` section of its `listener.ora` entry, and these variables are passed to the agent process. Otherwise, it provides the agent with a "clean" environment. The environment variables set for the agent are independent of those set for the client and server. Therefore, external procedures, which run in the agent process, cannot read environment variables set for the client or server processes.

   Note: It is possible for you to set and read environment variables themselves by using the standard C procedures `setenv` and `getenv`, respectively. Environment variables, set this way, are specific to the agent process, which means that they can be read by all functions executed in that process, but not by any other process running on the same host.

3. Determine whether the agent for your external procedure will run in dedicated mode (the default) or multithreaded mode. In dedicated mode, one "dedicated" agent is launched for each session. In multithreaded mode, a single multithreaded `extproc` agent is launched. The multithreaded `extproc` agent handles calls using different threads for different users. In a configuration where many users can call the external procedures, using a multithreaded `extproc` agent is recommended to conserve system resources.

   If the agent will run in dedicated mode, additional configuration of the agent process is not necessary.

   If the agent will run in multithreaded mode, your database administrator must configure the database system to start the agent in multithreaded mode (as a multithreaded `extproc` agent). To do this, use the agent control utility, `agtctl`. For example, start `extproc` using the following command:

   ```
   agtctl startup extproc agent_sid
   ```

   where *`agent_sid`* is the system identifier that this `extproc` agent will service. An entry for this system identifier is typically added as an entry in the file `tnsnames.ora`. For more information about using `agtctl` for `extproc` administration, see "Administering the Multithreaded extproc Agent" on page A-4.

Note:

- If you use a multithreaded `extproc` agent, the library you call must be thread safe—to avoid errors such as a corrupt call stack.
- The database server, the agent process, and the listener process that spawns the agent process must all reside on the same host.
- By default, the agent process runs on the same database instance as your main application. In situations where reliability is critical, you might want to run the agent process for the external procedure on a separate database instance (still on the same host), so that any problems in the agent do not affect the primary database server. To do so, specify the separate database instance using a database link.

Figure A–1 on page A-3 illustrates the architecture of the multithreaded `extproc` agent.

Identify the DLL

In this context, a DLL is any dynamically loadable operating-system file that stores external procedures.

For security reasons, your DBA controls access to the DLL. Using the CREATE LIBRARY statement, the DBA creates a schema object called an alias library, which represents the DLL. Then, if you are an authorized user, the DBA grants you `EXECUTE` privileges on the **alias** library. Alternatively, the DBA might grant you `CREATE ANY LIBRARY` privileges, in which case you can create your own alias libraries using the following syntax:

```
CREATE LIBRARY [schema_name.]library_name
  {IS | AS} 'file_path'
  [AGENT 'agent_link'];
```

It is recommended that you specify the full path to the DLL, rather than just the DLL name. In the following example, you create alias library `c_utils`, which represents DLL `utils.so`:

```
CREATE LIBRARY C_utils AS '/DLLs/utils.so';
```

To allow flexibility in specifying the DLLs, you can specify the root part of the path as an environment variable using the notation `${`*`VAR_NAME`*`}`, and set up that variable in the `ENVS` section of the `listener.ora` entry.

In the following example, the agent specified by the name `agent_link` is used to run any external procedure in the library `C_Utils`. The environment variable `EP_LIB_HOME` is expanded by the agent to the appropriate path for that instance, such as `/usr/bin/dll`. Variable `EP_LIB_HOME` must be set in the file `listener.ora`, for the agent to be able to access it.

```
create or replace database link agent_link using 'agent_tns_alias';
create or replace library C_utils is
  '${EP_LIB_HOME}/utils.so' agent 'agent_link';
```

For security reasons, `extproc`, by default, loads only DLLs that are in directory `$ORACLE_HOME/bin` or `$ORACLE_HOME/lib`. Also, only local sessions—that is, Oracle Database client processes that are running on the same system—are allowed to connect to `extproc`.

To load DLLs from other directories, set the environment variable `EXTPROC_DLLS`. The value for this environment variable is a colon-separated list of DLL names qualified with the complete path. For example:

```
EXTPROC_DLLS=/private1/home/scott/dll/myDll.so:/private1/home/scott/dll/newDll.so
```

While you can set up environment variables for `extproc` through the `ENVS` parameter in the file `listener.ora`, you can also set up environment varilables in the `extproc` initialization file `extproc.ora` in directory `$ORACLE_HOME/hs/admin`. When both `extproc.ora` and `ENVS` parameter in `listener.ora` are used, the environment variables defined in `extproc.ora` take precedence. See the Oracle Net manual for more information on the `EXTPROC` feature.

Note:

- On a Windows system, specify the path using a drive letter and backslash characters (\) in the path.
- This technique does not apply to VMS systems, where the `ENVS` section of listener.ora is not supported.

Publish the External Procedures

You find or write a new external C procedure, then add it to the DLL. When the procedure is in the DLL, you publish it using the call specification mechanism described in the following section.

Publishing External Procedures

Oracle Database can only use external procedures that are published through a call specification, which maps names, parameter types, and return types for your Java class method or C external procedure to their SQL counterparts. It is written like any other PL/SQL stored procedure 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, `extproc`, for running the procedure on a different system.

You begin the declaration using the normal `CREATE OR REPLACE` syntax for a procedure, function, package specification, package body, type specification, or type body.

The call specification follows the name and parameter declarations. Its syntax is:

```
{IS | AS} LANGUAGE {C | JAVA}
```

Note: Oracle Database uses a PL/SQL variant of the ANSI SQL92 External Procedure, which replaces the ANSI clause `AS EXTERNAL` with this call specification syntax.

This is then followed by either:

```
NAME  java_string_literal_name
```

Where *java_string_literal_name* is the signature of your Java method, or by:

```
LIBRARY library_name
[NAME c_string_literal_name]
[WITH CONTEXT]
[PARAMETERS (external_parameter[, external_parameter]...)];
```

Where `library_name` is the name of your alias library, `c_string_literal_name` is the name of your external C procedure, and `external_parameter` stands for:

```
{  CONTEXT
 | SELF [{TDO | property}]
 | {parameter_name | RETURN} [property] [BY REFERENCE] [external_datatype]}
```

`property` stands for:

```
{INDICATOR [{STRUCT | TDO}] | LENGTH | DURATION | MAXLEN | CHARSETID |
CHARSETFORM}
```

Note: Unlike Java, C does not understand SQL types; therefore, the syntax is more intricate

Topics:

- AS LANGUAGE Clause for Java Class Methods
- AS LANGUAGE Clause for External C Procedures

AS LANGUAGE Clause for Java Class Methods

The `AS LANGUAGE` clause is the interface between PL/SQL and a Java class method.

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.

Topics:

- LIBRARY
- NAME
- LANGUAGE
- CALLING STANDARD
- WITH CONTEXT
- PARAMETERS
- AGENT IN

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 procedure.

Note: The terms `LANGUAGE` and `CALLING STANDARD` apply only to the superseded `AS EXTERNAL` clause.

LANGUAGE

Specifies the third-generation language in which the external procedure was written. If you omit this subclause, then the language name defaults to C.

CALLING STANDARD

Specifies the calling standard under which the external procedure was compiled. The supported calling standard is C. If you omit this subclause, then the calling standard defaults to C.

WITH CONTEXT

Specifies that a context pointer is 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 runs this procedure. This is intended for situations where the external procedure agent, `extproc`, runs using multiple agent processes, to ensure robustness if the agent process of one external procedure fails. You can pass the name of the agent process (corresponding to the name of a database link), and if `tnsnames.ora` and `listener.ora` are set up properly across both instances, the external procedure is called on the other instance. Both instances must be on the same host.

This is similar to the `AGENT` clause of the `CREATE LIBRARY` statement; specifying the value at run time through `AGENT IN` allows greater flexibility.

When the agent name is specified this way, it overrides any agent name declared in the alias library. If no agent name is specified, the default is the `extproc` agent on the same instance as the calling program.

Publishing Java Class Methods

Java classes and their methods are stored in RDBMS libunits in which you can load Java sources, binaries and resources using the `LOADJAVA` utility or the `CREATEJAVA` SQL statements. Libunits can be considered analogous to DLLs written, for example, in C—although they map one-to-one with Java classes, whereas DLLs can contain more than one procedure.

The `NAME`-clause string uniquely identifies the Java method. The PL/SQL function or procedure and Java must correspond with regard to parameters. If the Java method takes no parameters, then you must code an empty parameter list for it.

When you load Java classes into the RDBMS, they are not published to SQL automatically. This is because the methods of many Java classes are called only from other Java classes, or take parameters for which there is no appropriate SQL type.

Suppose you want to publish the following Java method named `J_calcFactorial`, which returns the factorial of its argument:

```
package myRoutines.math;
public class Factorial {
   public static int J_calcFactorial (int n) {
      if (n == 1) return 1;
      else return n * J_calcFactorial(n - 1);
   }
}
```

The following call specification publishes Java method `J_calcFactorial` as PL/SQL stored function `plsToJavaFac_func`, using SQL*Plus:

```
CREATE OR REPLACE FUNCTION Plstojavafac_func (N NUMBER) RETURN NUMBER AS
   LANGUAGE JAVA
   NAME 'myRoutines.math.Factorial.J_calcFactorial(int) return int';
```

Publishing External C Procedures

In the following example, you write a PL/SQL standalone function named `plsCallsCdivisor_func` that publishes C function `Cdivisor_func` as an external function:

```
CREATE OR REPLACE FUNCTION Plscallscdivisor_func (
/* Find greatest common divisor of x and y: */
 x     PLS_INTEGER,
 y     PLS_INTEGER)
RETURN PLS_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
- PL/SQL Package Specifications
- PL/SQL Package Bodies
- Object Type Specifications
- Object Type Bodies

Note: Oracle Database version 8.0, `AS EXTERNAL` did not allow call specifications in package or type bodies.

Note: In the following examples, the `AUTHID` and `SQL_NAME_RESOLVE` clauses might or might not be required to fully stipulate a call specification.

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about calling external procedures from PL/SQL
- *Oracle Database SQL Language Reference* for more information about the SQL `CALL` statement

Examples:

- Example: Locating a Call Specification in a PL/SQL Package
- Example: Locating a Call Specification in a PL/SQL Package
- Example: Locating a Call Specification in a PL/SQL Package
- Example: Locating a Call Specification in a PL/SQL Package
- Example: Locating a Call Specification in a PL/SQL Package

- Example: Locating a Call Specification in a PL/SQL Package
- Example: Locating a Call Specification in a PL/SQL Package

Example: Locating a Call Specification in a PL/SQL Package

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
   PROCEDURE plsToC_demoExternal_proc (x PLS_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 PLS_INTEGER, y VARCHAR2, z DATE);
END;

CREATE OR REPLACE PACKAGE BODY Demo_pack
   SQL_NAME_RESOLVE CURRENT_USER
AS
   PROCEDURE plsToC_demoExternal_proc (x PLS_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 might 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/password
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 PLS_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 PLS_INTEGER, y VARCHAR2, z DATE)
);

CREATE OR REPLACE TYPE BODY Demo_typ
AS
   MEMBER PROCEDURE plsToJ_demoExternal_proc (x PLS_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 procedure.

```
CREATE OR REPLACE PROCEDURE plsToJ_demoExternal_proc (x PLS_INTEGER, y VARCHAR2, z
DATE)
   AUTHID CURRENT_USER
AS LANGUAGE JAVA
   NAME 'pkg1.class4.methodProc1(int,java.lang.String,java.sql.Date)';
```

Example: C with Optional AUTHID

Here is an example of `AS EXTERNAL` publishing a C procedure in a standalone PL/SQL program, in which the `AUTHID` clause is optional. This maintains compatibility with the external procedures of Oracle Database version 8.0.

```
CREATE OR REPLACE PROCEDURE plsToC_demoExternal_proc (x PLS_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 PLS_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToC_demoExternal_proc (x PLS_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToC_InBody_proc (x PLS_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToJ_InBody_proc (x PLS_INTEGER, y VARCHAR2, z DATE);

   PROCEDURE plsToJ_InSpec_proc (x PLS_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 PLS_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 PLS_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 PLS_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 PLS_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 PLS_INTEGER, y VARCHAR2, z DATE)
   IS LANGUAGE JAVA
      NAME 'pkg1.class4.J_InBody_meth(int,java.lang.String,java.sql.Date)';
END;
```

Passing Parameters to External C Procedures with Call Specifications

Call specifications allows a mapping between PL/SQL and C datatypes. See "Specifying Datatypes" for datatype mappings.

Passing parameters to an external C procedure is complicated by several circumstances:

- The available set of PL/SQL datatypes does not correspond one-to-one with the set of C datatypes.
- Unlike C, PL/SQL includes the RDBMS concept of nullity. Therefore, PL/SQL parameters can be NULL, whereas C parameters cannot.
- The external procedure might need the current length or maximum length of CHAR, LONG RAW, RAW, and VARCHAR2 parameters.
- The external procedure might need character set information about CHAR, VARCHAR2, and CLOB parameters.
- PL/SQL might need the current length, maximum length, or null status of values returned by the external procedure.

In the following sections, you learn how to specify a parameter list that deals with these circumstances.

Note: The maximum number of parameters that you can pass to a C external procedure is 128. However, if you pass float or double parameters by value, then the maximum is less than 128. How much less depends on the number of such parameters and your operating system. To get a rough estimate, count each float or double passed by value as two parameters.

Topics:

- Specifying Datatypes
- External Datatype Mappings
- Passing Parameters BY VALUE or BY REFERENCE
- Declaring Formal Parameters
- Overriding Default Datatype Mapping
- Specifying Properties

Specifying Datatypes

Do not pass parameters to an external procedure directly. Instead, pass them to the PL/SQL procedure that published the external procedure, specifying PL/SQL datatypes for the parameters. PL/SQL datatypes map to default external datatypes, as shown in Table 14–1.

Note: The PL/SQL datatypes `BINARY_INTEGER` and `PLS_INTEGER` are identical. For simplicity, this document uses "`PLS_INTEGER`" to mean both `BINARY_INTEGER` and `PLS_INTEGER`.

Table 14–1 Parameter Datatype Mappings

PL/SQL Datatype	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[1] NATURALN[1] POSITIVE[1] POSITIVEN[1] SIGNTYPE[1]	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	UNSIGNED INT
FLOAT REAL	FLOAT	FLOAT
DOUBLE PRECISION	DOUBLE	DOUBLE

Table 14–1 (Cont.) Parameter Datatype Mappings

PL/SQL Datatype	Supported External Types	Default External Type
CHAR CHARACTER LONG NCHAR NVARCHAR2 ROWID VARCHAR VARCHAR2	STRING OCISTRING	STRING
LONG RAW RAW	RAW OCIRAW	RAW
BFILE BLOB CLOB NCLOB	OCILOBLOCATOR	OCILOBLOCATOR
NUMBER DEC[1] DECIMAL[1] INT[1] INTEGER[1] NUMERIC[1] SMALLINT[1]	OCINUMBER	OCINUMBER
DATE	OCIDATE	OCIDATE
TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime	OCIDateTime
INTERVAL DAY TO SECOND INTERVAL YEAR TO MONTH	OCIInterval	OCIInterval
composite object types: ADTs	dvoid	dvoid
composite object types: collections (varrays, nested tables)	OCICOLL	OCICOLL

[1] This PL/SQL type compiles only if you use AS EXTERNAL in your callspec.

External Datatype Mappings

Each external datatype maps to a C datatype, and the datatype conversions are performed implicitly. To avoid errors when declaring C prototype parameters, see Table 14–2, 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 14–2 External Datatype Mappings

External Datatype Corresponding to PL/SL Type	If Mode is IN or RETURN, Specify in C Prototype...	If Mode is IN by Reference or RETURN by Reference, Specify in C Prototype...	If Mode is IN OUT or OUT, Specify in C Prototype...
CHAR	char	char *	char *
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *

Table 14–2 (Cont.) External Datatype Mappings

External Datatype Corresponding to PL/SL Type	If Mode is IN or RETURN, Specify in C Prototype...	If Mode is IN by Reference or RETURN by Reference, Specify in C Prototype...	If Mode is IN OUT or OUT, Specify in C Prototype...
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
CHAR	char	char *	char *
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
SIZE_T	size_t	size_t *	size_t *
SB1	sb1	sb1 *	sb1 *
UB1	ub1	ub1 *	ub1 *
SB2	sb2	sb2 *	sb2 *
UB2	ub2	ub2 *	ub2 *
SB4	sb4	sb4 *	sb4 *
UB4	ub4	ub4 *	ub4 *
FLOAT	float	float *	float *
DOUBLE	double	double *	double *
STRING	char *	char *	char *
RAW	unsigned char *	unsigned char *	unsigned char *
OCILOBLOCATOR	OCILobLocator *	OCILobLocator **	OCILobLocator **
OCINUMBER	OCINumber *	OCINumber *	OCINumber *
OCISTRING	OCIString *	OCIString *	OCIString *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *
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 (nonfinal types)	dvoid*	dvoid*	dvoid**

Composite object types are not self describing. Their description is stored in a **Type Descriptor Object** (TDO). Objects and indicator structs for objects have no predefined OCI datatype, but must use the datatypes generated by Oracle Database's **Object Type Translator** (OTT). The optional TDO argument for `INDICATOR`, and for composite objects, in general, has the C datatype, OCIType *.

`OCICOLL` for `REF` and collection arguments *is* optional and only exists for the sake of completeness. You cannot map `REF`s or collections onto any other datatype and vice versa.

Passing Parameters BY VALUE or BY REFERENCE

If you specify `BY VALUE`, then scalar `IN` and `RETURN` arguments are passed by value (which is also the default). Alternatively, you might have them passed by reference by specifying `BY REFERENCE`.

By default, or if you specify `BY REFERENCE`, then scalar `IN OUT`, and `OUT` arguments are passed by reference. Specifying `BY VALUE` for `IN OUT`, and `OUT` arguments is not supported for C. The usefulness of the `BY REFERENCE/VALUE` clause is restricted to external datatypes that are, by default, passed by value. This is true for `IN`, and `RETURN` arguments of the following external types:

```
[UNSIGNED] CHAR
[UNSIGNED] SHORT
[UNSIGNED] INT
[UNSIGNED] LONG
SIZE_T
SB1
SB2
SB4
UB1
UB2
UB4
FLOAT
DOUBLE
```

All `IN` and `RETURN` arguments of external types not on this list, all `IN OUT` arguments, and all `OUT` arguments are passed by reference.

Declaring Formal Parameters

Generally, the PL/SQL procedure that publishes an external procedure declares a list of formal parameters, as the following example shows:

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

```
CREATE LIBRARY MathLib AS '/tmp/math.so';
```

```
CREATE OR REPLACE FUNCTION Interp_func (
/* Find the value of y at x degrees using Lagrange interpolation: */
   x    IN FLOAT,
   y    IN FLOAT)
RETURN FLOAT AS
   LANGUAGE C
   NAME "Interp_func"
   LIBRARY MathLib;
```

Each formal parameter declaration specifies a name, parameter mode, and PL/SQL datatype (which maps to the default external datatype). That might be all the information the external procedure needs. If not, then you can provide more information using the `PARAMETERS` clause, which lets you specify the following:

- Nondefault external datatypes
- The current or maximum length of a parameter
- `NULL`/`NOT NULL` indicators for parameters
- Character set IDs and forms
- The position of parameters in the list
- How `IN` parameters are passed (by value or by reference)

If you decide to use the `PARAMETERS` clause, keep in mind:

- For every formal parameter, there must be a corresponding parameter in the `PARAMETERS` clause.
- If you include the `WITH CONTEXT` clause, then you must specify the parameter `CONTEXT`, which shows the position of the context pointer in the parameter list.
- If the external procedure is a function, then you might specify the `RETURN` parameter, but it must be in the last position. If `RETURN` is not specified, the default external type is used.

Overriding Default Datatype Mapping

In some cases, you can use the `PARAMETERS` clause to override the default datatype mappings. For example, you can remap the PL/SQL datatype `BOOLEAN` from external datatype `INT` to external datatype `CHAR`.

Specifying Properties

You can also use the `PARAMETERS` clause to pass additional information about PL/SQL formal parameters and function results to an external procedure. Do this by specifying one or more of the following properties:

```
INDICATOR [{STRUCT | TDO}]
LENGTH
DURATION
MAXLEN
CHARSETID
CHARSETFORM
SELF
```

Table 14–3 shows the allowed and the default external datatypes, PL/SQL datatypes, and PL/SQL parameter modes allowed for a given property. Notice that `MAXLEN` (used to specify data returned from C back to PL/SQL) cannot be applied to an `IN` parameter.

Table 14–3 Properties and Datatypes

Property	Allowed External Types (C)	Default External Type (C)	Allowed PL/SQL Types	Allowed PL/SQL Modes	Default PL/SQL Passing Method

`INDICATOR`	`SHORT`	`SHORT`	`all scalars`	`IN`
				`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 PLS_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(x, x_ind, y, y_len, y_maxlen, retind)
int    x;
short  x_ind;
char  *y;
int   *y_len;
int   *y_maxlen;
short *retind;
```

A K&R prototype is needed because the indicator variable `x_ind` must be of datatype `short` and `short` must not be used in ISO/ANSI prototypes.

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.

Topics:

- INDICATOR
- LENGTH and MAXLEN
- CHARSETID and CHARSETFORM
- Repositioning Parameters
- SELF
- BY REFERENCE
- WITH CONTEXT
- Interlanguage Parameter Mode Mappings

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 must be treated accordingly.

In such cases, you can use the property `INDICATOR` to associate an indicator with a formal parameter. If the PL/SQL procedure is a function, then you can also associate an indicator with the function result, as shown earlier.

To check the value of an indicator, you can use the constants `OCI_IND_NULL` and `OCI_IND_NOTNULL`. If the indicator equals `OCI_IND_NULL`, then the associated parameter or function result is `NULL`. If the indicator equals `OCI_IND_NOTNULL`, then the parameter or function result is not `NULL`.

For `IN` parameters, which are inherently read-only, `INDICATOR` is passed by value (unless you specify `BY REFERENCE`) and is read-only (even if you specify `BY REFERENCE`). For `OUT`, `IN OUT`, and `RETURN` parameters, `INDICATOR` is passed by reference by default.

The `INDICATOR` can also have a `STRUCT` or TDO option. Because specifying `INDICATOR` as a property of an object is not supported, and because arguments of objects have complete indicator structs instead of `INDICATOR` scalars, you must specify this by using the `STRUCT` option. You must use the type descriptor object (TDO) option for composite objects and collections,

LENGTH and MAXLEN

In PL/SQL, there is no standard way to indicate the length of a `RAW` or string parameter. However, in many cases, you want to pass the length of such a parameter to and from an external procedure. Using the properties `LENGTH` and `MAXLEN`, you can specify parameters that store the current length and maximum length of a formal parameter.

Note: With a parameter of type `RAW` or `LONG RAW`, you must use the property `LENGTH`. Also, if that parameter is `IN OUT` and `NULL` or `OUT` and `NULL`, then you must set the length of the corresponding C parameter to zero.

For IN parameters, LENGTH is passed by value (unless you specify BY REFERENCE) and is read-only. For OUT, IN OUT, and RETURN parameters, LENGTH is passed by reference.

As mentioned earlier, MAXLEN does not apply to IN parameters. For OUT, IN OUT, and RETURN parameters, MAXLEN is passed by reference and is read-only.

CHARSETID and CHARSETFORM

Oracle Database provides globalization support, which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments.

By default, if the server and agent use the exact same $ORACLE_HOME value, the agent uses the same globalization support settings as the server (including any settings that were specified with ALTER SESSION statements).

If the agent is running in a separate $ORACLE_HOME (even if the same location is specified by two different aliases or symbolic links), the agent uses the same globalization support settings as the server except for the character set; the default character set for the agent is defined by the NLS_LANG and NLS_NCHAR environment settings for the agent.

The properties CHARSETID and CHARSETFORM identify the nondefault character set from which the character data being passed was formed. With CHAR, CLOB, and VARCHAR2 parameters, you can use CHARSETID and CHARSETFORM to pass the character set ID and form to the external procedure.

For IN parameters, CHARSETID and CHARSETFORM are passed by value (unless you specify BY REFERENCE) and are read-only (even if you specify BY REFERENCE). For OUT, IN OUT, and RETURN parameters, CHARSETID and CHARSETFORM are passed by reference and are read-only.

The OCI attribute names for these properties are OCI_ATTR_CHARSET_ID and OCI_ATTR_CHARSET_FORM.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information about OCI_ATTR_CHARSET_ID and OCI_ATTR_CHARSET_FORM
- *Oracle Database Globalization Support Guide* for more information about using national language data with the OCI

Repositioning Parameters

Remember, each formal parameter of the external procedure must have a corresponding parameter in the PARAMETERS clause. Their positions can differ, because PL/SQL associates them by name, not by position. However, the PARAMETERS clause and the C prototype for the external procedure must have the same number of parameters, and they must be in the same order.

SELF

SELF is the always-present argument of an object type's member 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 create a table of this object type. The user eventually wants to determine the age of each `Person` object in this table.

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

```
CONNECT SYSTEM/password
GRANT CONNECT,RESOURCE,CREATE LIBRARY TO SCOTT IDENTIFIED BY
password;
CONNECT SCOTT/password
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)
);
```

Typically, the member function is implemented in PL/SQL, but in this example it is an external procedure. The body of the member function is declared as follows:

```
CREATE OR REPLACE TYPE BODY Person1_typ AS
  MEMBER FUNCTION calcAge_func RETURN NUMBER
  AS LANGUAGE C
  NAME "age"
  LIBRARY agelib
  WITH CONTEXT
  PARAMETERS
  ( CONTEXT,
    SELF,
    SELF INDICATOR STRUCT,
    SELF TDO,
    RETURN INDICATOR
  );
END;
```

Notice that the `calcAge_func` member function does not take any arguments, but only returns a number. A member function is always called on an instance of the associated object type. The object instance itself always is an implicit argument of the member function. To refer to the implicit argument, the `SELF` keyword is used. This is incorporated into the external procedure syntax by supporting references to `SELF` in the parameters clause.

The matching table is created and populated.

```
CREATE TABLE Person_tab OF Person1_typ;

INSERT INTO Person_tab VALUES
   ('SCOTT', TO_DATE('14-MAY-85'));

INSERT INTO Person_tab VALUES
   ('TIGER', TO_DATE('22-DEC-71'));
```

Finally, we retrieve the information of interest from the table.

```
SELECT p.name, p.b_date, p.calcAge_func() FROM Person_tab p;

NAME                           B_DATE    P.CALCAGE_
------------------------------ --------- ----------
SCOTT                          14-MAY-85          0
TIGER                          22-DEC-71          0
```

The following is sample C code that implements the `external` member function and the Object-Type-Translator (OTT)-generated `struct` definitions:

```
#include <oci.h>

struct PERSON
{
    OCIString   *NAME;
    OCIDate      B_DATE;
};
typedef struct PERSON PERSON;

struct PERSON_ind
{
    OCIInd    _atomic;
    OCIInd    NAME;
    OCIInd    B_DATE;
};
typedef struct PERSON_ind PERSON_ind;

OCINumber *age (ctx, person_obj, person_obj_ind, tdo, ret_ind)
OCIExtProcContext *ctx;
PERSON         *person_obj;
PERSON_ind     *person_obj_ind;
OCIType        *tdo;
OCIInd         *ret_ind;
{
    sword      err;
    text       errbuf[512];
    OCIEnv    *envh;
    OCISvcCtx *svch;
    OCIError  *errh;
    OCINumber *age;
    int        inum = 0;
    sword      status;

    /* get OCI Environment */
    err = OCIExtProcGetEnv( ctx, &envh, &svch, &errh );

    /* initialize return age to 0 */
    age = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
    status = OCINumberFromInt(errh, &inum, sizeof(inum), OCI_NUMBER_SIGNED,
                              age);
    if (status != OCI_SUCCESS)
    {
      OCIExtProcRaiseExcp(ctx, (int)1476);
      return (age);
    }

    /* return NULL if the person object is null or the birthdate is null */
    if ( person_obj_ind->_atomic == OCI_IND_NULL ||
         person_obj_ind->B_DATE  == OCI_IND_NULL )
    {
        *ret_ind = OCI_IND_NULL;
        return (age);
    }

    /* The actual implementation to calculate the age is left to the reader,
       but an easy way of doing this is a callback of the form:
            select trunc(months_between(sysdate, person_obj->b_date) / 12)
            from DUAL;
    */
    *ret_ind = OCI_IND_NOTNULL;
    return (age);
}
```

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 DOUBLE PRECISION)
AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_findRoot"
   PARAMETERS (
      x BY REFERENCE);
```

In this case, the C prototype is:

```
void C_findRoot(double *x);
```

The default (used when there is no `PARAMETERS` clause) is:

```
void C_findRoot(double 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 is 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 PLS_INTEGER AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_getNum"
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      x BY REFERENCE,
      RETURN INDICATOR);
```

The C prototype is:

```
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.

Interlanguage Parameter Mode Mappings

PL/SQL supports the `IN`, `IN OUT`, and `OUT` parameter modes, as well as the `RETURN` clause for procedures returning values.

Executing External Procedures with CALL Statements

Now that you have published your Java class method or external C procedure, you are ready to call it.

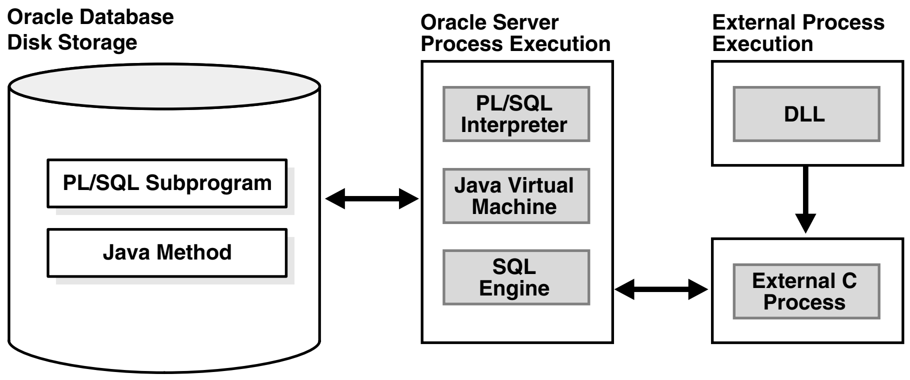
Do not call an external procedure directly. Instead, use the `CALL` statement to call the PL/SQL procedure that published the external procedure. See "CALL Statement Syntax" on page 14-29.

Such calls, which you code in the same manner as a call to a regular PL/SQL procedure, can appear in the following:

- Anonymous blocks
- Standalone and packaged procedures
- Methods of an object type
- Database triggers
- SQL statements (calls to packaged functions only).

Any PL/SQL block or procedure 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 14–1 shows how Oracle Database and external procedures interact.

Figure 14–1 Oracle Database and External Procedures



Topics:

- Preconditions for External Procedures
- CALL Statement Syntax

- Calling Java Class Methods
- Calling External C Procedures

Preconditions for External Procedures

Before calling external procedures, consider the privileges, permissions, and synonyms that exist in the execution environment.

Topics:

- Privileges of External Procedures
- Managing Permissions
- Creating Synonyms for External Procedures

Privileges of External Procedures

When external procedures are called through CALL specifications, they execute with definer's privileges, rather than invoker privileges.

A program executing with invoker privileges is not bound to a particular schema. It executes at the calling site and accesses database items (such as tables and views) with the caller's visibility and permissions. However, a program executing with definer's privileges is bound to the schema in which it is defined. It executes at the defining site, in the definer's schema, and accesses database items with the definer's visibility and permissions.

Managing Permissions

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

```
CONNECT SYSTEM/password
GRANT CREATE ANY DIRECTORY TO SCOTT;
CONNECT SCOTT/password
CREATE OR REPLACE DIRECTORY bfile_dir AS '/tmp';
CREATE OR REPLACE JAVA RESOURCE NAMED "appImages" USING BFILE
(bfile_dir,'bfile_audio');
```

To call external procedures, a user must have the EXECUTE privilege on the call specification and on any resources used by the procedure.

In SQL*Plus, you can use the GRANT and REVOKE data control statements to manage permissions. For example:

```
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Public;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Public;
GRANT EXECUTE ON JAVA RESOURCE "appImages" TO Public;
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Scott;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Scott;
```

See Also:

- *Oracle Database SQL Language Reference* for more information about the GRANTstatement
- *Oracle Database SQL Language Reference* for more information about the REVOKE statement

Creating Synonyms for External Procedures

For convenience, you or your DBA can create synonyms for external procedures using the `CREATE PUBLIC SYNONYM` statement. In the following example, your DBA creates a public synonym, which is accessible to all users. If `PUBLIC` is not specified, then the synonym is private and accessible only within its schema.

```
CREATE PUBLIC SYNONYM Rfac FOR Scott.RecursiveFactorial;
```

CALL Statement Syntax

Call the external procedure through 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 equivalent to executing a procedure `myproc` using a SQL statement of the form "`SELECT myproc(...) FROM DUAL`," except that the overhead associated with performing the `SELECT` is not incurred.

For example, here is an anonymous PL/SQL block that uses dynamic SQL to call `plsToC_demoExternal_proc`, which we published. PL/SQL passes three parameters to the external C procedure `C_demoExternal_proc`.

```
DECLARE
   xx NUMBER(4);
   yy VARCHAR2(10);
   zz DATE;
 BEGIN
 EXECUTE IMMEDIATE
 'CALL plsToC_demoExternal_proc(:xxx, :yyy, :zzz)' USING xx,yy,zz;
 END;
```

The semantics of the `CALL` statement is identical to the that of an equivalent `BEGIN-END` block.

Note: `CALL` is the only SQL statement that cannot be put, by itself, in a PL/SQL `BEGIN-END` block. It can be part of an `EXECUTE IMMEDIATE` statement within a `BEGIN-END` block.

Calling Java Class Methods

To call the `J_calcFactorial` class method published earlier:

1. Declare and initialize two SQL*Plus host variables:

   ```
   VARIABLE x NUMBER
   VARIABLE y NUMBER
   EXECUTE :x := 5;
   ```

2. Call `J_calcFactorial`:

   ```
   CALL
   J_calcFactorial
   (:x) INTO :y;
   PRINT y
   ```

Result:

```
Y
```

```
------
   120
```

Calling External C Procedures

To call an external C procedure, PL/SQL must find the path of the appropriate DLL. The PL/SQL engine retrieves the path from the data dictionary, based on the library alias from the `AS LANGUAGE` clause of the procedure declaration.

Next, PL/SQL alerts a Listener process which, in turn, spawns a session-specific agent. By default, this agent is named `extproc`, although you can specify other names in the `listener.ora` file. The Listener hands over the connection to the agent, and PL/SQL passes to the agent the name of the DLL, the name of the external procedure, and any parameters.

Then, the agent loads the DLL and runs the external procedure. Also, the agent handles service calls (such as raising an exception) and callbacks to Oracle Database. Finally, the agent passes to PL/SQL any values returned by the external procedure.

Note: Although some DLL caching takes place, there is no guarantee that your DLL will remain in the cache; therefore, do not store global variables in your DLL.

After the external procedure completes, the agent remains active throughout your Oracle Database session; when you log off, the agent is killed. Consequently, you incur the cost of launching the agent only once, no matter how many calls you make. Still, call an external procedure only when the computational benefits outweigh the cost.

Here, we call PL/SQL function `plsCallsCdivisor_func`, which we published previously, from an anonymous block. PL/SQL passes the two integer parameters to external function `Cdivisor_func`, which returns their greatest common divisor.

```
DECLARE
   g    PLS_INTEGER;
   a    PLS_INTEGER;
   b    PLS_INTEGER;
CALL plsCallsCdivisor_func(a, b);
IF g IN (2,4,8) THEN ...
```

Handling Errors and Exceptions in Multilanguage Programs

The PL/SQL compiler raises compile time errors if an `AS EXTERNAL` call specification is found in a `TYPE` or `PACKAGE` specification.

C programs can raise exceptions through the `OCIExtproc` functions.

Using Service Routines with External C Procedures

When called from an external procedure, a **service routine** can raise exceptions, allocate memory, and call 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 Linux and UNIX.

Service procedures:

- OCIExtProcAllocCallMemory
- OCIExtProcRaiseExcp
- OCIExtProcRaiseExcpWithMsg

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: Do not have the external procedure 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 might need to set up data structures similar to the following for certain examples to work:

```
CONNECT SYSTEM/password
DROP USER y CASCADE;
GRANT CONNECT,RESOURCE,CREATE LIBRARY TO y IDENTIFIED BY
password;
CONNECT y/password
CREATE LIBRARY stringlib AS
'/private/varora/ilmswork/Cexamples/john2.so';
```

```
CREATE OR REPLACE FUNCTION plsToC_concat_func (
   str1 IN VARCHAR2,
   str2 IN VARCHAR2)
RETURN VARCHAR2 AS LANGUAGE C
NAME "concat"
LIBRARY stringlib
WITH CONTEXT
PARAMETERS (
CONTEXT,
str1   STRING,
str1   INDICATOR short,
str2   STRING,
str2   INDICATOR short,
RETURN INDICATOR short,
RETURN LENGTH short,
RETURN STRING);
```

When called, `C_concat` concatenates two strings, then returns the result:

```
select plsToC_concat_func('hello ', 'world') from DUAL;
PLSTOC_CONCAT_FUNC('HELLO','WORLD')
-----------------------------------------------------------------------------
hello world
```

If either string is `NULL`, the result is also `NULL`. As the following example shows, `C_concat` uses `OCIExtProcAllocCallMemory` to allocate memory for the result string:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>
#include <ociextp.h>

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char   *str1;
short  str1_i;
char   *str2;
short  str2_i;
short  *ret_i;
short  *ret_l;
{
  char *tmp;
  short len;
  /* Check for null inputs. */
  if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
  {
      *ret_i = (short)OCI_IND_NULL;
      /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
      tmp = OCIExtProcAllocCallMemory(ctx, 1);
      tmp[0] = '\0';
      return(tmp);
  }
  /* Allocate memory for result string, including NULL terminator. */
  len = strlen(str1) + strlen(str2);
  tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

  strcpy(tmp, str1);
  strcat(tmp, str2);

  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}

#ifdef LATER
static void checkerr (/*_ OCIError *errhp, sword status _*/);

void checkerr(errhp, status)
```

```
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                    errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char   *str1;
short  str1_i;
char   *str2;
short  str2_i;
short  *ret_i;
short  *ret_l;
{
  char *tmp;
  short len;
  /* Check for null inputs. */
  if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
  {
     *ret_i = (short)OCI_IND_NULL;
     /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
     tmp = OCIExtProcAllocCallMemory(ctx, 1);
     tmp[0] = '\0';
     return(tmp);
  }
  /* Allocate memory for result string, including NULL terminator. */
  len = strlen(str1) + strlen(str2);
  tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

  strcpy(tmp, str1);
  strcat(tmp, str2);

  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}

/*======================================================================*/
int main(char *argv, int argc)
{
  OCIExtProcContext *ctx;
  char              *str1;
  short             str1_i;
  char              *str2;
  short             str2_i;
  short             *ret_i;
  short             *ret_l;
  /* OCI Handles */
  OCIEnv        *envhp;
  OCIServer     *srvhp;
  OCISvcCtx     *svchp;
  OCIError      *errhp;
  OCISession    *authp;
  OCIStmt       *stmthp;
  OCILobLocator *clob, *blob;
  OCILobLocator *Lob_loc;

  /* Initialize and Logon */
  (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                       (dvoid * (*)(dvoid *, size_t)) 0,
                       (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                       (void (*)(dvoid *, dvoid *)) 0 );

  (void) OCIEnvInit( (OCIEnv **) &envhp,
                    OCI_DEFAULT, (size_t) 0,
                    (dvoid **) 0 );

  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                   (size_t) 0, (dvoid **) 0);

  /* Server contexts */
  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                   (size_t) 0, (dvoid **) 0);

  /* Service context */
  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                   (size_t) 0, (dvoid **) 0);

  /* Attach to Oracle Database */
  (void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);

  /* Set attribute server context in the service context */
  (void) OCIAttrSet ((dvoid *) svchp, OCI_HTYPE_SVCCTX,
                     (dvoid *)srvhp, (ub4) 0,
                    OCI_ATTR_SERVER, (OCIError *) errhp);
  (void) OCIHandleAlloc((dvoid *) envhp,
                        (dvoid **)&authp, (ub4) OCI_HTYPE_SESSION,
                        (size_t) 0, (dvoid **) 0);

  (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) "samp", (ub4)4,
                (ub4) OCI_ATTR_USERNAME, errhp);

  (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) "samp", (ub4) 4,
                (ub4) OCI_ATTR_PASSWORD, errhp);

  /* Begin a User Session */
  checkerr(errhp, OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,
                          (ub4) OCI_DEFAULT));

  (void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                  (dvoid *) authp, (ub4) 0,
                  (ub4) OCI_ATTR_SESSION, errhp);

  /* -----------------------User Logged In------------------------------*/
  printf ("user logged in \n");

  /* allocate a statement handle */
  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
           OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  checkerr(errhp, OCIDescriptorAlloc((dvoid *)envhp, (dvoid **) &Lob_loc,
                                   (ub4) OCI_DTYPE_LOB,
                                   (size_t) 0, (dvoid **) 0));

  /* ------- subroutine called  here-----------------------*/
  printf ("calling concat...\n");
  concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l);

  return 0;
}

#endif
```

OCIExtProcRaiseExcp

This service routine raises a predefined exception, which must have a valid Oracle Database error number in the range 1..32,767. After doing any necessary cleanup, your external procedure must return immediately. (No values are assigned to `OUT` or `IN OUT` parameters.) The C prototype for this function follows:

```
int OCIExtProcRaiseExcp(
   OCIExtProcContext *with_context,
   size_t errnum);
```

The parameters `with_context` and `error_number` are the context pointer and Oracle Database error number. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

In SQL*Plus, suppose you publish external procedure `plsTo_divide_proc`, as follows:

```
CREATE OR REPLACE PROCEDURE plsTo_divide_proc (
   dividend IN PLS_INTEGER,
   divisor  IN PLS_INTEGER,
   result   OUT FLOAT)
AS LANGUAGE C
   NAME "C_divide"
   LIBRARY MathLib
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      dividend INT,
      divisor  INT,
      result   FLOAT);
```

When called, `C_divide` finds the quotient of two numbers. As the following example shows, if the divisor is zero, `C_divide` uses `OCIExtProcRaiseExcp` to raise the predefined exception `ZERO_DIVIDE`:

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int    dividend;
int    divisor;
float  *result;
{
  /* Check for zero divisor. */
  if (divisor == (int)0)
  {
    /* Raise exception ZERO_DIVIDE, which is Oracle error 1476. */
    if (OCIExtProcRaiseExcp(ctx, (int)1476) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /* Incorrect parameters were passed. */
      assert(0);
    }
  }
  *result = (float)dividend / (float)divisor;
}
```

OCIExtProcRaiseExcpWithMsg

This service routine raises a user-defined exception and returns a user-defined error message. The C prototype for this function follows:

```
int OCIExtProcRaiseExcpWithMsg(
   OCIExtProcContext *with_context,
   size_t  error_number,
   text   *error_message,
   size_t  len);
```

The parameters `with_context`, `error_number`, and `error_message` are the context pointer, Oracle Database error number, and error message text. The parameter `len` stores the length of the error message. If the message is a null-terminated string, then `len` is zero. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

In the previous example, we published external procedure `plsTo_divide_proc`. In the following example, you use a different implementation. With this version, if the divisor is zero, then `C_divide` uses `OCIExtProcRaiseExcpWithMsg` to raise a user-defined exception:

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int    dividend;
int    divisor;
float  *result;
  /* Check for zero divisor. */
  if (divisor == (int)0)
  {
    /* Raise a user-defined exception, which is Oracle error 20100,
       and return a null-terminated error message. */
    if (OCIExtProcRaiseExcpWithMsg(ctx, (int)20100,
          "divisor is zero", 0) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /*  Incorrect parameters were passed. */
      assert(0);
    }
  }
  *result = dividend / divisor;

}
```

Doing Callbacks with External C Procedures

To enable callbacks, use the function OCIExtProcGetEnv.

Topics:

- OCIExtProcGetEnv
- Object Support for OCI Callbacks
- Restrictions on Callbacks
- Debugging External Procedures
- Example: Calling an External Procedure
- Global Variables in External C Procedures
- Static Variables in External C Procedures
- Restrictions on External C Procedures

OCIExtProcGetEnv

This service routine enables OCI callbacks to the database during an external procedure call. The environment handles obtained by using this function reuse the existing connection to go back to the database. If you need to establish a new connection to the database, you cannot use these handles; instead, you must create your own.

The C prototype for this function follows:

```
sword OCIExtProcGetEnv ( OCIExtProcContext *with_context,
   OCIEnv envh,
   OCISvcCtx svch,
   OCIError errh )
```

The parameter `with_context` is the context pointer, and the parameters `envh`, `svch`, and `errh` are the OCI environment, service, and error handles, respectively. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

Both external C procedures and Java class methods can call-back to the database to do SQL operations. For a working example, see "Example: Calling an External Procedure" on page 14-41.

Note: Callbacks are not necessarily a same-session phenomenon; you might execute an SQL statement in a different session through `OCIlogon`.

An external C procedure executing on Oracle Database can call a service routine to obtain OCI environment and service handles. With the OCI, you can use callbacks to execute SQL statements and PL/SQL subprograms, fetch data, and manipulate LOBs. Callbacks and external procedures operate in the same user session and transaction context, and so have the same user privileges.

In SQL*Plus, suppose you run the following script:

```
CREATE TABLE Emp_tab (empno NUMBER(10))

CREATE PROCEDURE plsToC_insertIntoEmpTab_proc (
   empno PLS_INTEGER)
AS LANGUAGE C
   NAME "C_insertEmpTab"
   LIBRARY insert_lib
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      empno LONG);
```

Later, you might call service routine `OCIExtProcGetEnv` from external procedure `plsToC_insertIntoEmpTab_proc`, as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <oratypes.h>
#include <oci.h>   /* includes ociextp.h */
...
void C_insertIntoEmpTab (ctx, empno)
OCIExtProcContext *ctx;
long empno;
{
  OCIEnv    *envhp;
  OCISvcCtx *svchp;
  OCIError  *errhp;
  int        err;
  ...
  err = OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp);
  ...
}
```

If you do not use callbacks, you do not need to include `oci.h`; instead, just include `ociextp.h`.

Object Support for OCI Callbacks

To execute object-related callbacks from your external procedures, the OCI environment in the `extproc` agent is fully initialized in object mode. You retrieve handles to this environment with the `OCIExtProcGetEnv` procedure.

The object run-time environment lets you use static, as well as dynamic, object support provided by OCI. To utilize static support, use the OTT to generate C structs for the appropriate object types, and then use conventional C code to access the object attributes.

For those objects whose types are unknown at external procedure creation time, an alternative, dynamic, way of accessing objects is first to call `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 call `OCIExtProc`. service routines. After every external procedure call, the callback mechanism is cleaned up and all OCI handles are freed.

Restrictions on Callbacks

With callbacks, the following SQL statements and OCI subprograms are not supported:

- Transaction control statements such as `COMMIT`
- Data definition statements such as `CREATE`
- The following object-oriented OCI subprograms:

```
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 subprograms such as `OCIGetPieceInfo`
- The following OCI subprograms:

```
OCIEnvInit
OCIInitialize
OCIPasswordChange
OCIServerAttach
OCIServerDetach
OCISessionBegin
OCISessionEnd
OCISvcCtxToLda
OCITransCommit
OCITransDetach
OCITransRollback
OCITransStart
```

Also, with OCI subprogram `OCIHandleAlloc`, the following handle types are not supported:

```
OCI_HTYPE_SERVER
OCI_HTYPE_SESSION
OCI_HTYPE_SVCCTX
OCI_HTYPE_TRANS
```

Debugging External Procedures

Usually, when an external procedure fails, its prototype is faulty. In other words, the prototype does not match the one generated internally by PL/SQL. This can happen if you specify an incompatible C datatype. For example, to pass an `OUT` parameter of type `REAL`, you must specify `float *`. Specifying `float`, `double *`, or any other C datatype results in a mismatch.

In such cases, you might get:

```
lost RPC connection to external routine agent
```

This error, which means that `extproc` terminated abnormally because the external procedure caused a core dump. To avoid errors when declaring C prototype parameters, see the preceding tables.

To help you debug external procedures, PL/SQL provides the utility package `DEBUG_EXTPROC`. To install the package, run the script `dbgextp.sql`, which you can find in the PL/SQL demo directory. (For the location of the directory, see your Oracle Database Installation or User's Guide.)

To use the package, follow the instructions in `dbgextp.sql`. Your Oracle Database account must have `EXECUTE` privileges on the package and `CREATE LIBRARY` privileges.

Note: `DEBUG_EXTPROC` works only on platforms with debuggers that can attach to a running process.

Example: Calling an External Procedure

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.

Global Variables in External C Procedures

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 nonthreaded configuration of the agent process, only one function is active at a time. In the case of the multithreaded `extproc` agent, multiple functions can be active at the same time, and two or more functions might try to access the global variable concurrently, with unsuccessful results.

- DLL caching

  Global variables are also used to store data that is intended to persist beyond the lifetime of a function. For example, suppose that functions `func1` and `func2` try to pass data to each other. Because of the DLL caching feature, it is possible that after `func1` completes, the DLL will be unloaded, causing all global variables to lose their values. When `func2` executes, the DLL is reloaded, and all global variables are initialized to 0, which is inconsistent with their values at the completion of `func1`.

Static Variables in External C Procedures

There are two types of static variables: external and internal. An external static variable is a special case of a global variable, so its usage is discouraged. Internal static variables are local to a particular function, but remain in existence rather than coming and going each time the function is activated. Therefore, they provide private, permanent storage within a single function. These variables are used to pass on data to subsequent calls to the same function. But, because of the DLL caching feature mentioned previously, the DLL might be unloaded and reloaded between calls, which means that the internal static variable loses its value.

> **See Also:** Template `makefile` in the RDBMS subdirectory `/public` for help creating a dynamic link library

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, ensure that the datatypes of parameters in the

PARAMETERS clause are compatible with those in the C prototype, because no implicit conversions are done.

- With a parameter of type RAW or LONG RAW, you must use the property LENGTH. Also, if that parameter is IN OUT or OUT and null, then you must set the length of the corresponding C parameter to zero.

Restrictions on External C Procedures

The following restrictions apply to external procedures:

- This feature is available only on platforms that support DLLs.
- Only C procedures and procedures callable from C code are supported.
- You cannot pass PL/SQL cursor variables or records to an external procedure. For records, use instances of object types instead.
- In the LIBRARY subclause, you cannot use a database link to specify a remote library.
- The maximum number of parameters that you can pass to a external procedure is 128. However, if you pass float or double parameters by value, then the maximum is less than 128. How much less depends on the number of such parameters and your operating system. To get a rough estimate, count each float or double passed by value as two parameters.

15

Developing Applications with Oracle XA

This chapter explains how to use the Oracle XA library. Typically, you use this library in applications that work with transaction monitors. The XA features are most useful in applications in which transactions interact with more than one database.

Topics:

- X/Open Distributed Transaction Processing (DTP)
- Oracle XA Library Subroutines
- Developing and Installing XA Applications
- Troubleshooting XA Applications
- Oracle XA Issues and Restrictions

See Also:

- *X/Open CAE Specification - Distributed Transaction Processing: The XA Specification*, X/Open Document Number XO/CAE/91/300, for an overview of XA, including basic architecture. Access at `http://www.opengroup.org/pubs/catalog/c193.htm`.
- *Oracle Call Interface Programmer's Guide* for background and reference information about the Oracle XA library.
- The Oracle Database platform-specific documentation for information on library linking filenames.
- README for changes, bugs, and restrictions in the Oracle XA library for your platform.

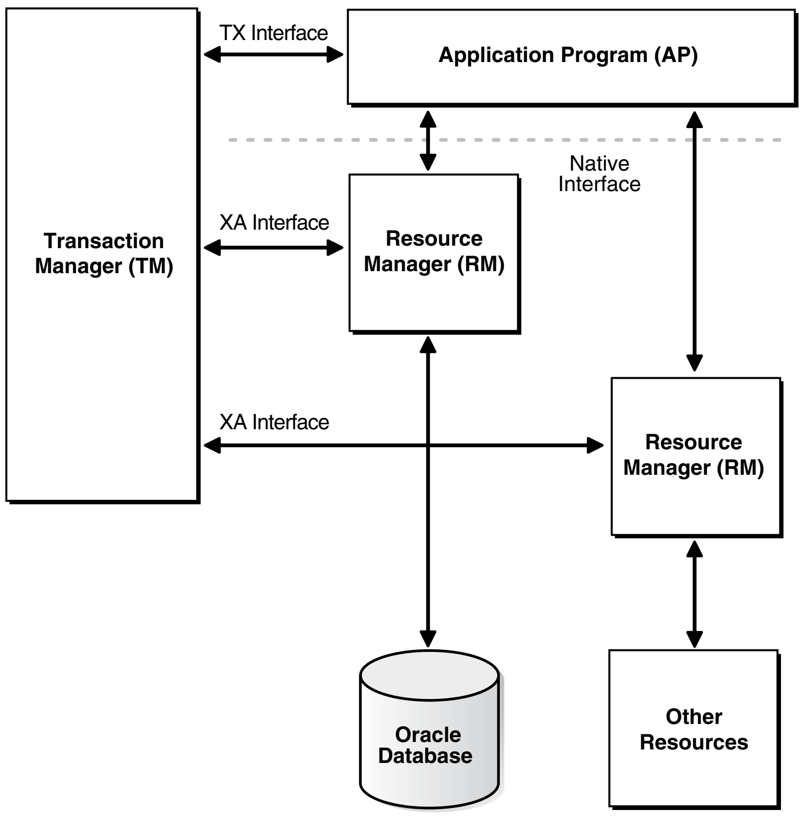
X/Open Distributed Transaction Processing (DTP)

The X/Open Distributed Transaction Processing (DTP) architecture defines a standard architecture or interface that enables multiple application programs (APs) to share resources provided by multiple, and possibly different, resource managers (RMs). It coordinates the work between APs and RMs into global transactions.

The Oracle XA library conforms to the X/Open software architecture's XA interface specification. The Oracle XA library is an external interface that enables a client-side transaction manager (TM) that is not an Oracle client-side TM to coordinate global transactions, thereby allowing inclusion of database RMs that are not Oracle Database RMs in distributed transactions. For example, a client application can manage an Oracle Database transaction and a transaction in an NTFS file system as a single, global transaction.

Figure 15–1 illustrates a possible X/Open DTP model.

Figure 15–1 Possible DTP Model



Topics:

- DTP Terminology
- Required Public Information

DTP Terminology

- Resource Manager (RM)
- Distributed Transaction
- Branch
- Transaction Manager (TM)
- Transaction Processing Monitor (TPM)
- Two-Phase Commit Protocol
- Application Program (AP)
- TX Interface
- Tight and Loose Coupling
- Dynamic and Static Registration

Resource Manager (RM)

A resource manager controls a shared, recoverable resource that can be returned to a consistent state after a failure. Examples are relational databases, transactional queues, and transactional file systems. Oracle Database is an RM and uses its online redo log and undo segments to return to a consistent state after a failure.

Distributed Transaction

A distributed transaction, also called a **global transaction**, is a client transaction that involves updates to multiple distributed resources and requires "all-or-none" semantics across distributed RMs.

Branch

A branch is a unit of work contained within one RM. Multiple branches make up one global transaction. In the case of Oracle Database, each branch maps to a local transaction inside the database server.

Transaction Manager (TM)

A transaction manager provides an API for specifying the boundaries of the transaction and manages commit and recovery. The TM implements a two-phase commit engine to provide "all-or-none" semantics across distributed RMs.

An **external TM** is a middle-tier component that resides outside Oracle Database. Normally, the database is its own internal TM. Using a standards-based TM enables Oracle Database to cooperate with other heterogeneous RMs in a single transaction.

Transaction Processing Monitor (TPM)

A TM is usually provided by a transaction processing monitor (TPM) vendor. A TPM coordinates the flow of transaction requests between the client processes that issue requests and the back-end servers that process them. Basically, a TPM coordinates transactions that require the services of several different types of back-end processes, such as application servers and RMs distributed over a network.

The TPM synchronizes any commits or rollbacks required to complete a distributed transaction. The TM portion of the TPM is responsible for controlling when distributed commits and rollbacks take place. Thus, if a distributed application program takes 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 perform this task.

Because the TM controls distributed commits or rollbacks, it must communicate directly with Oracle Database (or any other RM) through the XA interface. It uses Oracle XA library subroutines, which are described in "Oracle XA Library Subroutines" on page 15-5, to tell Oracle Database how to process the transaction, based on its knowledge of all RMs in the transaction.

Two-Phase Commit Protocol

The Oracle XA library interface follows the two-phase commit protocol. The sequence of events is as follows:

1. In the prepare phase, the TM asks each RM to guarantee that it can 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 might roll back any work, reply negatively to the TM, and forget about the transaction. The protocol allows the application, or any RM, to roll back the transaction unilaterally until the prepare phase completes.
2. In phase two, the TM records the commit decision and issues a commit or rollback to all RMs participating in the transaction. TM can issue a commit for an RM only if all RMs have replied affirmatively to phase one.

Application Program (AP)

An application program 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 its native interface, for example, SQL.

TX Interface

An application program starts and completes all transaction control operations through the TM through an interface called **TX**. The AP does not directly use the XA interface. APs are not aware of branches that fork in the middle-tier: application threads do not explicitly join, leave, suspend, and resume branch work, instead the TM portion of the transaction processing monitor manages the branches of a global transaction for APs. Ultimately, APs call the TM to commit all-or-none.

Note: The naming conventions for the TX interface and associated subroutines are vendor-specific. For example, the `tx_open` call might be referred to as `tp_open` on your system. In some cases, the calls might be implicit, for example, at the entry to a transactional RPC. See the documentation supplied with the transaction processing monitor for details.

Tight and Loose Coupling

Application threads are **tightly coupled** if the RM considers them as a single entity for all isolation semantic purposes. Tightly coupled branches must see changes in each other. Furthermore, an external client must either see all changes of a tightly coupled set or none of the changes. If application threads are not tightly coupled, then they are **loosely coupled**.

Dynamic and Static Registration

Oracle Database supports both dynamic and static registration. In **dynamic registration**, the RM executes an application callback before starting any work. In **static registration**, you must call `xa_start` for each RM before starting any work, even if some RMs are not involved.

Required Public Information

As a resource manager, Oracle Database must publish the information described in Table 15–1.

Table 15–1 Required XA Features Published by Oracle Database

XA Feature	Oracle Database Details
`xa_switch_t` structures	The Oracle Database `xa_switch_t` structure name is `xaosw` for static registration and `xaoswd` for dynamic registration. These structures contain entry points and other information for the resource manager.
`xa_switch_t` resource manager	The Oracle Database resource manager name within the `xa_switch_t` structure is `Oracle_XA`.
Close string	The close string used by `xa_close` is ignored and can 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 15-8.

Table 15–1 (Cont.) Required XA Features Published by Oracle Database

XA Feature	Oracle Database Details
Libraries	Libraries needed to link applications using Oracle XA have platform-specific names. The procedure is similar to linking an ordinary precompiler or OCI program except that you might have to link any TPM-specific libraries. If you are not using `sqllib`, then link with `$ORACLE_HOME/rdbms/lib/xaonsl.o` or `$ORACLE_HOME/rdbms/lib32/xaonsl.o` (for 32 bit application on 64 bit platforms).
Requirements	None. The functionality to support XA is part of both Standard Edition and Enterprise Edition.

Oracle XA Library Subroutines

The Oracle XA library subroutines enable a TM to tell Oracle Database how to process transactions. Generally, the TM must open the resource by using `xa_open`. Typically, the opening of the resource results from the AP's call to `tx_open`. Some TMs might 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. The close might occur when the AP calls `tx_close` or when the application terminates.

The TM instructs the RMs to perform several other tasks, which include the following:

- Starting a new transaction and associating it with an ID
- Rolling back a transaction
- Preparing and committing a transaction

Topics:

- Oracle XA Library Subroutines
- Oracle XA Interface Extensions

Oracle XA Library Subroutines

XA Library subroutines are described in Table 15–2.

Table 15–2 XA Library Subroutines

XA Subroutine	Description
`xa_open`	Connects to the RM.
`xa_close`	Disconnects from the RM.
`xa_start`	Starts a new transaction and associates it with the given transaction ID (XID), or associates the process with an existing transaction.
`xa_end`	Disassociates the process from the given XID.
`xa_rollback`	Rolls back the transaction associated with the given XID.
`xa_prepare`	Prepares the transaction associated with the given XID. This is the first phase of the two-phase commit protocol.
`xa_commit`	Commits the transaction associated with the given XID. This is the second phase of the two-phase commit protocol.

Table 15–2 (Cont.) XA Library Subroutines

XA Subroutine	Description
xa_recover	Retrieves a list of prepared, heuristically committed, or heuristically rolled back transactions.
xa_forget	Forgets the heuristically completed transaction associated with the given XID.

In general, the AP does not need to worry about the subroutines in Table 15–2 except to understand the role played by the xa_open string.

Oracle XA Interface Extensions

Oracle Database's XA interface includes some additional functions, which are described in Table 15–3.

Table 15–3 Additional Functions in the XA Interface for Oracle Database

Function	Description
`OCISvcCtx *xaoSvcCtx(text *dbname)`	Returns the OCI service handle for a given XA connection. The dbname parameter must be the same as the DB 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) by using OCISvcCtxToLda [Version 8 OCI]. Client applications must remember to convert the Version 7 LDA to a service handle by using OCILdaToSvcCtx after completing the OCI calls.
`OCIEnv *xaoEnv(text *dbname)`	Returns the OCI environment handle for a given XA connection. The dbname parameter must be the same as the DB parameter passed in the xa_open string.
`int xaosterr(OCISvcCtx *SvcCtx,sb4 error)`	Converts an Oracle Database error code to an XA error code (only applicable to dynamic registration). The first parameter is the service handle used to execute the work in the database. The second parameter is the error code that was returned from Oracle Database. Use this function to determine if the error returned from an OCI statement was caused because the xa_start failed. The function returns XA_OK if the error was not generated by the XA module or a valid XA error if the error was generated by the XA module.

Developing and Installing XA Applications

This section explains how to develop and install Oracle XA applications:

- DBA or System Administrator Responsibilities
- Application Developer Responsibilities
- Defining the xa_open String
- Developing and Installing XA Applications
- Managing Transaction Control with Oracle XA
- Migrating Precompiler or OCI Applications to TPM Applications
- Managing Oracle XA Library Thread Safety

- Using the DBMS_XA Package

DBA or System Administrator Responsibilities

The responsibilities of the DBA or system administrator are as follows:

1. Define the open string, with help from the application developer. This task is described in "Defining the xa_open String" on page 15-8.
2. Ensure that the static data dictionary view `DBA_PENDING_TRANSACTIONS` exists and grant the `SELECT` privilege to the view for all Oracle users specified in the `xa_open` string.

   Grant `FORCE TRANSACTION` privilege to the Oracle user who might commit or roll back pending (in-doubt) transactions that he or she created, using the command `COMMIT FORCE` *`local_tran_id`* or `ROLLBACK FORCE` *`local_tran_id`*.

   Grant `FORCE ANY TRANSACTION` privilege to the Oracle user who might commit or roll back XA transactions created by other users. For example, if user A might commit or roll back a transaction that was created by user B, user A must have `FORCE ANY TRANSACTION` privilege.

   In Oracle Database version 7 client applications, all Oracle Database accounts used by Oracle XA library must have the `SELECT` privilege on the dynamic performance view `V$XATRANS$`. This view must have been created during the XA library installation. If necessary, you can manually create the view by running the SQL script `xaview.sql` as Oracle Database user `SYS`.

   > **See Also:** Your Oracle Database platform-specific documentation for the location of the `catxpend.sql` script

3. Using the open string information, install the RM into the TPM configuration. Follow the TPM vendor instructions.

   The DBA or system administrator must be aware that a TPM system starts the process that connects to Oracle Database. See your TPM documentation to determine what environment exists for the process and what user ID it will have. Be sure that correct values are set for `$ORACLE_HOME` and `$ORACLE_SID` in this environment.

4. Grant the user ID write permission to the directory in which the system will write the XA trace file.

   > **See Also:** "Defining the xa_open String" on page 15-8 for information on how to specify an Oracle System Identifier (SID) or a trace directory that is different from the defaults

5. Start the relevant database instances to bring Oracle XA applications on-line. Perform this task before starting any TPM servers.

Application Developer Responsibilities

The responsibilities of the application developer are as follows:

1. Define the open string with help from the DBA or system administrator, as explained in "Defining the xa_open String" on page 15-8.
2. Develop the applications.

Observe special restrictions on transaction-oriented SQL statements for precompilers.

See Also: "Developing and Installing XA Applications" on page 15-6

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.

Topics:

- Syntax of the xa_open String
- Required Fields for the xa_open String
- Optional Fields for the xa_open String

Syntax of the xa_open String

You can define an open string with the syntax shown in Example 15–1.

Example 15–1 xa_open String

```
ORACLE_XA{+required_fields...} [+optional_fields...]
```

The following strings shows sample parameter settings:

```
ORACLE_XA+DB=MANAGERS+SqlNet=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=SID3+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/xalog
```

The following sections describe valid parameters for the *`required_fields`* and *`optional_fields`* placeholders.

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 might or might not be case-sensitive depending on the platform.
- There is no way to use the plus character (+) as part of the actual information string.

Required Fields for the xa_open String

The *`required_fields`* placeholder in Example 15–1 refers to any of the following name-value pairs described in Table 15–4.

Table 15–4 Required Fields of xa_open string

Syntax Element	Description
`Acc=P//`	Specifies that no explicit user or password information is provided and that the operating system authentication form is used. For more information see *Oracle Database Administrator's Guide*.
`Acc=P/`*`user`*`/`*`password`*	Specifies the username and password for a valid Oracle Database account. For example, `Acc=P/hr/hr` indicates that the user is `hr` and the password is `hr`. As described in "DBA or System Administrator Responsibilities" on page 15-7, ensure that hr has the `SELECT` privilege on the `DBA_PENDING_TRANSACTIONS` table.
`SesTm=`*`session_time_limit`*	Specifies the maximum number of seconds allowed in a transaction between one service and the next, or between a service and the commit or rollback of the transaction, before the system aborts the transaction. For example, `SesTM=15` indicates that the session idle time limit is 15 seconds. For example, if the TPM uses remote subprogram 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 value of `0` indicates no limit. 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`, then the `SesTM` setting is not effective after the parent process is terminated.

Optional Fields for the xa_open String

The *`optional_fields`* placeholder in Example 15–1 refers to any of the following name-value pairs described in Table 15–5.

Table 15–5 Optional Fields in the xa_open String

Syntax Element	Description
`NoLocal= true | false`	Specifies whether local transactions are allowed. The default value is `false`. If the application needs to disallow local transactions, then set the value to `true`.

Table 15–5 (Cont.) Optional Fields in the xa_open String

Syntax Element	Description
`DB=`*`db_name`*	Specifies the name used by Oracle Database precompilers to identify the database. For example, `DB=payroll` specifies that the database name is `payroll` and that the application server program uses that name in `AT` clauses. Application programs that use only the default database for the Oracle Database precompiler (that is, they do not use the `AT` clause in their SQL statements) must omit the `DB=`*`db_name`* clause in the open string. Applications that use explicitly named databases must indicate that database name in their `DB=`*`db_name`* field. Oracle Database Version 7 OCI programs need to call the `sqlld2` function to obtain the correct context for logon data area (`Lda_Def`), which is the equivalent of an OCI service context. Version 8 and higher OCI programs need to call the `xaoSvcCtx` function to get the `OCISvcCtx` service context. The *`db_name`* is not the sid and is not used to locate the database to be opened. Rather, it correlates the database opened by this open string with the name used in the application program to execute SQL statements. The sid is set from either the environment variable `ORACLE_SID` of the TPM application server or the sid given in the Oracle Net clause in the open string. The Oracle Net clause is described later in this section. Some TPM vendors provide a way to name a group of servers that use the same open string. You might find it convenient to choose the same name both for that purpose and for db_name.
`LogDir=`*`log_dir`*	Specifies the path name on the local system where the Oracle XA library error and tracing information is tomust be logged. The default is `$ORACLE_HOME/rdbms/log` if `ORACLE_HOME` is set; otherwise, it specifies the current directory. For example, `LogDir=/xa_trace` indicates that the logging information is located under the `/xa_trace` directory. Ensure that the directory exists and the application server can write to it.
`Objects= true | false`	Specifies whether the application is initialized in object mode. The default value is false. If the application needs to use certain API calls that require object mode, such as `OCIRawAssignBytes`, then set the value to `true`.
`MaxCur=`*`maximum_#_of_open_cursors`*	Specifies the number of cursors to be allocated when the database is opened. It serves the same purpose as the precompiler option `maxopencursors`. For example, `MaxCur=5` indicates that the precompiler tries to keep five open cursors cached. This parameter overrides the precompiler option `maxopencursors` that you might have specified in your source code or at compile time.
`SqlNet=`*`db_link`*	Specifies the Oracle Net database link to use to log on to the system. This string must be an entry in `tnsnames.ora`. For example, the string `SqlNet=inst1_disp` might connect to a shared server at instance 1 if so defined in `tnsnames.ora`. You can use the `SqlNet` parameter to specify the `ORACLE_SID` in cases where you cannot control the server environment variable. You must also use it when the server needs to access more than one Oracle Database instance. To use the Oracle Net string without actually accessing a remote database, use the Pipe driver. For example, specify `SqlNet=localsid1`, where `localsid1` is an alias defined in the `tnsnames.ora` file.

Table 15–5 (Cont.) Optional Fields in the xa_open String

Syntax Element	Description
`Loose_Coupling=true | false`	Specifies whether locks are shared. Oracle Database transaction branches within the same global transaction can be coupled tightly or loosely. If branches are loosely coupled, then they do not share locks. Set the value to `true` for loosely coupled branches. If branches are tightly coupled, then they share locks. Set the value to `false` for tightly coupled branches. The default value is `false`.
`SesWt=`*`session_wait_limit`*	Specifies the number of seconds Oracle Database waits for a transaction branch that is being used by another session before `XA_RETRY` is returned. The default value is 60 seconds.
`Threads=true | false`	Specifies whether the application is multithreaded. The default value is `false`. If the application is multithreaded, then the setting is `true`.

Using Oracle XA with Precompilers

When used in an Oracle XA application, cursors are valid only for the duration of the transaction. Explicit cursors must be opened after the transaction begins, and closed before the commit or rollback.

You have the following options 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. 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=maildb+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/logs
```

The `DB=`*`db_name`* is absent, indicating an empty database ID string.

The syntax of a SQL statement is:

```
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 might include the default database as well as one or more named databases. 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. Configure the following open strings in the transaction manager:

Example 15–2 Sample Open String Configuration

```
ORACLE_XA+DB=MANAGERS+SqlNet=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=SID3+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/xalog
```

There is no `DB=`*`db_name`* field in the last open string in Example 15–2.

In the application server program, 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` field) needs no declaration.

When doing the update, enter statements similar to the following:

```
EXEC SQL AT PAYROLL UPDATE Emp_Tab SET Sal=4500 WHERE Empno=7788;
EXEC SQL AT MANAGERS UPDATE Emp_Tab SET Mgr=7566 WHERE Empno=7788;
EXEC SQL UPDATE Emp_Tab SET Deptno=30 WHERE Empno=7788;
```

There is no `AT` clause in the last statement because it is referring to the default database.

In Oracle Database precompilers release 1.5.3 or later, you can use a character host variable in the `AT` clause, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
  DB_NAME1 CHARACTER(10);
  DB_NAME2 CHARACTER(10);
EXEC SQL END DECLARE SECTION;
    ...
SET DB_NAME1 = 'PAYROLL'
SET DB_NAME2 = 'MANAGERS'
    ...
EXEC SQL AT :DB_NAME1 UPDATE...
EXEC SQL AT :DB_NAME2 UPDATE...
```

Caution: Do not have XA applications create connections other than those created through `xa_open`. Work performed on non-XA connections is outside the global transaction and must be committed separately.

Using Oracle XA with OCI

Oracle Call Interface applications that use the Oracle XA library must not call `OCISessionBegin` to log on to the resource manager. Rather, the logon must be done through the TPM. The applications can execute the function `xaoSvcCtx` to obtain the service context structure when they need to access the resource manager.

In applications that need to pass the environment handle to OCI functions, you can also call `xaoEnv` to find that handle.

Because an application server can have multiple concurrent open Oracle Database resource managers, it must call the function `xaoSvcCtx` with the correct arguments to obtain the correct service context.

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

Managing Transaction Control with Oracle XA

When you use the XA library, 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 provided by the transaction manager, including the TX interface listed in Table 15–6, but not the XA Library Subroutines listed in Table 15–2.

The TMs typically control the transactions through the XA interface. This interface includes the functions described in Table 15–2.

Table 15–6 TX Interface Functions

TX Function	Description
`tx_open`	Logs into the resource manager(s)
`tx_close`	Logs out of the resource manager(s)
`tx_begin`	Starts a new transaction
`tx_commit`	Commits a transaction
`tx_rollback`	Rolls back the transaction

Most TPM applications use a client/server architecture in which an application client requests services and an application server provides them. The examples shown in "Examples of Precompiler Applications" on page 15-14 use such a client/server model. A service is a logical unit of work, which in the case of Oracle Database as the resource manager, comprises a set of SQL statements that perform a related unit of work.

For example, when a service named "credit" receives an account number and the amount to be credited, it executes SQL statements to update information in certain tables in the database. In addition, a service might request other services. For example, a "transfer fund" service might request services from a "credit" and "debit" service.

Typically, application clients request services from the application servers to perform tasks within a transaction. For some TPM systems, however, the application client itself can offer its own local services. As shown in "Examples of Precompiler Applications" on page 15-14, you can encode transaction control statements within either the client or the server.

To have more than one process participating in the same transaction, the TPM provides a communication API that enables 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 includes 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 server has already logged onto the RMs system, in a TPM-specific manner. Example 15–3 shows a transaction started by an application server.

Example 15–3 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 15–4 shows a transaction started by an application client.

Example 15–4 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 that uses the Oracle XA library, you must do the following:

1. Reorganize the application into a framework of "services" so 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, 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, OCIServerAttach, and OCIEnvCreate (for OCI) with tx_open. Replace the disconnect statements EXEC SQL COMMIT/ROLLBACK WORK RELEASE (for precompilers) or OCISessionEnd/OCIServerDetach (for OCI) with tx_close.

3. Ensure that the application replaces the regular commit or rollback statements for any global transactions and begins the transaction explicitly.

   For example, replace the COMMIT/ROLLBACK statements EXEC SQL COMMIT/ROLLBACK WORK (for precompilers), or OCITransCommit/OCITransRollback (for OCI) with tx_commit/tx_rollback and start the transaction by calling tx_begin.

   Note: The preceding is only true for global rather than local transactions. Commit or roll back local transactions with the Oracle API.

4. Ensure that the application resets the fetch state before ending a transaction. In general, use release_cursor=no. Use release_cursor=yes only when you are certain that a statement will execute only once.

Table 15–7 lists the TPM functions that replace regular Oracle Database statements when migrating precompiler or OCI applications to TPM applications.

Table 15–7 TPM Replacement Statements

Regular Oracle Database Statements	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)

Managing Oracle XA Library Thread Safety

If you use a transaction monitor that supports threads, then the Oracle XA library enables you to write applications that are thread-safe. Nevertheless, keep certain issues in mind.

A **thread of control** (or thread) refers to the set of connections to resource managers. In an nonthreaded system, each process is considered a thread of control because each process has its own set of connections to RMs and maintains its own independent resource manager table. In a threaded system, each thread has an autonomous set of connections to RMs and each thread maintains a *private* RM table. This private table must be allocated for each new thread and de-allocated when the thread terminates, even if the termination is abnormal.

Note: In Oracle Database, each thread that accesses the database must have its own connection.

Topics:

- Specifying Threading in the Open String
- Restrictions on Threading in Oracle XA

Specifying Threading in the Open String

The `xa_open` string provides the clause `Threads=`. You must specify this clause as `true` to enable the use of threads by the TM. The default is `false`. In most cases, the TM creates the threads; the application does not know when a new thread is created. Therefore, it is advisable to allocate a service context on the stack within each service that is written for a TM application. Before doing any Oracle Database-related calls in that service, you must call the `xaoSvcCtx` function to retrieve the initialized OCI service context. You can then use this context for OCI calls within the service.

Restrictions on Threading in Oracle 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 TM vendor.
- The Pro* statements `EXEC SQL ALLOCATE` and `EXEC SQL USE` are not supported. Therefore, when threading is enabled, you cannot use embedded SQL statements across non-XA connections.
- If one thread in a process connects to Oracle Database through XA, then all other threads in the process that connect to Oracle Database must also connect through XA. You cannot connect through `EXEC SQL CONNECT` in one thread and through `xa_open` in another thread.

Using the DBMS_XA Package

PL/SQL applications can use the Oracle XA library by means of the `DBMS_XA` package, which is described in *Oracle Database PL/SQL Packages and Types Reference*.

In Example 15–5, one PL/SQL session starts a transaction but does not commit it, a second session resumes the transaction, and a third session commits the transaction.

Example 15–5 Using the DBMS_XA Package

```
REM Session 1 starts a transaction and does some work.
CONNECT HR/password
SET SERVEROUTPUT ON
DECLARE
  rc  PLS_INTEGER;
  oer PLS_INTEGER;
  xae EXCEPTION;
BEGIN
  rc  := DBMS_XA.XA_START(DBMS_XA_XID(123), DBMS_XA.TMNOFLAGS);

  IF rc!=DBMS_XA.XA_OK THEN
    oer := DBMS_XA.XA_GETLASTOER();
    DBMS_OUTPUT.PUT_LINE('ORA-' || oer || ' occurred, XA_START failed');
    RAISE xae;
  ELSE DBMS_OUTPUT.PUT_LINE('XA_START(new xid=123)     OK');
  END IF;

  UPDATE employees SET salary=salary*1.1 WHERE employee_id = 100;
  rc  := DBMS_XA.XA_END(DBMS_XA_XID(123), DBMS_XA.TMSUSPEND);

  IF rc!=DBMS_XA.XA_OK THEN
    oer := DBMS_XA.XA_GETLASTOER();
    DBMS_OUTPUT.PUT_LINE('ORA-' || oer || ' occurred, XA_END failed');
    RAISE xae;
  ELSE DBMS_OUTPUT.PUT_LINE('XA_END(suspend xid=123)   OK');
  END IF;

  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
       ('XA error('||rc||') occurred, rolling back the transaction ...');
      rc := DBMS_XA.XA_END(DBMS_XA_XID(123), DBMS_XA.TMSUCCESS);
      rc := DBMS_XA.XA_ROLLBACK(DBMS_XA_XID(123));

      IF rc != DBMS_XA.XA_OK THEN
        oer := DBMS_XA.XA_GETLASTOER();
        DBMS_OUTPUT.PUT_LINE('XA-'||rc||', ORA-' || oer ||
         ' XA_ROLLBACK does not return XA_OK');
        raise_application_error(-20001, 'ORA-'||oer||
         ' error in rolling back a failed transaction');
      END IF;

      raise_application_error(-20002, 'ORA-'||oer||
       ' error in transaction processing, transaction rolled back');
END;
/
SHOW ERRORS
DISCONNECT

REM Session 2 resumes the transaction and does some work.
CONNECT HR/password
SET SERVEROUTPUT ON
DECLARE
  rc  PLS_INTEGER;
  oer PLS_INTEGER;
  s   NUMBER;
  xae EXCEPTION;
BEGIN
  rc  := DBMS_XA.XA_START(DBMS_XA_XID(123), DBMS_XA.TMRESUME);

  IF rc!=DBMS_XA.XA_OK THEN
    oer := DBMS_XA.XA_GETLASTOER();
    DBMS_OUTPUT.PUT_LINE('ORA-' || oer || ' occurred, xa_start failed');
    RAISE xae;
  ELSE DBMS_OUTPUT.PUT_LINE('XA_START(resume xid=123)  OK');
  END IF;

  SELECT salary INTO s FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE('employee_id = 100, salary = ' || s);
  rc  := DBMS_XA.XA_END(DBMS_XA_XID(123), DBMS_XA.TMSUCCESS);

  IF rc!=DBMS_XA.XA_OK THEN
```

```
      oer := DBMS_XA.XA_GETLASTOER();
      DBMS_OUTPUT.PUT_LINE('ORA-' || oer || ' occurred, XA_END failed');
      RAISE xae;
    ELSE DBMS_OUTPUT.PUT_LINE('XA_END(detach xid=123)    OK');
    END IF;

    EXCEPTION
      WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE
         ('XA error('||rc||') occurred, rolling back the transaction ...');
        rc := DBMS_XA.XA_END(DBMS_XA_XID(123), DBMS_XA.TMSUCCESS);
        rc := DBMS_XA.XA_ROLLBACK(DBMS_XA_XID(123));

        IF rc != DBMS_XA.XA_OK THEN
          oer := DBMS_XA.XA_GETLASTOER();
          DBMS_OUTPUT.PUT_LINE('XA-'||rc||', ORA-' || oer ||
           ' XA_ROLLBACK does not return XA_OK');
          raise_application_error(-20001, 'ORA-'||oer||
           ' error in rolling back a failed transaction');
        END IF;

        raise_application_error(-20002, 'ORA-'||oer||
         ' error in transaction processing, transaction rolled back');
  END;
  /
  SHOW ERRORS
  DISCONNECT

  REM Session 3 commits the transaction.
  CONNECT HR/password
  SET SERVEROUTPUT ON
  DECLARE
    rc  PLS_INTEGER;
    oer PLS_INTEGER;
    xae EXCEPTION;
  BEGIN
    rc  := DBMS_XA.XA_COMMIT(DBMS_XA_XID(123), TRUE);

    IF rc!=DBMS_XA.XA_OK THEN
      oer := DBMS_XA.XA_GETLASTOER();
      DBMS_OUTPUT.PUT_LINE('ORA-' || oer || ' occurred, XA_COMMIT failed');
      RAISE xae;
    ELSE DBMS_OUTPUT.PUT_LINE('XA_COMMIT(commit xid=123)  OK');
    END IF;

    EXCEPTION
      WHEN xae THEN
        DBMS_OUTPUT.PUT_LINE
         ('XA error('||rc||') occurred, rolling back the transaction ...');
        rc := DBMS_XA.XA_ROLLBACK(DBMS_XA_XID(123));

        IF rc != DBMS_XA.XA_OK THEN
          oer := DBMS_XA.XA_GETLASTOER();
          DBMS_OUTPUT.PUT_LINE('XA-'||rc||', ORA-' || oer ||
           ' XA_ROLLBACK does not return XA_OK');
          raise_application_error(-20001, 'ORA-'||oer||
           ' error in rolling back a failed transaction');
        END IF;

        raise_application_error(-20002, 'ORA-'||oer||
         ' error in transaction processing, transaction rolled back');
  END;
  /
  SHOW ERRORS
  DISCONNECT
  QUIT
```

Troubleshooting XA Applications

Topics:

- Accessing Oracle XA Trace Files
- Managing In-Doubt or Pending Oracle XA Transactions
- Using SYS Account Tables to Monitor Oracle XA Transactions

Accessing Oracle XA Trace Files

The Oracle XA library logs any error and tracing information to its trace file. This information is useful in supplementing the XA error codes. For example, it can indicate whether an `xa_open` failure is caused by an incorrect open string, failure to find the Oracle Database instance, or a logon authorization failure.

The name of the trace file is `xa_`*`db_namedate`*`.trc`, where *`db_name`* is the database name 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 `NULL`.

For example, `xa_NULL06022005.trc` indicates a trace file that was created on June 2, 2005. Its `DB` field was not specified in the open string when the resource manager was opened. The filename `xa_Finance12152004.trc` indicates a trace file was created on December 15, 2004. 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.

Suppose that a trace file contains the following contents:

```
1032.12345.2:  ORA-01017:  invalid username/password;  logon denied
1032.12345.2:  xaolgn:  XAER_INVAL;  logon denied
```

Table 15–8 explains the meaning of each element.

Table 15–8 Sample Trace File Contents

String	Description
`1032`	The time when the information is logged.
`12345`	The process ID (PID).
`2`	The resource manager ID.
`xaolgn`	The name of the module.
`XAER_INVAL`	The error returned as specified in the XA standard.
`ORA-01017`	The Oracle Database information that was returned.

Topics:

- xa_open String DbgFl
- Trace File Locations

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. You can set it to any combination of the following values:

- `0x1`, which enables you to trace the entry and exit to each subprogram in the XA interface. This value can be useful in seeing exactly which XA calls the TP Monitor is making and which transaction identifier it is generating.
- `0x2`, which enables you to trace the entry to and exit from other nonpublic XA library routines. This is generally of use only to Oracle Database developers.
- `0x4`, which enables you to trace various other "interesting" calls made by the XA library, such as specific calls to the OCI. This is generally of use only to Oracle Database developers.

Note: The flags are independent bits of an `ub4`, so to obtain printout from two or more flags, you must set a combined value of the flags.

Trace File Locations

The XA application determines a location for the trace file according to the following algorithm:

1. The `LogDir` directory specified in the open string.
2. If you do not specify `LogDir` in the open string, then the Oracle XA application attempts to create the trace file in the following directory (if the Oracle home is accessible):
   - `%ORACLE_HOME%\rdbms\trace` on Windows
   - `$ORACLE_HOME/rdbms/log` on Linux and UNIX
3. If the Oracle XA application cannot determine where the Oracle home is located, then the application creates the trace file in the current working directory.

Managing In-Doubt or Pending Oracle XA Transactions

In-doubt or pending transactions are transactions that were prepared but not committed to the database. In general, the TM provided by the TPM system resolves any failure and recovery of in-doubt or pending transactions. The DBA might have to override an in-doubt transaction if the following situations occur:

- It is locking data that is required by other transactions.
- It is not resolved in a reasonable amount of time.

See the TPM documentation for more information about overriding in-doubt transactions in such circumstances and about how to decide whether to commit or roll back the in-doubt transaction.

Using SYS Account Tables to Monitor Oracle XA Transactions

The following views under the Oracle Database `SYS` account contain transactions generated by regular Oracle Database applications and Oracle XA applications:

- `DBA_PENDING_TRANSACTIONS`
- `V$GLOBAL_TRANSACTION`
- `DBA_2PC_PENDING`
- `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, the following SQL statement provide 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$GLOBAL_TRANSACTION`. Whereas `DBA_PENDING_TRANSACTIONS` gives a list of prepared transactions, `V$GLOBAL_TRANSACTION` provides a list of all active global transactions.

Oracle XA Issues and Restrictions

This section contains the following topics:

Using Database Links in Oracle XA Applications

Oracle XA applications can access other Oracle Database instances through database links with the following restrictions:

- They must use the shared server configuration.

  The transaction processing monitors (TPMs) use shared servers to open the connection to an Oracle Database A. Then the operating system network connection required for the database link is opened by the dispatcher instead of a dedicated server process. This allows different services or threads to operate on the transaction.

If this restriction is not satisfied, then when you use database links within an XA transaction, it creates an operating system network connection between the dedicated server process and the other Oracle Database B. Because this network connection cannot be moved from one dedicated server process to another, you cannot detach from this dedicated server process of database A. Then when you access the database B through a database link, you receive an ORA-24777 error.

- The other database being accessed must be another Oracle Database.

Assuming that these restrictions are satisfied, Oracle Database allows such links and propagates the transaction protocol (prepare, rollback, and commit) to the other Oracle Database instances.

If using the shared server configuration is not possible, then access the remote database through the Pro*C/C++ application by using `EXEC SQL AT` syntax.

The `init.ora` parameter `OPEN_LINKS_PER_INSTANCE` specifies the number of open database link connections that can be migrated. 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 database link connection provided the user that created the connection is the same as the user who created the transaction. This parameter is different from the `init.ora` parameter `OPEN_LINKS`, which specifies the maximum number of concurrent open connections (including database links) to remote databases in one session. The `OPEN_LINKS` parameter does not apply to XA applications.

Managing Transaction Branches in Oracle XA Applications

Oracle Database transaction branches within the same global transaction can be coupled tightly or loosely. If the transaction branches are **tightly coupled**, then they share locks. Consequently, pre-`COMMIT` updates in one transaction branch are visible in other branches that belong to the same global transaction. In loosely coupled transaction branches, the branches do not share locks and do not see updates in other branches.

In a tightly coupled branch, Oracle Database obtains the DX lock before executing any statement. Because the system does not obtain a lock before executing the statement, loosely coupled transaction branches result in greater concurrency. The disadvantage is that all transaction branches must go through the two phases of commit, that is, the system cannot use XA one-phase optimization.

Table 15–9 summarizes the trade-offs between tightly coupled branches and loosely coupled branches.

Table 15–9 Tightly and Loosely Coupled Transaction Branches

Attribute	Tightly Coupled Branches	Loosely Coupled Branches
Two Phase Commit	Read-only optimization	Two phases
	[prepare for all branches, commit for last branch]	[prepare and commit for all branches]
Serialization	Database call	None

Using Oracle XA with Oracle Real Application Clusters (Oracle RAC)

As of Release 11.1, an XA transaction can span Oracle RAC instances, allowing any application that uses XA to take full advantage of the Oracle RAC environment, enhancing the availability and scalability of the application.

GLOBAL_TXN_PROCESSES Initialization Parameter

The initialization parameter `GLOBAL_TXN_PROCESSES` specifies the initial number of GTX*n* background processes for each Oracle RAC instance. Its default value is 1.

Leave this parameter at its default value cluster-wide if distributed transactions might span more than one Oracle RAC instance. This allows the units of work performed across these Oracle RAC instances to share resources and act as a single transaction (that is, the units of work are tightly coupled). It also allows 2PC requests to be sent to any node in the cluster.

> **See Also:** *Oracle Database Reference* for more information about `GLOBAL_TXN_PROCESSES`.

> **Note:** If you leave the initialization parameter `GLOBAL_TXN_PROCESSES` at its default setting in the initialization file of every Oracle RAC instance, you do not need to read the following topics, which apply only to the Distributed Transaction Processing (DTP) services introduced in release 10.2:
>
> - Managing Transaction Branches on Oracle RAC
> - Managing Instance Recovery in Oracle RAC with DTP Services (10.2)
> - Global Uniqueness of XIDs in Oracle RAC
> - Tight and Loose Coupling

Managing Transaction Branches on Oracle RAC

Oracle Database permits different instances to operate on different transaction branches in Oracle RAC. For example, Node 1 can operate on branch A while Node 2 operates on branch B. Before Release 11.1, if transaction branches were on different instances, then they were loosely coupled and did not share locks. In this case, Oracle Database treated different units of work in different application threads as separate entities that did not share resources.

A different case is when multiple instances operate on a single transaction branch. For example, assume that a single transaction lands on Node 1 and Node 2 as follows:

Node 1

1. `xa_start`
2. SQL operations
3. `xa_end` (SUSPEND)

Node 2

1. `xa_start` (RESUME)
2. `xa_prepare`
3. `xa_commit`
4. `xa_end`

In the immediately preceding sequence, Oracle Database returns an error because Node 2 must not resume a branch that is physically located on a different node (Node 1).

Before Release 11.1, the way to achieve tight coupling in Oracle RAC was to use **Distributed Transaction Processing (DTP) services**, that is, services whose cardinality (one) ensured that all tightly-coupled branches landed on the same instance—whether or not load balancing was enabled. Middle-tier components addressed Oracle Database through a common logical database service name that mapped to a single Oracle RAC instance at any point in time. An intermediate name resolver for the database service hid the physical characteristics of the database instance. DTP services enabled all participants of a tightly-coupled global transaction to create branches on one instance.

As of Release 11.1, the DTP service is no longer required to support XA transactions with tightly coupled branches. By default, tightly coupled branches that land on different RAC instances remain tightly coupled; that is, they share locks and resources across RAC instances.

For example, when you use a DTP service, the following sequence of actions occurs on the same instance:

1. `xa_start`
2. SQL operations
3. `xa_end` (SUSPEND)
4. `xa_start` (RESUME)
5. SQL operations
6. `xa_prepare`
7. `xa_commit` or `xa_rollback`

Moreover, multiple tightly-coupled branches land on the same instance if each addresses the Oracle RM with the same DTP service.

To leverage all instances in the cluster, create multiple DTP services, with one or more on each node that hosts distributed transactions. All branches of a global distributed transaction exist on the same instance. Thus, you can leverage all instances and nodes of an Oracle RAC cluster to balance the load of many distributed XA transactions, thereby maximizing application throughput.

See Also: *Oracle Real Application Clusters Administration and Deployment Guide* to learn how to manage distributed transactions in a Real Application Clusters configuration.

Managing Instance Recovery in Oracle RAC with DTP Services (10.2)

Prior to Oracle Database 10g Release 2 (10.2), TM was responsible for detecting failure and triggering failover and failback in Oracle RAC. To ensure that information about in-doubt transactions was propagated to `DBA_2PC_PENDING`, TM had to call `xa_recover` before resolving the in-doubt transactions. If an instance failed, then the XA client library could not fail over to another instance until it had run the `SYS.DBMS_XA.DIST_TXN_SYNC` procedure to ensure that the undo segments of the failed instance were recovered. As of Release 10.2, there is no such requirement to call `xa_recover` in cases where the TM has enough information about in-flight transactions.

Note: In releases subsequent to Oracle Database 9*i* Release 2, `xa_recover` is required to wait for distributed DML to complete on remote sites.

Using DTP services in Oracle RAC has the following benefits:

- Automates instance failure detection.
- Automates instance failover and failback. When an instance fails, the DTP service hosted on this instance fails over to another instance. The failover forces clients to reconnect; nevertheless, the logical names for the service remain the same. Failover is automatic and does not require an administrator intervention. The administrator can induce failback by a service relocate statement, but all failback-related recovery is automatically handled within the database server.
- Enables Oracle Database rather than the client to drive instance recovery. The database does not require middle-tier TM involvement to determine the state of transactions prepared by other instances.

See Also: *Oracle Real Application Clusters Administration and Deployment Guide* to learn how to manage instance recovery

Global Uniqueness of XIDs in Oracle RAC

Before Release 11.1, Oracle RAC database cannot determine whether a given XID is unique for XA transactions throughout the cluster.

For example, suppose that there is an XID `Fmt(x).Tx(1).Br(1)` on Oracle RAC instance 1 and another XID `Fmt(x).Tx(1).Br(1)` on Oracle RAC instance 2. Each of these can start a branch and execute SQL even though the XID is not unique across Oracle RAC instances.

As of Release 11.1, Oracle RAC database detects the duplicate XIDs across RAC instances and prevents a branch with a duplicate XID from starting.

Tight and Loose Coupling

Oracle Database transaction branches within the same global transaction can be coupled either tightly or loosely (for details, see "Managing Transaction Branches in Oracle XA Applications" on page 15-22). Ordinarily, coupling type is determined by the value of the `Loose_Coupling` field of the `xa_open` string (see Table 15–5 on page 15-9). However, if branches are landed on different Oracle RAC instances when running Oracle Real Application Clusters, they are loosely coupled even if `Loose_Coupling=false`.

SQL-Based Oracle XA Restrictions

This section describes restrictions concerning the following SQL operations:

- Rollbacks and Commits
- DDL Statements
- Session State
- EXEC SQL

Rollbacks and Commits

Because the transaction manager is responsible for coordinating and monitoring the progress of the global transaction, the application must not contain any Oracle Database-specific statement that independently rolls back or commits a global transaction. However, you can use rollbacks and commits in a local transaction.

Do not use `EXEC SQL ROLLBACK WORK` for precompiler applications when you are in the middle of a global transaction. Similarly, an OCI application must not execute

`OCITransRollback`, or the Version 7 equivalent `orol`. You can roll back a global transaction by calling `tx_rollback`.

Similarly, a precompiler application must not have the `EXEC SQL COMMIT WORK` statement in the middle of a global transaction. An OCI application must not execute `OCITransCommit` or the Version 7 equivalent `ocom`. For example, use `tx_commit` or `tx_rollback` to end a global transaction.

DDL Statements

Because a DDL SQL statement, such as `CREATE TABLE`, implies an implicit commit, the Oracle XA application cannot execute any DDL SQL statements.

Session State

Oracle Database does not guarantee that session state will be valid between TPM services. For example, if a TPM service updates a session variable (such as a global package variable), then another TPM service that executes as part of the same global transaction might not see the change. Use savepoints only within a TPM service. The application must not refer to a savepoint that was created in another TPM service. Similarly, an application must not attempt to fetch from a cursor that was executed in another TPM service.

EXEC SQL

Do not use the `EXEC SQL` statement to connect or disconnect. That is, do not use `EXEC SQL CONNECT`, `EXEC SQL COMMIT WORK RELEASE` or `EXEC SQL ROLLBACK WORK RELEASE`.

Miscellaneous Restrictions

- Oracle Database does not support association migration (a means whereby a transaction manager might resume a suspended branch association in another branch).
- The optional XA feature asynchronous XA calls is not supported.
- Set the `TRANSACTIONS` initialization parameter to the expected number of concurrent global transactions. The initialization parameter `OPEN_LINKS_PER_INSTANCE` specifies the number of open database link connections that can be migrated. These database link connections are used by XA transactions so that the connections are cached after a transaction is committed.

  See Also: "Using Database Links in Oracle XA Applications" on page 15-21

- The maximum number of `xa_open` calls for each thread is 32.
- When building an XA application based on TP-monitor, ensure that the TP-monitors libraries (that define the symbols `ax_reg` and `ax_unreg`) are placed in the link line before Oracle Database's client shared library. If your platform does not support shared libraries or if your linker is not sensitive to ordering of libraries in the link line, use Oracle Database's nonshared client library. These link restrictions are applicable only when using XA's dynamic registration (Oracle XA switch `xaoswd`).

16 Developing Applications on the Publish-Subscribe Model

This chapter explains how to develop applications on the publish-subscribe model.

Topics:

Introduction to the Publish-Subscribe Model

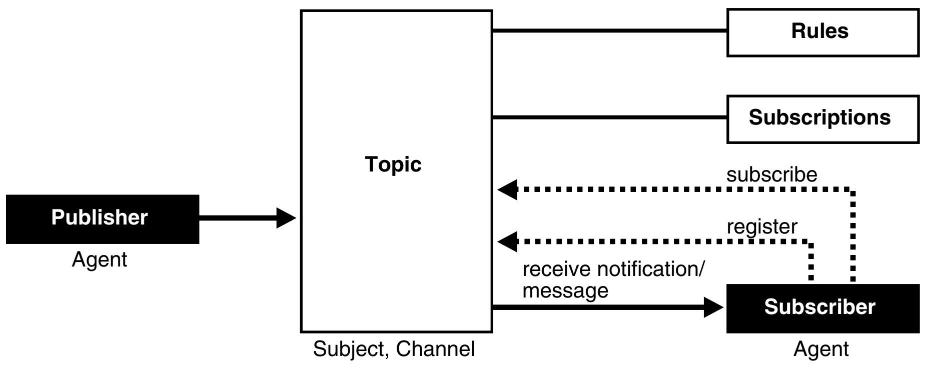
Because the database is the most significant resource of information within the enterprise, Oracle created a publish-subscribe solution for enterprise information delivery and messaging to complement this role.

Networking technologies and products enable a high degree of connectivity across a large number of computers, applications, and users. In these environments, it is important to provide asynchronous communications for the class of distributed systems that operate in a loosely-coupled and autonomous fashion, and which require operational immunity from network failures. This requirement is 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 16–1 illustrates publish and subscribe functionality.

Figure 16–1 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 run time, publishers post messages to various queues. The queue (in other words, the delivery mechanisms of the underlying infrastructure) then delivers messages that match the various subscriptions to the appropriate subscribers.

Publish-Subscribe Architecture

Oracle Database includes the following features to support database-enabled publish-subscribe messaging:

- Database Events
- Oracle Advanced Queuing
- Client Notification

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: *Oracle Database PL/SQL Language Reference*

Oracle Advanced Queuing

Oracle Advanced Queuing (AQ) supports a queue-based publish-subscribe paradigm. Database queues serve as a durable store for messages, along with capabilities to allow publish and subscribe based on queues. A rules-engine and subscription service dynamically route messages to recipients based on expressed interest. This allows decoupling of addressing between senders and receivers to complement the existing explicit sender-receiver message addressing.

See Also: *Oracle Streams Advanced Queuing User's Guide*

Client Notification

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

queue

A **queue** is an entity that supports the notion of named subjects of interest. Queues can be characterized as persistent or nonpersistent (lightweight).

A **persistent queue** serves as a durable container for messages. Messages are delivered in a deferred and reliable mode.

The underlying infrastructure of a **nonpersistent, or lightweight, queue** pushes the messages published to connected clients in a lightweight, at-best-once, manner.

agent

Publishers and subscribers are internally represented as agents.

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.

client

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. Several clients can act on behalf of a single agent. 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 is to be done, and a callback, indicating *how* there delivery is to 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.

receiving 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 (nonpersistent 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/password
DROP USER pubsub CASCADE;
CREATE USER pubsub IDENTIFIED BY password;
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/password
```

Scenario: This example shows how database 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. User `pubsub` needs `AQ_ADMINISTRATOR_ROLE` privileges to use AQ functionality.

```
Rem ------------------------------------------------------
REM create queue table for persistent multiple consumers:
Rem ------------------------------------------------------

CONNECT pubsub/password;

Rem  Create or replace a queue table
BEGIN
DBMS_AQADM.CREATE_QUEUE_TABLE(
   Queue_table        =>  'Pubsub.Raw_msg_table',
   Multiple_consumers =>   TRUE,
   Queue_payload_type =>  'RAW',
   Compatible         =>  '8.1');
END;
/
Rem ------------------------------------------------------
Rem  Create a persistent queue for publishing messages:
Rem ------------------------------------------------------

Rem  Create a queue for logon events
begin
BEGIN
   DBMS_AQADM.CREATE_QUEUE(
         Queue_name     =>   'Pubsub.Logon',
         Queue_table    =>   'Pubsub.Raw_msg_table',
         Comment        =>   'Q for error triggers');
END;
/

Rem ------------------------------------------------------
Rem  Start the queue:
Rem ------------------------------------------------------

BEGIN
   DBMS_AQADM.START_QUEUE('pubsub.logon');
END;
```

```
/

Rem ------------------------------------------------------
Rem  define new_enqueue for convenience:
Rem ------------------------------------------------------

CREATE OR REPLACE PROCEDURE New_enqueue(
               Queue_name      IN VARCHAR2,
               Payload         IN RAW ,
               Correlation     IN VARCHAR2 := NULL,
               Exception_queue IN VARCHAR2 := NULL)
AS

Enq_ct     DBMS_AQ.Enqueue_options_t;
Msg_prop   DBMS_AQ.Message_properties_t;
Enq_msgid  RAW(16);
Userdata   RAW(1000);

BEGIN
   Msg_prop.Exception_queue := Exception_queue;
   Msg_prop.Correlation := Correlation;
   Userdata := Payload;

DBMS_AQ.ENQUEUE(Queue_name, Enq_ct, Msg_prop, Userdata, Enq_msgid);
END;
/

Rem ------------------------------------------------------
Rem  add subscriber with rule based on current user name,
Rem  using correlation_id
Rem ------------------------------------------------------


DECLARE
Subscriber Sys.Aq$_agent;
BEGIN
   Subscriber := sys.aq$_agent('SNOOP', NULL, NULL);
DBMS_AQADM.ADD_SUBSCRIBER(
    Queue_name         => 'Pubsub.logon',
    Subscriber         => subscriber,
    Rule               => 'CORRID = ''SCOTT'' ');
END;
/

Rem ------------------------------------------------------
Rem  create a trigger on logon on database:
Rem ------------------------------------------------------


Rem  create trigger on after logon:
CREATE OR REPLACE TRIGGER pubsub.Systrig2
   AFTER LOGON
   ON DATABASE
   BEGIN
      New_enqueue('Pubsub.Logon', HEXTORAW('9999'), Dbms_standard.login_user);
   END;
/
```

- After subscriptions are created, the next step is for the client to register for notification using callback functions. This is done using the Oracle Call Interface

(OCI). The following code performs necessary steps for registration. The initial steps of allocating and initializing session handles are omitted here for sake of clarity.

```
ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;

/* callback function for notification of logon of user 'scott' on database: */

ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *pay;
ub4 payl;
dvoid *desc;
ub4 mode;
{
    printf("Notification : User Scott Logged on\n");
}

int main()
{
    OCISession *authp = (OCISession *) 0;
    OCISubscription *subscrhpSnoop = (OCISubscription *)0;

    /*****************************************************
       Initialize OCI Process/Environment
       Initialize Server Contexts
       Connect to Server
       Set Service Context
    ******************************************************/

    /* Registration Code Begins */

    /* Each call to initSubscriptionHn allocates
           and Initialises a Registration Handle */


    initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle */
        "ADMIN:PUBSUB.SNOOP", /* subscription name */
                  /* <agent_name>:<queue_name> */
        (dvoid*)notifySnoop); /* callback function */

     /*****************************************************
       The Client Process does not need a live Session for Callbacks
       End Session and Detach from Server
     ******************************************************/

    OCISessionEnd ( svchp,  errhp, authp, (ub4) OCI_DEFAULT);

    /* detach from server */
    OCIServerDetach( srvhp, errhp, OCI_DEFAULT);

    while (1)     /* wait for callback */
        sleep(1);

}

void initSubscriptionHn (subscrhp,
subscriptionName,
func)
```

```
OCISubscription **subscrhp;
char* subscriptionName;
dvoid * func;
{

    /* allocate subscription handle: */

    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)subscrhp,
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

    /* set subscription name in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) subscriptionName,
        (ub4) strlen((char *)subscriptionName),
        (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

    /* set callback function in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) func, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) 0, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

    /* set namespace in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) &namespace, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

    checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,

        OCI_DEFAULT));
}
```

If user `SCOTT` logs on to the database, the client is notified, and the call back function `notifySnoop` is invoked.

17 Using the Identity Code Package

The Identity Code Package is a feature in the Oracle Database that offers tools and techniques to store, retrieve, encode, decode, and translate between various product or identity codes, including Electronic Product Code (EPC), in an Oracle Database. The Identity Code Package provides new data types, metadata tables and views, and PL/SQL packages for storing EPC standard RFID tags or new types of RFID tags in a user table.

The Identity Code Package empowers the Oracle Database with the knowledge to recognize EPC coding schemes, support efficient storage and component level retrieval of EPC data, and complies with the EPCglobal Tag Data Translation 1.0 (TDT) standard that defines how to decode, encode, and translate between various EPC RFID tag representations.

The Identity Code Package also provides an extensible framework that allows developers to utilize pre-existing coding schemes with their applications that are not included in the EPC standard and make the Oracle Database adaptable to these older systems as well as to any evolving identity codes that may some day be part of a future EPC standard.

The Identity Code Package also lets developers create their own identity codes by first registering the new encoding category, registering the new encoding type, and then registering the new components associated with each new encoding type.

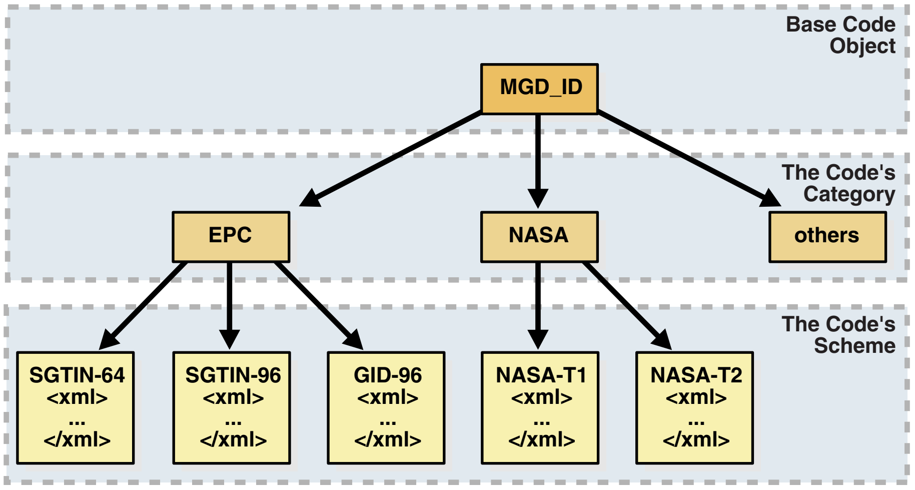
Topics.

Identity Concepts

A new database object `MGD_ID` is defined that lets users use EPC standard identity codes as well as use their own existing identity codes. See "Electronic Product Code (EPC) Concepts" on page 17-22 for a brief description of EPC concepts. The `MGD_ID` object serves as the base code object to which belong certain categories, or types of the

RFID tag, such as the EPC category, NASA category, and many other categories. Each category has a set of tag schemes or documents that define tag representation structures and their components. For the EPC category, the metadata needed to define encoding schemes (SGTIN-64, SGTIN-96, GID-96, and so forth) representing different encoding types (defined in the EPC standard v1.1) is preloaded by default into the database. Users can define encoding their own categories and schemes as shown in Figure 17–1 and load these into the database as well.

Figure 17–1 RFID Code Categories and Their Schemes



An `MGD_ID` object contains two attributes, a category_id and a list of components consisting of name-value pairs. When `MGD_ID` objects are stored, the tag representation must be parsed into these component name-value pairs upon object creation.

EPC standard version 1.1 defines one General Identifier type (GID) that is independent of any known, existing code schemes, five Domain Identifier types that are based on EAN.UCC specifications, and the identity type United States Department of Defense (USDOD). The five EAN.UCC based identity types are the serialized global trade identification number (SGTIN), the serial shipping container code (SSCC), the serialized global location number (SGLN), the global returnable asset identifier (GRAI) and the global individual asset identifier (GIAI).

Except GID, which has only one bit-level encoding, all the other identity types each have two encodings depending on their length: 64-bit and 96-bit. So in total there are thirteen different standard encodings for EPC tags. In addition, tags can be encoded in representations other than binary, such as the tag URI and pure identity representations.

Each EPC encoding has its own structure and organization, see Table 17–1. Note that the EPC encoding structure field names relate to the names in the parameter_list parameter name-value pairs in the Identity Code Package API. For example, for SGTIN-64, the structure field names are Filter Value, Company Prefix Index, Item Reference, and Serial Number.

Table 17–1 General Structure of EPC Encodings

Encoding Name	Header Length in bits	Field Names (parameter_list name-value pairs) and (length in bits)
GID-96	8	General Manager Number (8), Object Class (24), Serial Number (36)

Table 17–1 (Cont.) General Structure of EPC Encodings

Encoding Name	Header Length in bits	Field Names (parameter_list name-value pairs) and (length in bits)
SGTIN-64	2	Filter Value (3), Company Prefix Index (14), Item Reference 20), Serial Number (25)
SGTIN-96	8	Filter Value (3), Partition (3), Company Prefix (20-40), Item Reference (24-4), Serial Number (38)
SSCC-64	8	Filter Value (3), Company Prefix Index (14), Serial Reference (39)
SSCC-96	8	Filter Value (3), Partition (3), Company Prefix (20-40), Serial Reference (38-18), Unallocated (24)
SGLN-64	8	Filter Value (3), Company Prefix Index (14), Location Reference (20), Serial Number (19)
SGLN-96	8	Filter Value (3), Partition (3), Company Prefix (20-40), Location Reference (21-1), Serial Number (41)
GRAI-64	8	Filter Value (3), Company Prefix Index (14), Asset Type (20), Serial Number (19)
GRAI-96	8	Filter Value (3), Partition (3), Company Prefix (20-40), Asset Type (24-4), Serial Number (38)
GIAI-64	8	Filter Value (3), Company Prefix Index (14), Individual Asset Reference (39)
GIAI-96	8	Filter Value (3), Partition (3), Company Prefix (20-40), Individual Asset Reference (62-42)
USDOD-64	8	Filter Value (2), Government Managed Identifier (30), Serial Number (24)
USDOD-96	8	Filter Value (4), Government Managed Identifier (48), Serial Number (36)

EPCglobal defines eleven tag schemes (GID-96, SGTIN-64, SGTIN-96, and so forth). Each of these schemes has various representations; at this time, the most often used are BINARY, TAG_URI, and PURE_IDENTITY. For example, information in an SGTIN-64 can be represented in the following ways:

```
BINARY: 1001100000000000001000001110110001000010000011111110011000110010
PURE_IDENTITY:  urn:epc:id:sgtin:0037000.030241.1041970
TAG_URI: urn:epc:tag:sgtin-64:3.0037000.030241.1041970
LEGACY: gtin=00037000302414;serial=1041970
ONS_HOSTNAME: 030241.0037000.sgtin.id.onsepc.com
```

Note that some of these representations contain all information about the tag (BINARY and TAG_URI), while other representations contain only partial information (PURE_IDENTITY). It is therefore possible to translate a tag from its TAG_URI to its PURE_IDENTITY representation, but it is not possible to translate in the other direction without additional information being provided, namely the filter value must be supplied.

EPCglobal released a Tag Data Translation 1.0 (TDT) standard that defines how to decode, encode, and translate between various EPC RFID tag representations. Decoding refers to parsing a given representation into field/value pairs, and encoding refers to reconstructing representations from these fields. Translating refers to decoding one representation and instantly encoding it into another.

TDT defines this information using a set of XML files, each referred to as a scheme. For example, the SGTIN-64 scheme defines how to decode, encode, and translate between various SGTIN-64 representations, such as binary and pure identity. For details about the EPCglobal TDT schema, see the EPCglobal Tag Data Translation specification.

A key feature of the TDT specification is its ability to define any EPC scheme using the same XML schema. This approach creates a standard way of defining EPC metadata that RFID applications can then use to write their parsers, encoders, and translators. When the application is written according to the TDT specification, it must be able to update its set of EPC tag schemes and modify its action according to the new metadata.

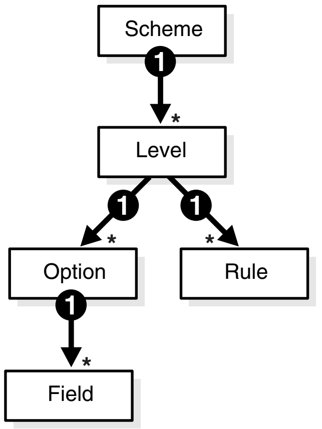
The Oracle metadata structure is similar, but not identical to the TDT standard. In order for Oracle to comply with the EPCglobal TDT specification, the Oracle RFID package must be able to ingest any TDT compatible scheme and seamlessly translate it into the generic Oracle defined metadata. See the `EPC_TO_ORACLE` Function in Table 17–4 for more information.

Reconstructing tag representation from fields, or in other words, encoding tag data into predefined representations is easily accomplished using the `MGD_ID.format` function. Likewise, the decoding of tag representations into `MGD_ID` objects and then encoding these objects into tag representations is also easily accomplished using the `MGDID.translate` function. See the `FORMAT` Member Function and the `TRANSLATE` Static Function in Table 17–3 for more information.

Because the EPCglobal TDT standard is powerful and highly extensible, the Oracle RFID standard metadata is a close relative of the TDT specification. See "Oracle Tag Data Translation Schema" on page 17-25 for the actual Oracle TDT XML schema. Developers can refer to this Oracle TDT XML schema to define their own tag structures.

Figure 17–2 shows the Oracle Tag Data Translation Markup Language Schema diagram.

Figure 17–2 Oracle Tag Data Translation Markup Language Schema



The top level element in a tag data translation xml is 'scheme'. Each scheme defines various tag encoding representations, or levels. SGTIN-64 and GID-96 are examples of tag encoding schemes, and BINARY or PURE_IDENTITY are examples of levels within these schemes. Each level has a set of options that define how to parse various representations into fields, and rules that define how to derive values for fields that require additional work, such as an external table lookup or the concatenation of other parsed out fields. See the EPCGlobal Tag Translator Specification for more information.

What is the Identity Code Package?

The Identity Code Package provides an extensible framework that supports the current RFID tags with the standard family of EPC bit encodings for the supported encoding types as well as new and evolving tag encodings that are not included in the current EPC standard.

The Identity Code Package defines the following object types:

- `MGD_ID` -- defines the following (see `MGD_ID` Object Type in Table 17–2 for more information):
  - Two attributes, `category_id` and `components`.
  - Four `MGD_ID` constructor functions for constructing identity code type objects to represent RFID tags.
  - A set of member subprograms for operating on these object types.

  "Using the Identity Code Package" on page 17-6 describes how to use these object types and member functions.

  "Identity Code Package Types" on page 17-18 and "DBMS_MGD_ID_UTL Package" on page 17-19 briefly describe the reference information for these object types along with a set of utility subprograms. See *Oracle Database PL/SQL Packages and Types Reference* for detailed reference information.
- `MGD_ID_COMPONENT` — defines two attributes, `comp_name`, which identifies the name of the component and `comp_value`, which identifies the components value.
- `MGD_ID_COMPONENT_VARRAY` — defines an array type that can store up to 128 elements of `MGD_IDCOMPONENT` type, which is used in two constructor functions for creating an identity code type object with a list of components.

The Identity Code Package supports EPC spec v1.1 by supplying the predefined `EPC_ENCODING_CATEGORY` encoding_category attribute definition with its bit-encoding structures for the supported encoding types. This information is stored as meta information in the supplied encoding metadata views, `MGD_USR_ID_CATEGORY`, `MGD_USR_ID_SCHEME`, the read-only views `MGD_ID_CATEGORY`, `MGD_ID_SCHEME`, and their underlying tables: `MGD_ID_CATEGORY_TAB`, `MGD_ID_SCHEME_TAB`, `MGD_ID_XML_VALIDATOR`. See the following sections and files for more information:

- "Electronic Product Code (EPC) Concepts" on page 17-22 describes the EPC spec v1.1 product code and its family of coding schemes.
- "Identity Code Metadata Tables and Views" on page 17-20 describes the structure of the identity code meta tables and views and how metadata are used by the Identity Code Package to interpret the various RFID tags.
- The `mgdmeta.sql` file describes the meta table data for the `EPC_ENCODING_CATEGORY` categories and each of its specific encoding schemes.

After storing many thousands of RFID tags into the column of `MGD_ID` column type of your user table, you can improve query performance by creating an index on this column. See the following sections for more information:

- "Building a Function-Based Index Using the Member Functions of the MGD_ID Column Type" on page 17-10 describes how to create a function based index or bitmap function based index using the member functions of the `MGD_ID` object type.

The Identity Code Package provides a utility package that consists of various utility subprograms. See the following section for more information:

- "Identity Code Package Types" on page 17-18 and "DBMS_MGD_ID_UTL Package" on page 17-19 describes each of the member subprograms. A proxy utility is used to set and remove proxy information. A metadata utility can be used to get a category ID, refresh a tag scheme for a category, remove a tag scheme for a category, and validate a tag scheme. A conversion utility is used to translate standard EPCglobal Tag Data Translation (TDT) files into Oracle TDT files.

The Identity Code Package is extensible and lets you create your own identity code types for any new or evolving RFID tags that you want to create. You can define your identity code types, `catagory_id` attribute values, and components structures for your own encoding types. See the following sections for more information:

- "Creating a New Category of Identity Codes" on page 17-14 describes how to create your own identity codes by first registering the new encoding category, and then registering the new schemes associated to the new encoding category.
- "Identity Code Metadata Tables and Views" on page 17-20 describes the structure of the identity code meta tables and views and how to register meta information by storing it in the supplied metadata tables and views.

Using the Identity Code Package

Topics:

- Storing RFID Tags in Oracle Database Using MGD_ID Object Type
- Creating Indexes on the MGD_ID Column Type
- Using MGD_ID Object Type Functions
- Defining a New Category of Identity Codes and Adding Encoding Schemes to an Existing Category

The examples in this chapter assume that the user has run the following set of statements before running the contents of each script:

```
CONNECT / AS SYSDBA;
CREATE USER MGDUSER IDENTIFIED BY MGDUSER;
GRANT CONNECT, RESOURCE TO MGDUSER;
CONNECT MGDUSER/MGDUSER;
SET SERVEROUTPUT ON;
```

Storing RFID Tags in Oracle Database Using MGD_ID Object Type

Topics:

- Creating a Table with MGD_ID Column Type and Storing EPC Tag Encodings in the Column
- Constructing MGD_ID Objects to Represent RFID Tags
- Inserting an MGD_ID Object into a Database Table
- Querying MGD_ID Column Type

Creating a Table with MGD_ID Column Type and Storing EPC Tag Encodings in the Column

You can create tables using `MGD_ID` as the column type to represent RFID tags, for example:

Example 1. Using the `MGD_ID` column type:

```
CREATE TABLE Warehouse_info (
             Code          MGD_ID,
             Arrival_time  TIMESTAMP,
             Location      VARCHAR2(256);
             ...);

SQL> describe warehouse_info;
Name                                      Null?    Type
----------------------------------------- -------- ----------------------------
CODE                                      NOT NULL MGDSYS.MGD_ID
ARRIVAL_TIME                                       TIMESTAMP(6)
LOCATION                                           VARCHAR2(256)
```

Constructing MGD_ID Objects to Represent RFID Tags

There are several ways to construct `MGD_ID` objects:

- Constructing an MGD_ID Object (SGTIN-64) Passing in the Category ID and a List of Components
- Constructing an MGD_ID object (SGTIN-64) and Passing in the Category ID, the Tag Identifier, and the List of Additional Required Parameters
- Constructing an MGD_ID object (SGTIN-64) and Passing in the Category Name, Category Version (if null, then the latest version will be used), and a List of Components
- Constructing an MGD_ID object (SGTIN-64) and Passing in the Category Name and Category Version, the Tag Identifier, and the List of Additional Required Parameters

Constructing an MGD_ID Object (SGTIN-64) Passing in the Category ID and a List of Components

If a RFID tag complies to the EPC standard, an `MGD_ID` object can be created using its category ID and a list of components. For example:

```
call DBMS_MGD_ID_UTL.set_proxy('www-proxy.us.oracle.com', '80');
call DBMS_MGD_ID_UTL.refresh_category('1');
select MGD_ID ('1',
               MGD_ID_COMPONENT_VARRAY(
               MGD_ID_COMPONENT('companyprefix','0037000'),
               MGD_ID_COMPONENT('itemref','030241'),
               MGD_ID_COMPONENT('serial','1041970'),
               MGD_ID_COMPONENT('schemes','SGTIN-64')
              )
             ) from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();

SQL> @constructor11.sql
.
.
.
MGD_ID ('1', MGD_ID_COMPONENT_VARRAY
        (MGD_ID_COMPONENT('companyprefix', '0037000'),
        MGD_ID_COMPONENT('itemref', '030241'),
        MGD_ID_COMPONENT('serial', '1041970'),
        MGD_ID_COMPONENT('schemes', 'SGTIN-64')))
.
.
.
```

Constructing an MGD_ID object (SGTIN-64) and Passing in the Category ID, the Tag Identifier, and the List of Additional Required Parameters Use this constructor when there is a list of additional parameters required to create the `MGD_ID` object. For example:

```
call DBMS_MGD_ID_UTL.set_proxy('www-proxy.us.oracle.com', '80');
call DBMS_MGD_ID_UTL.refresh_category('1');
select MGD_ID('1',
              'urn:epc:id:sgtin:0037000.030241.1041970',
              'filter=3;scheme=SGTIN-64') from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();


SQL> @constructor22.sql
.
.
.
MGD_ID('1', MGD_ID_COMPONENT_VARRAY(MGD_ID_COMPONENT('filter', '3'),
       MGD_ID_COMPONENT('schemes', 'SGTIN-64'),
       MGD_ID_COMPONENT('companyprefixlength', '7'),
       MGD_ID_COMPONENT('companyprefix', '0037000'),
       MGD_ID_COMPONENT('scheme', 'SGTIN-64'),
       MGD_ID_COMPONENT('serial', '1041970'),
       MGD_ID_COMPONENT('itemref', '030241')))
.
.
.
```

Constructing an MGD_ID object (SGTIN-64) and Passing in the Category Name, Category Version (if null, then the latest version will be used), and a List of Components Use this constructor when a category version must be specified along with a category ID and a list of components. For example:

```
call DBMS_MGD_ID_UTL.set_proxy('www-proxy.us.oracle.com', '80');
call DBMS_MGD_ID_UTL.refresh_category
  (DBMS_MGD_ID_UTL.get_category_id('EPC', NULL));
select MGD_ID('EPC', NULL,
              MGD_ID_COMPONENT_VARRAY(
              MGD_ID_COMPONENT('companyprefix','0037000'),
              MGD_ID_COMPONENT('itemref','030241'),
              MGD_ID_COMPONENT('serial','1041970'),
              MGD_ID_COMPONENT('schemes','SGTIN-64')
             )
            ) from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();

SQL> @constructor33.sql
.
.
.
MGD_ID('1', MGD_ID_COMPONENT_VARRAY
             (MGD_ID_COMPONENT('companyprefix', '0037000'),
              MGD_ID_COMPONENT('itemref', '030241'),
              MGD_ID_COMPONENT('serial', '1041970'),
              MGD_ID_COMPONENT('schemes', 'SGTIN-64')
             )
       )
.
.
.
```

Constructing an MGD_ID object (SGTIN-64) and Passing in the Category Name and Category Version, the Tag Identifier, and the List of Additional Required Parameters Use this constructor when the category version and an additional list of parameters is required.

```
call DBMS_MGD_ID_UTL.set_proxy('www-proxy.us.oracle.com', '80');
call DBMS_MGD_ID_UTL.refresh_category
  (DBMS_MGD_ID_UTL.get_category_id('EPC', NULL));
select MGD_ID('EPC', NULL,
              'urn:epc:id:sgtin:0037000.030241.1041970',
              'filter=3;scheme=SGTIN-64') from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();

SQL> @constructor44.sql
.
.
.
MGD_ID('1', MGD_ID_COMPONENT_VARRAY
       (MGD_ID_COMPONENT('filter', '3'),
        MGD_ID_COMPONENT('schemes', 'SGTIN-64'),
        MGD_ID_COMPONENT('companyprefixlength', '7'),
        MGD_ID_COMPONENT('companyprefix', '0037000'),
        MGD_ID_COMPONENT('scheme', 'SGTIN-64'),
        MGD_ID_COMPONENT('serial', '1041970'),
        MGD_ID_COMPONENT('itemref', '030241')
       )
      )
.
.
.
```

Inserting an MGD_ID Object into a Database Table

The following example shows how to populate the `WAREHOUSE_INFO` table by inserting each `MGD_ID` object into the table along with the additional column values.

```
call DBMS_MGD_ID_UTL.set_proxy('www-proxy.us.oracle.com', '80');

call DBMS_MGD_ID_UTL.refresh_category
  (DBMS_MGD_ID_UTL.get_category_id('EPC', NULL));

INSERT INTO WAREHOUSE_INFO (code, arrival_time, location)
   values (MGDSYS.MGD_ID ('EPC',
                          NULL,
                          'urn:epc:id:sgtin:0037000.030241.1041970',
                          null
                         ),
           SYSDATE,
           'SHELF_123');

INSERT INTO WAREHOUSE_INFO (code, arrival_time, location)
  values (MGDSYS.MGD_ID ('EPC',
                         NULL,
                         'urn:epc:id:sgtin:0037000.053021.1012353',
                         null
                        ),
          SYSDATE,
          'SHELF_456');
INSERT INTO WAREHOUSE_INFO (code, arrival_time, location)
  values (MGDSYS.MGD_ID ('EPC',
                         NULL,
                         'urn:epc:id:sgtin:0037000.020140.10174832',
                         null
                        ),
          SYSDATE,
          'SHELF_1034');

COMMITT;
call DBMS_MGD_ID_UTL.remove_proxy();
```

Querying MGD_ID Column Type

There are three ways to query on MGD_ID column type.

- Query the MGD_ID column type. Find all items with item reference 030241.

  ```
  SELECT location, wi.code.get_component('itemref') as itemref,
                   wi.code.get_component('serial') as serial
  FROM warehouse_info wi WHERE wi.code.get_component('itemref') = '030241';

  LOCATION       |ITEMREF   |SERIAL
  ---------------|----------|----------
  SHELF_123      |030241    |1041970
  ```

- Query using the member functions of the MGD_ID object type. Select the pure identity representations of all RFID tags in the table.

  ```
  SELECT wi.code.format(null,'PURE_IDENTITY')
     as PURE_IDENTITY FROM warehouse_info wi;

  PURE_IDENTITY
  --------------------------------------------------------------------------------
  urn:epc:id:sgtin:0037000.030241.1041970
  urn:epc:id:gid:0037000.053021.1012353
  urn:epc:id:sgtin:0037000.020140.10174832
  ```

  See "Using the get_component Function with the MGD_ID Object" on page 17-11 for more information and see Table 17–3 for a list of member functions.

Creating Indexes on the MGD_ID Column Type

This section contains the following section:

- Building a Function-Based Index Using the Member Functions of the MGD_ID Column Type

Building a Function-Based Index Using the Member Functions of the MGD_ID Column Type

You can improve the performance of queries based on a certain component of the RFID tags by creating a function-based index that uses the get_component member function or its variation convenience functions. For example:

```
CREATE INDEX warehouseinfo_idx2
  on warehouse_info(code.get_component('itemref'));
```

You can also improve the performance of queries based on a certain component of the RFID tags by creating a bitmap function based index that uses the `get_component` member function or its variation convenience functions. For example:

```
CREATE BITMAP INDEX warehouseinfo_idx3
  on warehouse_info(code.get_component('serial'));
```

Using MGD_ID Object Type Functions

The `MGD_ID` object type contain member subprograms that operate on these object types. See Table 17–2 for `MGD_ID_COMPONENT`, `MGD_ID_COMPONENT_VARRAY`, `MGD_ID` object type reference information. See the `mgdtyp.sql` file for the `MGD_ID` object type definition and its member subprograms. This section contains the following sections:

- Using the get_component Function with the MGD_ID Object
- Parsing Tag Data from Standard Representations
- Reconstructing Tag Representations from Fields
- Translating Between Tag Representations

Using the get_component Function with the MGD_ID Object

The `get_component` function is defined as follows:

```
MEMBER FUNCTION get_component(component_name IN VARCHAR2)
   RETURN VARCHAR2 DETERMINISTIC,
```

Each component in a identity code has a name. It is defined when the code type is registered. See "Defining a New Category of Identity Codes and Adding Encoding Schemes to an Existing Category" on page 17-13 for more information on how to create a new identity code type.

The `get_component` function takes the name of the component, `component_name` as a parameter, uses the metadata registered in the metadata table to analyze the identity code, and returns the component with the name `component_name`.

The `get_component` function can be used in a SQL query. For example, find the current location of the coded item for the component named `itemref`; or, in other words find all items with the item reference of 03024. Because the code tag has encoded the "itemref" as one of the components, you can use the following SQL query:

```
SELECT location,
       w.code.get_component('itemref') as itemref,
       w.code.get_component('serial')  as serial
FROM   warehouse_info w
       WHERE  w.code.get_component('itemref')  = '030241';

LOCATION       |ITEMREF   |SERIAL
---------------|----------|----------
SHELF_123      |030241    |1041970
```

See Table 17–3 for a list of other member functions.

Parsing Tag Data from Standard Representations

RFID readers read the bit strings stored in the tags. The tag data together with other information such as the reader ID and the timestamp, first go through an edge server to be processed, normalized, and preliminarily filtered. Then, in many application scenarios, the information needs to be persistently stored and later on be retrieved.

The Oracle Database is capable of understanding the code structures representations of various EPC tags as described in Table 17–1 because these code representation schemes defined in the EPC Standard are pre-registered. This gives the Oracle Database the ability to understand all the EPC code schemes and parse various tag representations into fields. Users can also register their own coding structures for the identity codes that use other encoding technologies. In this way the system is extensible.

As mentioned in "Identity Concepts" on page 17-1, each of the EPCGlobal tag schemes (GID-96, SGTIN-64, SGTIN-96, and so forth) has various representations with the most often used ones being BINARY, TAG_URI, and PURE_IDENTITY.

Some of these representations contain all the information about the tag (BINARY and TAG_URI), while representations contain only partial information (PURE_IDENTITY). It is therefore possible to translate a tag from it's TAG_URI to it's PURE_IDENTITY representation, but it is not possible to translate in the other direction (PURE_IDENTITY to TAG_URI) without supplying additional information, namely the filter value.

One of the MGD_ID constructors takes in four fields, the category name (such as EPC), the category version, the tag identifier (for EPC, the identifier must be in one of the representations previously described), and a parameter list for any additional parameters that may be required in order to parse the tag representation. For example, the following code creates an MGD_ID object from its BINARY representation.

```
SELECT MGD_ID
   ('EPC',
    null,
    '1001100000000000001000001110110001000010000011111110011000110010',
    null
   )
   AS NEW_RFID_CODE FROM DUAL;

NEW_RFID_CODE(CATEGORY_ID, COMPONENTS(NAME, VALUE))
--------------------------------------------------------------------------------
MGD_ID ('1',
        MGD_ID_COMPONENT_VARRAY(MGD_ID_COMPONENT('filter', '3'),
        MGD_ID_COMPONENT('schemes', 'SGTIN-64'),
        MGD_ID_COMPONENT('companyprefixlength', '7'),
        MGD_ID_COMPONENT('companyprefix', '0037000'),
        MGD_ID_COMPONENT('companyprefixindex', '1'),
        MGD_ID_COMPONENT('serial', '1041970'),
        MGD_ID_COMPONENT('itemref', '030241')
       )
      )
```

For example, an identical object can be created if the call is done with the TAG_URI representation of the tag as follows with the addition of the value of the filter value:

```
SELECT MGD_ID ('EPC',
                null,
                'urn:epc:tag:sgtin-64:3.0037000.030241.1041970',
                null
              )
  as NEW_RFID_CODE FROM DUAL;

NEW_RFID_CODE(CATEGORY_ID, COMPONENTS(NAME, VALUE))
--------------------------------------------------------------------------------
MGD_ID ('1',
        MGD_ID_COMPONENT_VARRAY (
         ( MGD_ID_COMPONENT('filter', '3'),
           MGD_ID_COMPONENT('schemes', 'SGTIN-64'),
           MGD_ID_COMPONENT('companyprefixlength', '7'),
           MGD_ID_COMPONENT('companyprefix', '0037000'),
           MGD_ID_COMPONENT('serial', '1041970'),
           MGD_ID_COMPONENT('itemref', '030241')
         )
       )
```

Reconstructing Tag Representations from Fields

Another useful feature of the Identity Code package is the ability to encode tag data into predefined representations. For example, a warehouse wants to send certain inventory to a retailer, but first it wants to send an invoice that tells the retailer what inventory to expect. The invoice can be a list of pure identity URIs that the warehouse intends to send. If all the inventory in the `WAREHOUSE_INFO` table is to be sent, the following example constructs the desired URIs.

```
SELECT wi.code.format (null,'PURE_IDENTITY')
  as PURE_IDENTITY FROM warehouse_info wi;

PURE_IDENTITY
--------------------------------------------------------------------------------
urn:epc:id:sgtin:0037000.030241.1041970
urn:epc:id:gid:0037000.053021.1012353
urn:epc:id:sgtin:0037000.020140.10174832
```

Translating Between Tag Representations

The Identity Code package can decode tag representations into MGD_ID objects and encode these objects into tag representations. These two steps can be combined into one step using the `MGD_ID.translate` function. Static translation allows for the conversion of an RFID tag from one representation to another. For example:

```
SELECT MGD_ID.translate ('EPC',
                         null,
                         'urn:epc:id:sgtin:0037000.030241.1041970',
                         'filter=3;scheme=SGTIN-64',
                         'BINARY'
                        )
  as BINARY FROM DUAL;

BINARY
--------------------------------------------------------------------------------
10011000000000000001000001110110001000010000011111110011000110010
```

In this example, the binary representation contains more information than the pure identity representation. Specifically, it also contains the filter value and in this case the scheme value must also be specified to distinguish SGTIN-64 from SGTIN-96. Thus, the function call must provide the missing filter parameter information and specify the scheme name in order for translation call to succeed.

Defining a New Category of Identity Codes and Adding Encoding Schemes to an Existing Category

This section describes the following:

- Creating a New Category of Identity Codes
- Adding Two New Metadata Schemes to a Newly Created Category

Creating a New Category of Identity Codes

Because the EPCglobal TDT standard is powerful and highly extensible, the Oracle RFID standard metadata is a close relative of the TDT specification. Thus, the Identity Code package is extensible and lets you create your own categories and tag structures using generic metadata. To do this, use the `DBMS_MGD_ID_UTIL.create_category` function to create a new category of identity codes.

For example, suppose you want to create a category called MGD_SAMPLE_CATEGORY, which will have two types of tags, a CONTRACTOR_TAG and an EMPLOYEE_TAG. This new category and its two new metadata schemes might be used within a company that needs to grant different access privileges to people who are full time employees from those who are contractors, and thus require that their security software be able to identify quickly between the two badge types at an RFID reader. The following script creates a new category named 'MGD_SAMPLE_CATEGORY', with a 1.0 category version, having an agency name as Oracle, with a URI as `http://www.oracle.com/mgd/sample`. See "Adding Two New Metadata Schemes to a Newly Created Category" on page 17-14 for an example.

Adding Two New Metadata Schemes to a Newly Created Category

Next, create an `CONTRACTOR_TAG` metadata scheme such as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<TagDataTranslation version="0.04" date="2005-04-18T16:05:00Z"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema"
                    xmlns="oracle.mgd.idcode">
 <scheme name="CONTRACTOR_TAG" optionKey="1" xmlns="">
  <level type="URI" prefixMatch="mycompany.contractor.">
   <option optionKey="1" pattern="mycompany.contractor.([0-9]*).([0-9]*)"
           grammar="''mycompany.contractor.'' contractorID ''.'' divisionID">
    <field seq="1" characterSet="[0-9]*" name="contractorID"/>
    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
   </option>
  </level>
  <level type="BINARY" prefixMatch="11">
   <option optionKey="1" pattern="11([01]{7})([01]{6})"
           grammar="''11'' contractorID divisionID ">
    <field seq="1" characterSet="[01]*" name="contractorID"/>
    <field seq="2" characterSet="[01]*" name="divisionID"/>
   </option>
  </level>
 </scheme>
</TagDataTranslation>
```

The `CONTRACTOR_TAG` scheme contains two encoding levels, or ways in which the tag can be represented. The first level is URI and the second level is BINARY. The URI representation starts with the prefix "`mycompany.contractor.`" and is then followed by two numeric fields separated by a period. The names of the two fields are `contractorID` and `divisionID`. The pattern field in the option tag defines the parsing structure of the tag URI representation, and the grammar field defines how to reconstruct the URI representation. The BINARY representation can be understood in a similar fashion. This representation starts with the prefix "01" and is then followed by the same two fields, `contractorID` and `divisionID`, this time, in their respective binary formats. Given this XML metadata structure, contractor tags can now be decoded from their URI and BINARY representations and the resulting fields can be re-encoded into one of these representations.

The EMPLOYEE_TAG scheme is defined in a similar fashion and is shown as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<TagDataTranslation version="0.04" date="2005-04-18T16:05:00Z"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema"
                    xmlns="oracle.mgd.idcode">
 <scheme name="EMPLOYEE_TAG" optionKey="1" xmlns="">
  <level type="URI" prefixMatch="mycompany.employee.">
   <option optionKey="1" pattern="mycompany.employee.([0-9]*).([0-9]*)"
           grammar="''mycompany.employee.'' employeeID ''.'' divisionID">
    <field seq="1" characterSet="[0-9]*" name="employeeID"/>
    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
   </option>
  </level>
  <level type="BINARY" prefixMatch="01">
   <option optionKey="1" pattern="01([01]{7})([01]{6})"
           grammar="''01'' employeeID divisionID ">
    <field seq="1" characterSet="[01]*" name="employeeID"/>
    <field seq="2" characterSet="[01]*" name="divisionID"/>
   </option>
  </level>
 </scheme>
</TagDataTranslation>;
```

To add these schemes to the category ID previously created, use the `DBMS_MGD_ID_UTIL.add_scheme` function.

The following script creates the `MGD_SAMPLE_CATEGORY` category, adds a contractor scheme and an employee scheme to the `MGD_SAMPLE_CATEGORY` category, validates the `MGD_SAMPLE_CATEGORY` scheme, tests the tag translation of the contractor scheme and the employee scheme, then removes the contractor scheme, tests the tag translation of the contractor scheme and this returns the expected exception for the removed contractor scheme, tests the tag translation of the employee scheme and this returns the expected values, then removes the `MGD_SAMPLE_CATEGORY` category

```
--contents of add_scheme2.sql
SET LINESIZE 160
CALL DBMS_MGD_ID_UTL.set_proxy('www-proxy.us.oracle.com', '80');
---------------------------------------------------------------------
---CREATE CATEGORY, ADD_SCHEME, REMOVE_SCHEME, REMOVE_CATEGORY-------
---------------------------------------------------------------------
DECLARE
  amt          NUMBER;
  buf          VARCHAR2(32767);
  pos          NUMBER;
  tdt_xml      CLOB;
  validate_tdtxml VARCHAR2(1042);
  category_id  VARCHAR2(256);
BEGIN
  -- remove the testing category if already existed
  DBMS_MGD_ID_UTL.remove_category('MGD_SAMPLE_CATEGORY', '1.0');
  -- create the testing category 'MGD_SAMPLE_CATEGORY', version 1.0
  category_id := DBMS_MGD_ID_UTL.CREATE_CATEGORY('MGD_SAMPLE_CATEGORY', '1.0', 'Oracle',
'http://www.oracle.com/mgd/sample');
  -- add contractor scheme to the category
  DBMS_LOB.CREATETEMPORARY(tdt_xml, true);
  DBMS_LOB.OPEN(tdt_xml, DBMS_LOB.LOB_READWRITE);

  buf := '<?xml version="1.0" encoding="UTF-8"?>
<TagDataTranslation version="0.04" date="2005-04-18T16:05:00Z"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema"
                    xmlns="oracle.mgd.idcode">
```

```
 <scheme name="CONTRACTOR_TAG" optionKey="1" xmlns="">
  <level type="URI" prefixMatch="mycompany.contractor.">
   <option optionKey="1" pattern="mycompany.contractor.([0-9]*).([0-9]*)"
           grammar="''mycompany.contractor.'' contractorID ''.'' divisionID">
    <field seq="1" characterSet="[0-9]*" name="contractorID"/>
    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
   </option>
  </level>
  <level type="BINARY" prefixMatch="11">
   <option optionKey="1" pattern="11([01]{7})([01]{6})"
           grammar="''11'' contractorID divisionID ">
    <field seq="1" characterSet="[01]*" name="contractorID"/>
    <field seq="2" characterSet="[01]*" name="divisionID"/>
   </option>
  </level>
 </scheme>
</TagDataTranslation>';

  amt := length(buf);
  pos := 1;
  DBMS_LOB.WRITE(tdt_xml, amt, pos, buf);
  DBMS_LOB.CLOSE(tdt_xml);

  DBMS_MGD_ID_UTL.ADD_SCHEME(category_id, tdt_xml);

  -- add employee scheme to the category
  DBMS_LOB.CREATETEMPORARY(tdt_xml, true);
  DBMS_LOB.OPEN(tdt_xml, DBMS_LOB.LOB_READWRITE);

  buf := '<?xml version="1.0" encoding="UTF-8"?>
<TagDataTranslation version="0.04" date="2005-04-18T16:05:00Z"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema"
                    xmlns="oracle.mgd.idcode">
 <scheme name="EMPLOYEE_TAG" optionKey="1" xmlns="">
  <level type="URI" prefixMatch="mycompany.employee.">
   <option optionKey="1" pattern="mycompany.employee.([0-9]*).([0-9]*)"
           grammar="''mycompany.employee.'' employeeID ''.'' divisionID">
    <field seq="1" characterSet="[0-9]*" name="employeeID"/>
    <field seq="2" characterSet="[0-9]*" name="divisionID"/>
   </option>
  </level>
  <level type="BINARY" prefixMatch="01">
   <option optionKey="1" pattern="01([01]{7})([01]{6})"
           grammar="''01'' employeeID divisionID ">
    <field seq="1" characterSet="[01]*" name="employeeID"/>
    <field seq="2" characterSet="[01]*" name="divisionID"/>
   </option>
  </level>
 </scheme>
</TagDataTranslation>';

  amt := length(buf);
  pos := 1;
  DBMS_LOB.WRITE(tdt_xml, amt, pos, buf);
  DBMS_LOB.CLOSE(tdt_xml);
  DBMS_MGD_ID_UTL.ADD_SCHEME(category_id, tdt_xml);

  -- validate the scheme
  dbms_output.put_line('Validate the MGD_SAMPLE_CATEGORY Scheme');
  validate_tdtxml := DBMS_MGD_ID_UTL.validate_scheme(tdt_xml);
```

```
dbms_output.put_line(validate_tdtxml);
dbms_output.put_line('Length of scheme xml is: '||DBMS_LOB.GETLENGTH(tdt_xml));

-- test tag translation of contractor scheme
dbms_output.put_line(
  mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                   'mycompany.contractor.123.45',
                   NULL, 'BINARY'));

dbms_output.put_line(
  mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                   '111111011101101',
                   NULL, 'URI'));

-- test tag translation of employee scheme
dbms_output.put_line(
  mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                   'mycompany.employee.123.45',
                   NULL, 'BINARY'));

dbms_output.put_line(
  mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                   '011111011101101',
                   NULL, 'URI'));

DBMS_MGD_ID_UTL.REMOVE_SCHEME(category_id, 'CONTRACTOR_TAG');

-- Test tag translation of contractor scheme. Doesn't work any more.
BEGIN
  dbms_output.put_line(
    mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                     'mycompany.contractor.123.45',
                     NULL, 'BINARY'));

  dbms_output.put_line(
    mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                     '111111011101101',
                     NULL, 'URI'));
EXCEPTION
  WHEN others THEN
    dbms_output.put_line('Contractor tag translation failed: '||SQLERRM);
END;

-- Test tag translation of employee scheme. Still works.
BEGIN
  dbms_output.put_line(
    mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                     'mycompany.employee.123.45',
                     NULL, 'BINARY'));
  dbms_output.put_line(
    mgd_id.translate('MGD_SAMPLE_CATEGORY', NULL,
                     '011111011101101',
                     NULL, 'URI'));
EXCEPTION
  WHEN others THEN
    dbms_output.put_line('Employee tag translation failed: '||SQLERRM);
END;

-- remove the testing category, which also removes all the associated schemes
DBMS_MGD_ID_UTL.remove_category('MGD_SAMPLE_CATEGORY', '1.0');
```

```
END;
/
SHOW ERRORS;
call DBMS_MGD_ID_UTL.remove_proxy();

SQL> @add_scheme3.sql
.
.
.
Validate the MGD_SAMPLE_CATEGORY Scheme
EMPLOYEE_TAG;URI,BINARY;divisionID,employeeID
Length of scheme xml is: 933
111111011101101
mycompany.contractor.123.45
011111011101101
mycompany.employee.123.45
Contractor tag translation failed: ORA-55203: Tag data translation level not found
ORA-06512: at "MGDSYS.DBMS_MGD_ID_UTL", line 54
ORA-06512: at "MGDSYS.MGD_ID", line 242
ORA-29532: Java call terminated by uncaught Java
exception: oracle.mgd.idcode.exceptions.TDTLevelNotFound: Matching level not
found for any configured scheme
011111011101101
mycompany.employee.123.45
.
.
.
```

Identity Code Package Types

Table 17–2 describes the Identity Code Package object types.

Table 17–2 Identity Code Package Object Types

Object Type Name	Description
`MGD_ID_COMPONENT` Object Type	A datatype that specifies the name and value pair attributes that define a component.
`MGD_ID_COMPONENT_VARRAY` Object Type	A datatype that specifies a list of up to 128 components as name-value attribute pairs used in two constructor functions for creating an identity code type object.
`MGD_ID` Object Type	Represents an identity code type that specifies the category identifier for the code category for this identity code and its list of components.

Table 17–3 describes the subprograms in the `MGD_ID` object type.

All the values and names passed to the subprograms defined in the `MGD_ID` object type are case-insensitive unless otherwise noted. To preserve case, enclose values in double quotation marks.

Table 17–3 MGD_ID Object Type Subprograms

Subprogram	Description
`MGD_ID` Constructor Function	Creates an identity code type object, `MGD_ID`, and returns self as a result.

Table 17–3 (Cont.) MGD_ID Object Type Subprograms

Subprogram	Description
FORMAT Member Function	Returns a representation of an identity code given an MGD_ID component.
GET_COMPONENT Member Function	Returns the value of an MGD_ID component.
TO_STRING Member Function	Concatenates the category_id parameter value with the components name-value attribute pair.
TRANSLATE Static Function	Translates one MGD_ID representation of an identity code into a different MGD_ID representation.

DBMS_MGD_ID_UTL Package

Table 17–4 describes the Utility subprograms in the DBMS_MGD_ID_UTL package.

All the values and names passed to the subprograms defined in the MGD_ID object type are case-insensitive unless otherwise noted. To preserve case, enclose values in double quotation marks.

Table 17–4 DBMS_MGD_ID_UTL Package Utility Subprograms

Subprogram	Description
ADD_SCHEME Procedure	Adds a tag data translation scheme to an existing category.
CREATE_CATEGORY Function	Creates a new category or a new version of a category.
EPC_TO_ORACLE Function	Converts the EPCglobal tag data translation (TDT) XML to Oracle tag data translation XML.
GET_CATEGORY_ID Function	Returns the category ID given the category name and the category version.
GET_COMPONENTS Function	Returns all relevant separated component names separated by semicolon (';') for the specified scheme.
GET_ENCODINGS Function	Returns a list of semicolon (';') separated encodings (formats) for the specified scheme.
GET_JAVA_LOGGING_LEVEL Function	Returns an integer representing the current Java trace logging level.
GET_PLSQL_LOGGING_LEVEL Function	Returns an integer representing the current PL/SQL trace logging level.
GET_SCHEME_NAMES Function	Returns a list of semicolon (';') separated scheme names for the specified category.
GET_TDT_XML Function	Returns the Oracle tag data translation XML for the specified scheme.
GET_VALIDATOR Function	Returns the Oracle tag data translation schema.
REFRESH_CATEGORY Function	Refreshes the metadata information on the Java stack for the specified category.
REMOVE_CATEORY Function	Removes a category including all the related TDT XML.
REMOVE_PROXY Procedure	Unsets the host and port of the proxy server.
REMOVE_SCHEME Procedure	Removes the tag scheme for a category.
SET_JAVA_LOGGING_LEVEL Procedure	Sets the Java logging level.

Table 17–4 (Cont.) DBMS_MGD_ID_UTL Package Utility Subprograms

Subprogram	Description
`SET_PLSQL_LOGGING_LEVEL` Procedure	Sets the PL/SQL tracing logging level.
`SET_PROXY` Procedure	Sets the host and port of the proxy server for Internet access.
`VALIDATE_SCHEME` Function	Validates the input tag data translation XML against the Oracle tag data translation schema.

Identity Code Metadata Tables and Views

This section describes the structure of identity code metadata tables and views and explains how the metadata are used by the Identity Code Package to interpret the various RFID tags. The creation of these meta tables, views, and triggers will be done automatically during the Identity Code Package installation.

Encoding metadata views are used to store encoding categories and schemes. Application developers can insert the meta information of their own identity codes into these views. The `MGD_ID` object type is designed to understand the new encodings as long as the metadata for the new encodings are stored in the meta tables. If an application developer only uses the encodings defined in the EPC specification v1.1, the developer does not have to worry about the meta tables because product codes specified in EPC spec v1.1 are predefined.

There are two encoding metadata views.

- `user_mgd_id_category` — this view is used to store the encoding category information defined by the session user.
- `user_mgd_id_scheme` — this view is used to store the encoding type information defined by the session user.

In addition, the following read-only views are defined for a user to query the system predefined encoding metadata as well as the metadata defined by the user.

- `mgd_id_category` — this view is used to query the encoding category information defined by the system or the session user
- `mgd_id_scheme` — this view is used to query the encoding type information defined by the system or the session user.

The underlying metadata tables for the preceding views are:

- `mgd_id_xml_validator`
- `mgd_id_category_tab`
- `mgd_id_scheme_tab`

Users other than the Identity Code Package system users cannot operate on these tables. Users must not use the metadata tables directly. They must use the read only views and the metadata functions described in the `DBMS_MGD_ID_UTL` package.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_MGD_ID_UTL` package

Metadata View Definitions

Table 17–5, Table 17–6, Table 17–7, and Table 17–8 describe the metadata view definitions for the MGD_ID_CATEGORY, USER_ID_CATEGORY, MGD_ID_SCHME, and USER_MGD_ID_SCHME respectively as defined in the `mgdview.sql` file.

Table 17–5 Definition and Description of the MGD_ID_CATEGORY Metadata View

Column Name	Data Type	Description
`CATEGORY_ID`	`NUMBER(4)`	Category identifier
`CATEGORY_NAME`	`VARCHAR2(256)`	Category name
`AGENCY`	`VARCHAR2(256)`	Organization that defined the category
`VERSION`	`VARCHAR2(256)`	Category version
`URI`	`VARCHAR2(256)`	URI that describes the category

Table 17–6 Definition and Description of the USER_MGD_ID_CATEGORY Metadata View

Column Name	Data Type	Description
`CATEGORY_ID`	`NUMBER(4)`	Category identifier
`CATEGORY_NAME`	`VARCHAR2(256)`	Category name
`AGENCY`	`VARCHAR2(256)`	Organization that defined the category
`VERSION`	`VARCHAR2(256)`	Category version
`URI`	`VARCHAR2(256)`	URI that describes the category

Table 17–7 Definition and Description of the MGD_ID_SCHEME Metadata View

Column Name	Data Type	Description
`CATEGORY_ID`	`NUMBER(4)`	Category identifier
`TYPE_NAME`	`VARCHAR2(256)`	Encoding scheme name, for example, SGTIN-96, GID-96, and so forth
`TDT_XML`	`CLOB`	Tag data translation XML for this encoding scheme
`ENCODINGS`	`VARCHAR2(256)`	Encodings separated by a comma (,), for example, `LEGACY`, `TAG_ENCODING`, `PURE_IDENTITY`, `BINARY` (for SGTIN-96)
`COMPONENTS`	`VARCHAR2(1024)`	Relevant component names, extracted from each level and then combined. Each is separated by a comma (,). For example, objectclass, generalmanager, serial (for GID-96)

Table 17–8 Definition and Description of the USER_MGD_ID_SCHEME Metadata View

Column Name	Data Type	Description
`CATEGORY_ID`	`NUMBER(4)`	Category identifier
`TYPE_NAME`	`VARCHAR2(256)`	Encoding scheme name, for example, SGTIN-96, GID-96, and so forth
`TDT_XML`	`CLOB`	Tag data translation XML for this encoding scheme

Table 17–8 (Cont.) Definition and Description of the USER_MGD_ID_SCHEME Metadata

Column Name	Data Type	Description
`ENCODINGS`	`VARCHAR2(256)`	Encodings separated by a comma (,), for example, `LEGACY`, `TAG_ENCODING`, `PURE_IDENTITY`, `BINARY` (for SGTIN-96)
`COMPONENTS`	`VARCHAR2(1024)`	Relevant component names, extracted from each level and then combined. Each is separated by a comma (,). For example, objectclass, generalmanager, serial (for GID-96)

Electronic Product Code (EPC) Concepts

This section describes the following topics:

- RFID Technology and EPC v1.1 Coding Schemes
- Product Code Concepts and Their Current Use

RFID Technology and EPC v1.1 Coding Schemes

Radio Frequency Identification (RFID) technology continues to gain momentum with suppliers, distributors, manufacturers, and retailers for its ability to eliminate line-of-site processes and automate critical supply chain transactions. Electronic Product Code (EPC), an identification scheme for universally identifying objects using RFID tags and other means, is gaining widespread acceptance as an emerging standard. Its capabilities will enable companies to reduce warehouse and distribution costs through improved inventory control and extended supply chain visibility.

The standardized EPC Identifier is a metacoding scheme designed to support the needs of various industries. As a result, the EPC represents a family of coding schemes and a means to make them unique across all possible EPC-compliant tags. EPC Version 1.1 includes the following specific coding schemes:

- General Identifier (GID)
- Serialized version of the EAN.UCC Global Trade Item Number (GTIN)
- EAN.UCC Serial Shipping Container Code (SSCC)
- EAN.UCC Global Location Number (GLN)
- EAN.UCC Global Returnable Asset Identifier (GRAI)
- EAN.UCC Global Individual Asset Identifier (GIAI)

RFID applications require the storage of a large volume of EPC data into a database. The efficient use of EPC data also requires that the database recognizes the different coding schemes of EPC data.

EPC is an emerging standard. It does not cover all the numbering schemes used in the various industries and is itself still evolving (the changes from EPC version 1.0 to EPC version 1.1 are significant).

Identity Code Package empowers the Oracle Database with the knowledge to recognize EPC coding schemes. It makes the Oracle Database a database system that not only provides efficient storage and component level retrieval for EPC data, but also has the built-in features to support EPC data encoding and decoding, as well as conversion between bit encoding and URI encoding.

Identity Code Package provides an extensible framework that allows developers to define their own coding schemes that are not included in the EPC standard. This extensibility feature also makes the Oracle Database adaptable to the evolving future EPC standard.

This chapter describes the requirement of storing, retrieving, encoding and decoding various product codes, including EPC, in an Oracle Database and shows how the Identity Code Package solution meets all these requirements by providing new data types, metadata tables, and PL/SQL packages for these purposes.

Product Code Concepts and Their Current Use

This section describes the following product codes:

- Electronic Product Code (EPC)
- Global Trade Identification Number (GTIN) and Serializable Global Trade Identification Number (SGTIN)
- Serial Shipping Container Code (SSCC)
- Global Location Number (GLN) and Serializable Global Location Number (SGLN)
- Global Returnable Asset Identifier (GRAI)
- Global Individual Asset Identifier (GIAI)
- RFID EPC Network

Electronic Product Code (EPC)

The Electronic Product Code™ (EPC™) is an identification scheme for universally identifying physical objects using Radio Frequency Identification (RFID) tags and other means. The standardized EPC data consists of an EPC (or EPC Identifier) that uniquely identifies an individual object, as well as an optional Filter Value when judged to be necessary to enable effective and efficient reading of the EPC tags. In addition to this standardized data, certain classes of EPC tags will allow user-defined data.

The EPC Identifier is a meta-coding scheme designed to support the needs of various industries by accommodating both existing coding schemes where possible and defining new schemes where necessary. The various coding schemes are referred to as Domain Identifiers, to indicate that they provide object identification within certain domains such as a particular industry or group of industries. As such, EPC represents a family of coding schemes (or "namespaces") and a means to make them unique across all possible EPC-compliant tags.

The EPCGlobal EPC Data Standards Version 1.1 defines the abstract content of the Electronic Product Code, and its concrete realization in the form of RFID tags, Internet URIs, and other representations. In EPC Version 1.1, the specific coding schemes include a General Identifier (GID), a serialized version of the EAN.UCC Global Trade Item Number (GTIN®), the EAN.UCC Serial Shipping Container Code (SSCC®), the EAN.UCC Global Location Number (GLN®), the EAN.UCC Global Returnable Asset Identifier (GRAI®), and the EAN.UCC Global Individual Asset Identifier (GIAI®).

EPC Pure Identity The EPC pure identity is the identity associated with a specific physical or logical entity, independent of any particular encoding vehicle such as an RF tag, bar code or database field. As such, a pure identity is an abstract name or number used to identify an entity. A pure identity consists of the information required to uniquely identify a specific entity, and no more.

EPC Encoding EPC encoding is a pure identity, together with additional information such as filter value, rendered into a specific syntax (typically consisting of value fields of specific sizes). A given pure identity may have a number of possible encodings, such as a Barcode Encoding, various Tag Encodings, and various URI Encodings. Encodings may also incorporate additional data besides the identity (such as the Filter Value used in some encodings), in which case the encoding scheme specifies what additional data it can hold.

For example, the Serial Shipping Container Code (SSCC) format as defined by the EAN.UCC System is an example of a pure identity. An SSCC encoded into the EPC-SSCC 96-bit format is an example of an encoding.

EPC Tag Bit-Level Encoding EPC encoding on a tag is a string of bits, consisting of a tiered, variable length header followed by a series of numeric fields whose overall length, structure, and function are completely determined by the header value.

EPC Identity URI The EPC identity URI is a representation of a pure identity as a Uniform Resource Identifier (URI).

EPC Tag URI Encoding The EPC tag URI encoding represents a specific EPC tag bit-level encoding, for example, urn:epc:tag:sgtin-64:3.0652642.800031.400.

EPC Encoding Procedure The EPC encoding procedure is used to generate an EPC tag bit-level encoding using various information.

EPC Decoding Procedure The EPC decoding procedure is used to convert an EPC tag bit-level encoding to an EAN.UCC code.

Global Trade Identification Number (GTIN) and Serializable Global Trade Identification Number (SGTIN)

A Global Trade Identification Number (GTIN) is used for the unique identification of trade items worldwide within the EAN.UCC system. The Serialized Global Trade Identification Number (SGTIN) is a new identity type in EPC standard version1.1. It is based on the EAN.UCC GTIN code defined in the General EAN.UCC Specifications [GenSpec5.0]. A GTIN identifies a particular class of object, such as a particular kind of product or SKU. The combination of GTIN and a unique serial number is called a Serialized GTIN (SGTIN).

Serial Shipping Container Code (SSCC)

The Serial Shipping Container Code (SSCC) is defined by the General EAN.UCC Specifications [GenSpec5.0]. The unique identification of logistics units is achieved in the EAN.UCC system by the use of the SSCC. The SSCC is intended for assignment to individual objects.

Global Location Number (GLN) and Serializable Global Location Number (SGLN)

The Global Location Number (GLN) is defined by the General EAN.UCC Specifications [GenSpec5.0]. A GLN can represent either a discrete, unique physical location such as a dock door or a warehouse slot, or an aggregate physical location such as an entire warehouse. In addition, a GLN can represent a logical entity such as an organization that performs a business function (for example, placing an order). The combination of GLN and a unique serial number is called a Serialized GLN (SGLN). However, until the EAN.UCC community determines the appropriate way to extend GLN, the serial number field is reserved and must not be used.

Global Returnable Asset Identifier (GRAI)

A returnable asset is a reusable package or transport equipment of a certain value. Global Returnable Asset Identifier is (GRAI) is defined by the General EAN.UCC Specifications [GenSpec5.0] for the unique identification of a returnable asset.

Global Individual Asset Identifier (GIAI)

The Global Individual Asset Identifier (GIAI) is defined by the General EAN.UCC Specifications [GenSpec5.0]. Unlike the GTIN, the GIAI is already intended for assignment to individual objects. Global Individual Asset Identifier (GIAI) is used to uniquely identify an entity that is part of the fixed inventory of a company. The GIAI can be used to identify any fixed asset of an organization.

RFID EPC Network

The RFID EPC network is used to identify, track and locate assets. Physical objects are identified by a unique RFID enabled EPC.

Oracle Tag Data Translation Schema

The Oracle Tag Data Translation Schema is closely related to the EPCglobal TDT schema, however it is not exact. The Oracle TDT is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="oracle.mgd.idcode"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:tdt="oracle.mgd.idcode" elementFormDefault="qualified"
       attributeFormDefault="unqualified" version="1.0">

 <xsd:simpleType name="InputFormatList">
  <xsd:restriction base="xsd:string">
   <xsd:enumeration value="BINARY"/>
   <xsd:enumeration value="STRING"/>
  </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="LevelTypeList">
  <xsd:restriction base="xsd:string">
  </xsd:restriction>
 </xsd:simpleType>
 <xsd:simpleType name="SchemeNameList">
  <xsd:restriction base="xsd:string">
  </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="ModeList">
  <xsd:restriction base="xsd:string">
   <xsd:enumeration value="EXTRACT"/>
   <xsd:enumeration value="FORMAT"/>
  </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="CompactionMethodList">
  <xsd:restriction base="xsd:string">
   <xsd:enumeration value="32-bit"/>
   <xsd:enumeration value="16-bit"/>
   <xsd:enumeration value="8-bit"/>
   <xsd:enumeration value="7-bit"/>
   <xsd:enumeration value="6-bit"/>
   <xsd:enumeration value="5-bit"/>
  </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="PadDirectionList">
  <xsd:restriction base="xsd:string">
   <xsd:enumeration value="LEFT"/>
   <xsd:enumeration value="RIGHT"/>
  </xsd:restriction>
 </xsd:simpleType>

 <xsd:complexType name="Field">
  <xsd:attribute name="seq" type="xsd:integer" use="required"/>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="bitLength" type="xsd:integer"/>
  <xsd:attribute name="characterSet" type="xsd:string" use="required"/>
  <xsd:attribute name="compaction" type="tdt:CompactionMethodList"/>
  <xsd:attribute name="compression" type="xsd:string"/>
  <xsd:attribute name="padChar" type="xsd:string"/>
  <xsd:attribute name="padDir" type="tdt:PadDirectionList"/>
  <xsd:attribute name="decimalMinimum" type="xsd:long"/>
  <xsd:attribute name="decimalMaximum" type="xsd:long"/>
  <xsd:attribute name="length" type="xsd:integer"/>
 </xsd:complexType>

 <xsd:complexType name="Option">
  <xsd:sequence>
   <xsd:element name="field" type="tdt:Field" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="optionKey" type="xsd:string" use="required"/>
  <xsd:attribute name="pattern" type="xsd:string"/>
  <xsd:attribute name="grammar" type="xsd:string" use="required"/>
 </xsd:complexType>

 <xsd:complexType name="Rule">
  <xsd:attribute name="type" type="tdt:ModeList" use="required"/>
  <xsd:attribute name="inputFormat" type="tdt:InputFormatList" use="required"/>
  <xsd:attribute name="seq" type="xsd:integer" use="required"/>
  <xsd:attribute name="newFieldName" type="xsd:string" use="required"/>
  <xsd:attribute name="characterSet" type="xsd:string" use="required"/>
  <xsd:attribute name="padChar" type="xsd:string"/>
  <xsd:attribute name="padDir" type="tdt:PadDirectionList"/>
  <xsd:attribute name="decimalMinimum" type="xsd:long"/>
  <xsd:attribute name="decimalMaximum" type="xsd:long"/>
  <xsd:attribute name="length" type="xsd:string"/>
  <xsd:attribute name="function" type="xsd:string" use="required"/>
  <xsd:attribute name="tableURI" type="xsd:string"/>
  <xsd:attribute name="tableParams" type="xsd:string"/>
  <xsd:attribute name="tableXPath" type="xsd:string"/>
  <xsd:attribute name="tableSQL" type="xsd:string"/>
 </xsd:complexType>

 <xsd:complexType name="Level">
  <xsd:sequence>
   <xsd:element name="option" type="tdt:Option" minOccurs="1"
     maxOccurs="unbounded"/>
   <xsd:element name="rule" type="tdt:Rule" minOccurs="0"
     maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="type" type="tdt:LevelTypeList" use="required"/>
```

```
  <xsd:attribute name="prefixMatch" type="xsd:string"/>
  <xsd:attribute name="requiredParsingParameters" type="xsd:string"/>
  <xsd:attribute name="requiredFormattingParameters" type="xsd:string"/>
 </xsd:complexType>

 <xsd:complexType name="Scheme">
  <xsd:sequence>
   <xsd:element name="level" type="tdt:Level" minOccurs="4" maxOccurs="5"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="tdt:SchemeNameList" use="required"/>
  <xsd:attribute name="optionKey" type="xsd:string" use="required"/>
 </xsd:complexType>
 <xsd:complexType name="TagDataTranslation">
  <xsd:sequence>
   <xsd:element name="scheme" type="tdt:Scheme" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="version" type="xsd:string" use="required"/>
  <xsd:attribute name="date" type="xsd:dateTime" use="required"/>
 </xsd:complexType>
 <xsd:element name="TagDataTranslation" type="tdt:TagDataTranslation"/>
</xsd:schema>
```

A

Multithreaded extproc Agent

This appendix explains what the multithreaded `extproc` agent is, how it contributes to the overall efficiency of a distributed database system, and how to administer it.

Topics:

- Why Use the Multithreaded extproc Agent?
- Multithreaded extproc Agent Architecture
- Administering the Multithreaded extproc Agent

Why Use the Multithreaded extproc Agent?

This section explains how the multithreaded `extproc` agent contributes to the efficiency of external procedures.

Topics:

- The Challenge of Dedicated Agent Architecture
- The Advantage of Multithreading

The Challenge of Dedicated Agent Architecture

By default, an `extproc` agent is started for each user session and the `extproc` agent process terminates only when the user session ends.

This architecture can consume an unnecessarily large amount of system resources. For example, suppose that several thousand user sessions simultaneously spawn `extproc` agent processes. Because an `extproc` agent process is started for each session, several thousand `extproc` agent processes are running concurrently. The `extproc` agent processes operate regardless of whether each individual `extproc` agent process is active at the moment. Thus `extproc` agent processes and open connections can consume a disproportionate amount of system resources. When sessions connect to the Oracle database server, this problem is addressed by starting the server in shared server mode. Shared server mode allows database connections to be shared by a small number of server processes.

The Advantage of Multithreading

The Oracle shared server architecture assumes that even when several thousand user sessions are open, only a small percentage of these connections are active at any given time. In shared server mode, there is a pool of shared server processes. User sessions connect to dispatcher processes that place the requested tasks in a queue. The tasks are picked up by the first available shared server processes. The number of shared server processes is usually less that the number of user sessions.

The multithreaded `extproc` agent provides similar functionality for connections to external procedures. The multithreaded `extproc` agent architecture uses a pool of shared agent threads. The tasks requested by the user sessions are put in a queue and are picked up by the first available multithreaded `extproc` agent thread. Because only a small percentage of user connections are active at a given moment, using a multithreaded `extproc` architecture allows more efficient use of system resources.

Multithreaded extproc Agent Architecture

One multithreaded `extproc` agent must be started for each system identifier (SID) before attempting to connect to the external procedure. This is done using the agent control utility `agtctl`. This utility is also used to configure the agent and to shut down the agent.

Each Oracle Net listener that is running on a system listens for incoming connection requests for a set of SIDs. If the SID in an incoming Oracle Net connect string is one of the SIDs for which the listener is listening, then that listener processes the connection. Further, if a multithreaded `extproc` agent was started for the SID, then the listener passes the request to that `extproc` agent.

In the architecture for multithreaded `extproc` agents, each incoming connection request is processed by different kinds of threads:

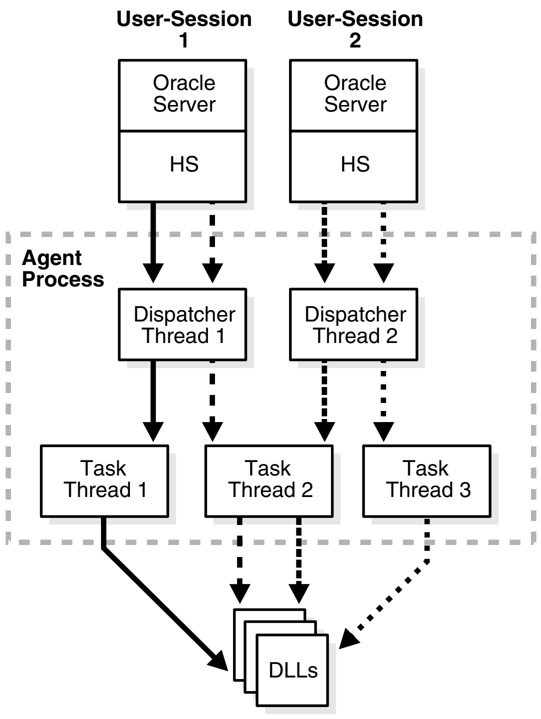
- A single **monitor** thread. The monitor thread is responsible for the following:
  - Maintaining communication with the listener
  - Monitoring the load on the process
  - Starting and stopping threads when required
- Several **dispatcher** threads. The dispatcher threads are responsible for the following:
  - Handling communication with the Oracle server
  - Passing task requests to the task threads
- Several **task** threads. The task threads handle requests from the Oracle processes.

Figure A–1 illustrates the architecture of the multithreaded `extproc` agent. User sessions 1 and 2 issue requests for callouts to functions in some DLLs. These requests get serviced through heterogeneous services to the multithreaded `extproc` agent. These requests get handled by the agent's dispatcher threads, which then pass them on to the task threads. The task thread that is actually handling a request is responsible for loading the respective DLL and calling the function therein.

- All requests from a user session get handled by the same dispatcher thread. For example, dispatcher 1 handles communication with user session 1, and dispatcher 2 handles communication with user session 2. This is the case for the lifetime of the session.
- The individual requests can, however, be serviced by different task threads. For example, task thread 1 can handle the request from user session 1 at one time, and handle the request from user session 2 at another time.

> **See Also:** *Oracle Database Administrator's Guide*. for details on managing processes for external procedures

Figure A–1 Multithreaded extproc Agent Architecture



These three thread types roughly correspond to the Oracle multithreaded server PMON, dispatcher, and shared server processes, respectively.

Note: All requests from a user session go through the same dispatcher thread, but can be serviced by different task threads. Also, several task threads can use the same connection to the external procedure.

The following sections explain each type of thread in more detail:

- Monitor Thread
- Dispatcher Threads
- Task Threads

See Also: "Administering the Multithreaded extproc Agent" on page A-4 for more information about starting and stopping the multithreaded `exproc` agent by using the agent control utility `agtctl`

Monitor Thread

When the agent control utility `agtctl` starts a multithreaded `extproc` agent for a SID, `agtctl` creates the monitor thread. The monitor thread performs the following functions:

- Creates the dispatcher and task threads.
- Registers the dispatcher threads with all the listeners that are handling connections to this `extproc` agent. While the dispatcher for this SID is running,

the listener does not start a new process when it gets an incoming connection. Instead, the listener gives the connection to this same dispatcher.

- Monitors the other threads and sends load information about the dispatcher threads to all the listener processes handling connections to this `extproc` agent. This enables the listeners to give incoming connections to the least loaded dispatcher.
- Continues to monitor each of the threads it has created.

Dispatcher Threads

Dispatcher threads perform the following functions:

- Accept incoming connections and task requests from Oracle servers.
- Place incoming requests on a queue for a task thread to pick up.
- Send results of a request back to the server that issued the request.

Note: After a user session establishes a connection with a dispatcher, all requests from that user session go to the same dispatcher until the end of the user session.

Task Threads

Task threads perform the following functions:

- Pick up requests from a queue.
- Perform the necessary operations.
- Place the results on a queue for a dispatcher to pick up.

Administering the Multithreaded extproc Agent

One multithreaded `extproc` agent must be started for each system identifier (SID) before attempting to connect to the external procedure.

A multithreaded `extproc` agent is started, stopped, and configured by an agent control utility called `agtctl`, which works like `lsnrctl`. However, unlike `lsnrctl`, which reads a configuration file (`listener.ora`), `agtctl` takes configuration information from the command line and writes it to a control file.

Topics:

- Agent Control Utility (agtctl) Commands
- Using agtctl in Single-Line Command Mode
- Using Shell Mode Commands
- Configuration Parameters for Multithreaded extproc Agent Control

Agent Control Utility (agtctl) Commands

You can start and stop `agtctl` and create and maintain its control file by using the commands shown in Table A–1.

Table A–1 Agent Control Utility (agtctl) Commands

Command	Description
`startup`	Starts a multithreaded `extproc` agent
`shutdown`	Stops a multithreaded `extproc` agent
`set`	Sets a configuration parameter for a multithreaded `extproc` agent
`unset`	Causes a parameter to revert to its default value
`show`	Displays the value of a configuration parameter
`delete`	Deletes the entry for a particular SID from the control file
`exit`	Exits shell mode
`help`	Lists available commands

These commands can be issued in one of two ways:

- You can issue commands from the UNIX or DOS shell. This mode is called single-line command mode.
- You can enter `agtctl` and an `AGTCTL>` prompt appears. You then can enter commands from within the `agtctl` shell. This mode is called shell mode.

The syntax and parameters for `agtctl` commands depend on the mode in which they are issued.

Note:

- All commands are case-sensitive.
- The agent control utility puts its control file in either the directory pointed to by the `AGTCTL_ADMIN` environment variable or in the directory pointed to by the `TNS_ADMIN` environment variable. Ensure that at least one of these environment variables is set and that it points to a directory to which the agent has access.
- If the multithreaded `extproc` agent requires an environment variable to be set, or if the `ENVS` parameter was used when configuring the `listener.ora` entry for the agent working in dedicated mode, then all required environment variables must be set in the UNIX or DOS shell that runs the `agtctl` utility.

Using agtctl in Single-Line Command Mode

This section describes the use of `agtctl` commands. They are presented in single-line command mode.

Setting Configuration Parameters for a Multithreaded extproc Agent

Set the configuration parameters for a multithreaded `extproc` agent before you start the agent. If a configuration parameter is not specifically set, a default value is used. Configuration parameters and their default values are shown in Table A–2.

Use the `set` command to set multithreaded `extproc` agent configuration parameters.

Syntax

```
agtctl set parameter parameter_value agent_sid
```

`parameter` is the parameter that you are setting.

`parameter_value` is the value being assigned to that parameter.

`agent_sid` is the SID that this agent will service. This must be specified for single-line command mode.

Example

```
agtctl set max_dispatchers 5 salesDB
```

Starting a Multithreaded extproc Agent

Use the `startup` command to start a multithreaded `extproc` agent.

Syntax

```
agtctl startup extproc agent_sid
```

`agent_sid` is the SID that this multithreaded `extproc` agent will service. This must be specified for single-line command mode.

Example

```
agtctl startup extproc salesDB
```

Shutting Down a Multithreaded extproc Agent

Use the `shutdown` command to stop a multithreaded `extproc` agent. There are three forms of shutdown:

- Normal (default)

  `agtctl` asks the multithreaded `extproc` agent to terminate itself gracefully. All sessions complete their current operations and then shut down.

- Immediate

  `agtctl` tells the multithreaded `extproc` agent to terminate immediately. The agent exits immediately regardless of the state of current sessions.

- Abort

  Without talking to the multithreaded `extproc` agent, `agtctl` issues a system call to kill it.

Syntax

```
agtctl shutdown [immediate|abort] agent_sid
```

`agent_sid` is the SID that the multithreaded `extproc` agent services. It must be specified for single-line command mode.

Example

```
agtctl shutdown immediate salesDB
```

Examining the Value of Configuration Parameters

To examine the value of a configuration parameter, use the `show` command.

Syntax

```
agtctl show parameter agent_sid
```

parameter is the parameter that you are examining.

agent_sid is the SID that this multithreaded `extproc` agent will service. This must be specified for single-line command mode.

Example

```
agtctl show max_dispatchers salesDB
```

Resetting a Configuration Parameter to Its Default Value

You can reset a configuration parameter to its default value using the `unset` command.

Syntax

```
agtctl unset parameter agent_sid
```

parameter is the parameter that you are resetting (or changing).

agent_sid is the SID that this multithreaded `extproc` agent services. It must be specified for single-line command mode.

Example

```
agtctl unset max_dispatchers salesDB
```

Deleting an Entry for a Specific SID from the Control File

The `delete` command deletes the entry for the specified SID from the control file.

Syntax

```
agtctl delete agent_sid
```

agent_sid is the SID entry to delete.

Example

```
agtctl delete salesDB
```

Requesting Help

Use the `help` command to view a list of available commands for `agtctl` or to see the syntax for a particular command.

Syntax

```
agtctl help [command]
```

command is the name of the command whose syntax you want to view. The default is all `agtctl` commands.

Example

```
agtctl help set
```

Using Shell Mode Commands

In shell mode, start `agtctl` by entering the following:

```
agtctl
```

This results in the prompt `AGTCTL>`. Thereafter, because you are issuing commands from within the `agtctl` shell, you do not need to prefix the command string with `agtctl`.

Set the name of the agent SID by entering the following:

```
AGTCTL> set agent_sid agent_sid
```

All subsequent commands are assumed to be for the specified SID until the `agent_sid` value is changed. Unlike single-line command mode, you do not specify *`agent_sid`* in the command string.

You can set the language for error messages as follows:

```
AGTCTL> set language language
```

The commands themselves are the same as those for the single-line command mode. To exit shell mode, enter `exit`.

The following are examples of shell mode commands.

Example: Setting a Configuration Parameter

This example sets a new value for the `shutdown_address` configuration parameter.

```
AGTCTL> set shutdown_address (address=(protocol=ipc)(key=oraDBsalesDB))
```

Example: Starting a Multithreaded extproc Agent

This example starts a multithreaded `extproc` agent.

```
AGTCTL> startup extproc
```

Configuration Parameters for Multithreaded extproc Agent Control

Table A–2 lists the configuration parameters for the agent control utility.

Table A–2 Initialization Parameters for agtctl

Parameter	Description	Default Value
`max_dispatchers`	Maximum number of dispatchers	1
`tcp_dispatchers`	Number of dispatchers listening on TCP (the rest are using IPC)	0
`max_task_threads`	Number of task threads	2

Table A–2 (Cont.) Initialization Parameters for agtctl

Parameter	Description	Default Value
`max_sessions`	Maximum number of sessions	5

`listener_address`	Address on which the listener is listening (needed for registration)	`(ADDRESS_LIST=`
		`    (ADDRESS=`
		`        (PROTOCOL=IPC)`
		`        (KEY=PNPKEY))`
		`    (ADDRESS=`
		`        (PROTOCOL=IPC)`
		`        (KEY=`*`listener_sid`*`))`
		`    (ADDRESS=`
		`        (PROTOCOL=TCP)`
		`        (HOST=127.0.0.1)`
		`        (PORT=1521)))`
		Note: *`listener_sid`* is the IPC key of the address, on the Oracle database, on which the listener is listening.

`shutdown_address`	Address the agent uses to communicate with the listener. This is the address on which the agent listens for all communication, including shutdown messages from `agtctl`.	`(ADDRESS=`
		`    (PROTOCOL=IPC)`
		`    (KEY=`*`listener_sid`*` || `*`agent_sid`*`))`
		`    (ADDRESS=`
		`        (PROTOCOL=TCP)`
		`        (HOST=127.0.0.1)`
		`        (PORT=1521))`
		Notes:
		■ *`agent_sid`* is the SID of the multithreaded `extproc` agent.
		■ `||` indicates that *`listener_sid`* and *`agent_sid`* are concatenated into one string.

Index