

Oracle® Database
Development Guide
12c Release 1 (12.1)
E41452-06

July 2014

Copyright © 1996, 2014, Oracle and/or its affiliates. All rights reserved.

Primary Authors: Chuck Murray, Tom Kyte

Contributing Authors: D. Adams, L. Ashdown, P. Huey, S. Moore, E. Paapanen, R. Strohm, R. Ward

Contributor: The Oracle Database 12c documentation is dedicated to Mark Townsend, who was an inspiration to all who worked on this release.

Contributors: D. Alpern, G. Arora, T. Chang, B. Cheng, R. Day, R. Decker, G. Doherty, D. Elson, A. Ganesh, M. Hartstein, Y. Hu, J. Huang, C. Iyer, N. Jain, V. Krishnaswamy, R. Kumar, S. Kumar, C. Lei, B. Llewellyn, K. Mohan, V. Moore, K. Muthukaruppan, J. Muller, R. Murthy, R. Pang, B. Sinha, S. Vemuri, W. Wang, D. Wong

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	xxi
Audience.....	xxxi
Documentation Accessibility	xxxi
Related Documents	xxxii
Conventions	xxxii
Changes in This Release for Oracle Database Development Guide	xxxv
Changes in Oracle Database 12c Release 1 (12.1.0.2).....	xxxv
Changes in Oracle Database 12c Release 1 (12.1.0.1).....	xxxv

Part I Database Development Fundamentals

1 Design Basics

1.1 Design for Performance	1-1
1.2 Design for Scalability.....	1-2
1.3 Design for Extensibility.....	1-2
1.3.1 Data Cartridges	1-2
1.3.2 External Procedures.....	1-3
1.3.3 User-Defined Functions and Aggregate Functions	1-3
1.3.4 Object-Relational Features.....	1-3
1.4 Design for Security	1-4
1.5 Design for Availability.....	1-4
1.6 Design for Portability	1-4
1.7 Design for Diagnosability	1-5
1.8 Design for Special Environments	1-5
1.8.1 Data Warehousing	1-5
1.8.2 Online Transaction Processing (OLTP)	1-6
1.9 Features for Special Scenarios	1-7
1.9.1 SQL Analytic Functions	1-7
1.9.2 Materialized Views.....	1-8
1.9.3 Partitioning	1-9
1.9.4 Temporal Validity Support	1-10

2 Performance and Scalability

2.1	Performance Strategies.....	2-1
2.1.1	Designing Your Data Model to Perform Well	2-1
2.1.1.1	Analyze the Data Requirements of the Application.....	2-2
2.1.1.2	Create the Database Design for the Application.....	2-2
2.1.1.3	Implement the Database Application.....	2-3
2.1.1.4	Maintain the Database and Database Application	2-4
2.1.2	Setting Performance Goals (Metrics)	2-4
2.1.3	Benchmarking Your Application.....	2-4
2.2	Tools for Performance	2-5
2.2.1	DBMS_APPLICATION_INFO Package	2-5
2.2.2	SQL Trace Facility (SQL_TRACE).....	2-6
2.2.3	EXPLAIN PLAN Statement.....	2-6
2.3	Monitoring Database Performance	2-7
2.3.1	Automatic Database Diagnostic Monitor (ADDM)	2-7
2.3.2	Monitoring Real-Time Database Performance	2-8
2.3.3	Responding to Performance-Related Alerts	2-8
2.3.4	SQL Advisors and Memory Advisors	2-8
2.4	Using Instrumentation	2-9
2.5	Testing for Performance.....	2-9
2.6	Using Bind Variables.....	2-10
2.7	Using Client Result Cache	2-11
2.7.1	About Client Result Cache	2-12
2.7.2	Benefits of Client Result Cache	2-12
2.7.3	Guidelines for Using Client Result Cache.....	2-13
2.7.3.1	SQL Hints.....	2-14
2.7.3.2	Table Annotation	2-14
2.7.3.3	Session Parameter.....	2-15
2.7.3.4	Effective Table Result Cache Mode.....	2-15
2.7.3.5	Displaying Effective Table Result Cache Mode	2-16
2.7.3.6	Result Cache Mode Use Cases.....	2-16
2.7.3.7	Queries Never Result Cached in Client Result Cache.....	2-16
2.7.4	Client Result Cache Consistency	2-17
2.7.5	Deployment-Time Settings for Client Result Cache	2-17
2.7.5.1	Server Initialization Parameters	2-18
2.7.5.2	Client Configuration Parameters	2-19
2.7.6	Client Result Cache Statistics	2-19
2.7.7	Validation of Client Result Cache	2-20
2.7.7.1	Measure Execution Times	2-20
2.7.7.2	Query V\$MYSTAT	2-20
2.7.7.3	Query V\$SQLAREA.....	2-20
2.7.8	Client Result Cache and Server Result Cache	2-21
2.7.9	Client Result Cache Demo Files.....	2-22
2.7.10	Client Result Cache Compatibility with Previous Releases	2-22
2.7.11	Where to Find More Information About Client Result Cache	2-22
2.8	Statement Caching	2-22
2.9	OCI Client Statement Cache Auto-Tuning.....	2-23

2.10	Client-Side Deployment Parameters.....	2-24
2.11	Using Query Change Notification.....	2-24
2.12	Using Database Resident Connection Pool.....	2-24
2.12.1	About Database Resident Connection Pool	2-25
2.12.2	Configuring DRCP	2-26
2.12.3	Using JDBC with DRCP	2-27
2.12.4	Using OCI Session Pool APIs with DRCP	2-27
2.12.5	Session Purity and Connection Class.....	2-28
2.12.5.1	Session Purity.....	2-28
2.12.5.2	Connection Class	2-28
2.12.5.3	Session Purity and Connection Class Defaults	2-29
2.12.6	Starting Database Resident Connection Pool	2-30
2.12.7	Enabling DRCP	2-30
2.12.8	Benefiting from the Scalability of DRCP in an OCI Application	2-30
2.12.9	Benefiting from the Scalability of DRCP in a Java Application	2-31
2.12.10	Best Practices for Using DRCP.....	2-31
2.12.11	Compatibility and Migration.....	2-32
2.12.12	DRCP Restrictions	2-32
2.12.13	Using DRCP with Custom Pools	2-33
2.12.14	Explicitly Marking Sessions Stateful or Stateless	2-33
2.12.15	Using DRCP with Oracle Real Application Clusters.....	2-34
2.12.16	Using DRCP with Pluggable Databases	2-35
2.12.17	DRCP with Data Guard	2-35
2.13	Using Runtime Connection Load Balancing.....	2-35
2.13.1	About Runtime Connection Load Balancing	2-35
2.13.2	Enabling and Disabling Runtime Connection Load Balancing	2-36
2.13.2.1	OCI.....	2-36
2.13.2.2	OCCI.....	2-37
2.13.2.3	JDBC	2-37
2.13.2.4	ODP.NET	2-37
2.13.3	Receiving Load Balancing Advisory FAN Events	2-37

3 Security

3.1	Enabling User Access with Grants, Roles, and Least Privilege.....	3-1
3.2	Automating Database Logins.....	3-2
3.3	Controlling User Access with Fine-Grained Access Control.....	3-3
3.4	Using Invoker's and Definer's Rights for Procedures and Functions	3-4
3.4.1	What Are Invoker's Rights and Definer's Rights?	3-4
3.4.2	Protecting Users Who Run Invoker's Rights Procedures and Functions	3-5
3.4.3	How Default Rights Are Handled for Java Stored Procedures	3-5
3.5	Managing External Procedures for Your Applications	3-6
3.6	Auditing User Activity	3-6

4 High Availability

4.1	Failover and Query Replay.....	4-1
4.1.1	Transparent Application Failover (TAF).....	4-1

4.1.1.1	About Transparent Application Failover.....	4-1
4.1.1.2	Configuring Transparent Application Failover	4-2
4.1.1.3	Using Transparent Application Failover Callbacks	4-2
4.1.2	Application Continuity	4-3
4.2	Fast Application Notification (FAN) and Fast Connection Failover (FCF).....	4-3
4.2.1	About Fast Application Notification (FAN)	4-3
4.2.2	About Fast Connection Failover (FCF)	4-4
4.3	Application Continuity and Transaction Guard	4-5
4.3.1	Transaction Guard	4-6
4.3.2	Application Continuity for Java	4-6
4.4	Service and Load Management for Database Clouds.....	4-7
4.4.1	About Service and Load Management for Database Clouds	4-7

5 Advanced PL/SQL Features

5.1	PL/SQL Data Types	5-1
5.2	Dynamic SQL.....	5-1
5.3	PL/SQL Optimize Level	5-1
5.4	Compiling PL/SQL Units for Native Execution	5-2
5.5	Exception Handling	5-2
5.6	Conditional Compilation	5-2
5.7	Bulk Binding	5-2

Part II SQL for Application Developers

6 SQL Processing for Application Developers

6.1	Description of SQL Statement Processing.....	6-1
6.1.1	Stages of SQL Statement Processing	6-2
6.1.2	Shared SQL Areas	6-3
6.2	Grouping Operations into Transactions	6-4
6.2.1	Deciding How to Group Operations in Transactions	6-4
6.2.2	Improving Transaction Performance.....	6-4
6.2.3	Managing Commit Redo Action.....	6-5
6.2.4	Determining Transaction Outcome After a Recoverable Outage.....	6-7
6.2.4.1	Understanding Transaction Guard	6-7
6.2.4.2	Understanding DBMS_APP_CONT.GET_LTXID_OUTCOME	6-9
6.2.4.3	Using Transaction Guard	6-11
6.3	Ensuring Repeatable Reads with Read-Only Transactions	6-11
6.4	Locking Tables Explicitly	6-12
6.4.1	Privileges Required to Acquire Table Locks.....	6-13
6.4.2	Choosing a Locking Strategy	6-13
6.4.2.1	When to Lock with ROW SHARE MODE and ROW EXCLUSIVE MODE	6-14
6.4.2.2	When to Lock with SHARE MODE	6-14
6.4.2.3	When to Lock with SHARE ROW EXCLUSIVE MODE	6-15
6.4.2.4	When to Lock with EXCLUSIVE MODE	6-16
6.4.3	Letting Oracle Database Control Table Locking	6-16
6.4.4	Explicitly Acquiring Row Locks.....	6-16

6.4.5	Examples of Concurrency Under Explicit Locking	6-17
6.5	Using Oracle Lock Management Services (User Locks).....	6-24
6.5.1	When to Use User Locks.....	6-24
6.5.2	Viewing and Monitoring Locks.....	6-25
6.6	Using Serializable Transactions for Concurrency Control	6-25
6.6.1	Transaction Interaction and Isolation Level	6-26
6.6.2	Setting Isolation Levels	6-27
6.6.3	Serializable Transactions and Referential Integrity	6-28
6.6.4	READ COMMITTED and SERIALIZABLE Isolation Levels	6-30
6.6.4.1	Transaction Set Consistency Differences	6-30
6.6.4.2	Choosing Transaction Isolation Levels.....	6-30
6.7	Nonblocking and Blocking DDL Statements	6-31
6.8	Autonomous Transactions.....	6-32
6.8.1	Examples of Autonomous Transactions.....	6-34
6.8.1.1	Ordering a Product.....	6-35
6.8.1.2	Withdrawing Money from a Bank Account	6-35
6.8.2	Declaring Autonomous Routines	6-38
6.9	Resuming Execution After Storage Allocation Errors.....	6-39
6.9.1	What Operations Have Resumable Storage Allocation?	6-39
6.9.2	Handling Suspended Storage Allocation	6-39
6.9.2.1	Using an AFTER SUSPEND Trigger in the Application.....	6-40
6.9.2.2	Checking for Suspended Statements	6-41

7 Using SQL Data Types in Database Applications

7.1	Using the Correct and Most Specific Data Type	7-1
7.1.1	How the Correct Data Type Increases Data Integrity	7-2
7.1.2	How the Most Specific Data Type Decreases Storage Requirements	7-2
7.1.3	How the Correct Data Type Improves Performance	7-2
7.2	Representing Character Data	7-5
7.3	Representing Numeric Data	7-6
7.3.1	Floating-Point Number Components	7-7
7.3.2	Floating-Point Number Formats	7-8
7.3.2.1	Binary Floating-Point Formats	7-8
7.3.3	Representing Special Values with Native Floating-Point Data Types.....	7-9
7.3.4	Comparing Native Floating-Point Values.....	7-10
7.3.5	Arithmetic Operations with Native Floating-Point Data Types.....	7-10
7.3.6	Conversion Functions for Native Floating-Point Data Types	7-11
7.3.7	Client Interfaces for Native Floating-Point Data Types	7-11
7.4	Representing Date and Time Data	7-12
7.4.1	Displaying Current Date and Time.....	7-13
7.4.2	Inserting and Displaying Dates	7-14
7.4.3	Inserting and Displaying Times	7-15
7.4.4	Arithmetic Operations with Datetime Data Types.....	7-16
7.4.5	Conversion Functions for Datetime Data Types.....	7-16
7.4.6	Importing, Exporting, and Comparing Datetime Types.....	7-17
7.5	Representing Specialized Data.....	7-17
7.5.1	Representing Spatial Data	7-18

7.5.2	Representing Multimedia Data.....	7-18
7.5.3	Representing Large Amounts of Data	7-18
7.5.3.1	Large Objects (LOBs).....	7-18
7.5.3.2	LONG and LONG RAW Data Types	7-19
7.5.4	Representing Searchable Text	7-19
7.5.5	Representing XML Data.....	7-19
7.5.6	Representing Dynamically Typed Data	7-20
7.5.7	Representing ANSI, DB2, and SQL/DS Data.....	7-21
7.6	Identifying Rows by Address	7-22
7.7	Displaying Metadata for SQL Operators and Functions	7-23
7.7.1	ARGn Data Type.....	7-24
7.7.2	DISP_TYPE Data Type.....	7-24
7.7.3	SQL Data Type Families	7-24

8 Using Regular Expressions in Database Applications

8.1	Overview of Regular Expressions	8-1
8.2	Oracle SQL Support for Regular Expressions.....	8-2
8.3	Oracle SQL and POSIX Regular Expression Standard	8-4
8.4	Operators in Oracle SQL Regular Expressions.....	8-4
8.4.1	POSIX Operators in Oracle SQL Regular Expressions.....	8-4
8.4.2	Oracle SQL Multilingual Extensions to POSIX Standard	8-7
8.4.3	Oracle SQL PERL-Influenced Extensions to POSIX Standard	8-7
8.5	Using Regular Expressions in SQL Statements: Scenarios	8-9
8.5.1	Using a Constraint to Enforce a Phone Number Format.....	8-9
8.5.2	Using Back References to Reposition Characters.....	8-10

9 Using Indexes in Database Applications

9.1	Guidelines for Managing Indexes	9-1
9.2	Managing Indexes.....	9-2
9.3	When to Use Domain Indexes.....	9-2
9.4	When to Use Function-Based Indexes	9-2
9.4.1	Advantages of Function-Based Indexes	9-3
9.4.2	Disadvantages of Function-Based Indexes	9-4
9.4.3	Examples of Function-Based Indexes	9-5

10 Maintaining Data Integrity in Database Applications

10.1	Enforcing Business Rules with Constraints	10-2
10.2	Enforcing Business Rules with Both Constraints and Application Code	10-3
10.3	Creating Indexes for Use with Constraints	10-4
10.4	When to Use NOT NULL Constraints	10-4
10.5	When to Use Default Column Values.....	10-6
10.6	Choosing a Primary Key for a Table (PRIMARY KEY Constraint)	10-6
10.7	When to Use UNIQUE Constraints.....	10-7
10.8	Enforcing Referential Integrity with FOREIGN KEY Constraints.....	10-8
10.8.1	FOREIGN KEY Constraints and NULL Values	10-10
10.8.2	Defining Relationships Between Parent and Child Tables	10-10

10.8.3	Rules for Multiple FOREIGN KEY Constraints	10-11
10.8.4	Deferring Constraint Checks.....	10-11
10.9	Minimizing Space and Time Overhead for Indexes Associated with Constraints	10-13
10.10	Guidelines for Indexing Foreign Keys.....	10-13
10.11	Referential Integrity in a Distributed Database.....	10-13
10.12	When to Use CHECK Constraints	10-14
10.12.1	Restrictions on CHECK Constraints	10-14
10.12.2	Designing CHECK Constraints	10-15
10.12.3	Rules for Multiple CHECK Constraints	10-15
10.12.4	Choosing Between CHECK and NOT NULL Constraints	10-15
10.13	Examples of Defining Constraints.....	10-16
10.13.1	Privileges Needed to Define Constraints	10-17
10.13.2	Naming Constraints	10-17
10.14	Enabling and Disabling Constraints	10-18
10.14.1	Why Disable Constraints?	10-18
10.14.2	Creating Enabled Constraints (Default)	10-18
10.14.3	Creating Disabled Constraints.....	10-19
10.14.4	Enabling Existing Constraints.....	10-19
10.14.5	Disabling Existing Constraints	10-20
10.14.6	Guidelines for Enabling and Disabling Key Constraints.....	10-20
10.14.7	Fixing Constraint Exceptions.....	10-21
10.15	Modifying Constraints	10-21
10.16	Renaming Constraints.....	10-22
10.17	Dropping Constraints.....	10-23
10.18	Managing FOREIGN KEY Constraints	10-24
10.18.1	Data Types and Names for Foreign Key Columns.....	10-24
10.18.2	Limit on Columns in Composite Foreign Keys.....	10-24
10.18.3	Foreign Key References Primary Key by Default	10-24
10.18.4	Privileges Required to Create FOREIGN KEY Constraints.....	10-24
10.18.5	Choosing How Foreign Keys Enforce Referential Integrity	10-24
10.19	Viewing Information About Constraints.....	10-25

Part III PL/SQL for Application Developers

11 Coding PL/SQL Subprograms and Packages

11.1	Overview of PL/SQL Subprograms	11-1
11.2	Overview of PL/SQL Packages	11-2
11.3	Overview of PL/SQL Units.....	11-3
11.3.1	PLSQL_OPTIMIZE_LEVEL Compilation Parameter	11-4
11.4	Creating PL/SQL Subprograms and Packages	11-6
11.4.1	Privileges Needed to Create Subprograms and Packages.....	11-6
11.4.2	Creating Subprograms and Packages	11-6
11.4.3	PL/SQL Object Size Limits.....	11-7
11.4.4	PL/SQL Data Types	11-7
11.4.4.1	PL/SQL Scalar Data Types	11-8
11.4.4.2	PL/SQL Composite Data Types.....	11-9

11.4.4.3	Abstract Data Types	11-10
11.4.5	Returning Result Sets to Clients	11-10
11.4.5.1	Advantages of Cursor Variables	11-11
11.4.5.2	Disadvantages of Cursor Variables.....	11-11
11.4.5.3	Returning Query Results Implicitly.....	11-14
11.4.6	Returning Large Amounts of Data from a Function	11-14
11.4.7	PL/SQL Function Result Cache.....	11-15
11.4.8	Overview of Bulk Binding.....	11-15
11.4.8.1	DML Statements that Reference Collections	11-16
11.4.8.2	SELECT Statements that Reference Collections.....	11-16
11.4.8.3	FOR Loops that Reference Collections and Return DML.....	11-17
11.4.9	PL/SQL Dynamic SQL	11-18
11.5	Altering PL/SQL Subprograms and Packages.....	11-18
11.6	Dropping PL/SQL Subprograms and Packages	11-19
11.7	Compiling PL/SQL Units for Native Execution	11-19
11.8	Invoking Stored PL/SQL Subprograms	11-19
11.8.1	Privileges Required to Invoke a Stored Subprogram	11-20
11.8.2	Invoking a Subprogram Interactively from Oracle Tools.....	11-20
11.8.3	Invoking a Subprogram from Another Subprogram	11-22
11.8.4	Invoking a Remote Subprogram	11-23
11.8.4.1	Synonyms for Remote Subprograms	11-24
11.8.4.2	Transactions That Invoke Remote Subprograms.....	11-25
11.9	Invoking Stored PL/SQL Functions from SQL Statements.....	11-25
11.9.1	Why Invoke PL/SQL Functions from SQL Statements?.....	11-26
11.9.2	Where PL/SQL Functions Can Appear in SQL Statements	11-27
11.9.3	When PL/SQL Functions Can Appear in SQL Expressions	11-27
11.9.4	Controlling Side Effects of PL/SQL Functions Invoked from SQL Statements	11-28
11.9.4.1	Restrictions on Functions Invoked from SQL Statements	11-28
11.9.4.2	PL/SQL Functions Invoked from Parallelized SQL Statements	11-29
11.9.4.3	PRAGMA RESTRICT_REFERENCES	11-30
11.10	Debugging Stored Subprograms	11-33
11.10.1	PL/Scope.....	11-33
11.10.2	PL/SQL Hierarchical Profiler	11-33
11.10.3	Compiling Code for Debugging	11-34
11.10.4	Privileges for Debugging PL/SQL and Java Stored Subprograms	11-34
11.10.5	DBMS_DEBUG Package	11-35
11.11	Package Invalidations and Session State	11-35

12 Using PL/Scope

12.1	Specifying Identifier Collection	12-1
12.2	PL/Scope Identifier Data for STANDARD and DBMS_STANDARD.....	12-2
12.2.1	Do You Need STANDARD and DBMS_STANDARD Identifier Data?	12-2
12.2.2	Does Your Database Have STANDARD and DBMS_STANDARD Identifier Data?	12-2
12.2.3	Recompiling STANDARD and DBMS_STANDARD.....	12-3
12.2.4	Running utlrp.sql to Recompile Invalid PL/SQL Objects.....	12-4
12.3	How Much Space is PL/Scope Data Using?	12-4

12.4	Viewing PL/Scope Data	12-5
12.4.1	Static Data Dictionary Views	12-5
12.4.1.1	Unique Keys	12-5
12.4.1.2	Context	12-5
12.4.1.3	Signature	12-7
12.4.2	Demo Tool.....	12-8
12.4.3	SQL Developer	12-8
12.5	Identifier Types that PL/Scope Collects.....	12-8
12.6	Usages that PL/Scope Reports	12-9
12.7	Sample PL/Scope Session.....	12-10

13 Using the PL/SQL Hierarchical Profiler

13.1	Overview of PL/SQL Hierarchical Profiler	13-1
13.2	Collecting Profile Data	13-2
13.3	Understanding Raw Profiler Output	13-3
13.3.1	Namespaces of Tracked Subprograms	13-6
13.3.2	Special Function Names.....	13-6
13.4	Analyzing Profile Data.....	13-7
13.4.1	Creating Hierarchical Profiler Tables	13-7
13.4.2	Understanding Hierarchical Profiler Tables.....	13-8
13.4.2.1	Hierarchical Profiler Database Table Columns	13-8
13.4.2.2	Distinguishing Between Overloaded Subprograms.....	13-10
13.4.2.3	Hierarchical Profiler Tables for Sample PL/SQL Procedure	13-10
13.4.2.4	Examples of Calls to DBMS_HPROF.analyze with Options.....	13-12
13.5	plshprof Utility	13-13
13.5.1	plshprof Options.....	13-14
13.5.2	HTML Report from a Single Raw Profiler Output File.....	13-14
13.5.2.1	First Page of Report	13-14
13.5.2.2	Function-Level Reports.....	13-15
13.5.2.3	Module-Level Reports	13-16
13.5.2.4	Namespace-Level Reports.....	13-16
13.5.2.5	Parents and Children Report for a Function	13-17
13.5.3	HTML Difference Report from Two Raw Profiler Output Files.....	13-18
13.5.3.1	Difference Report Conventions	13-19
13.5.3.2	First Page of Difference Report.....	13-19
13.5.3.3	Function-Level Difference Reports	13-20
13.5.3.4	Module-Level Difference Reports	13-21
13.5.3.5	Namespace-Level Difference Reports	13-22
13.5.3.6	Parents and Children Difference Report for a Function	13-22

14 Developing PL/SQL Web Applications

14.1	Overview of PL/SQL Web Applications.....	14-1
14.2	Implementing PL/SQL Web Applications.....	14-2
14.2.1	PL/SQL Gateway	14-2
14.2.1.1	mod_plsql	14-2
14.2.1.2	Embedded PL/SQL Gateway	14-3

14.2.2	PL/SQL Web Toolkit	14-3
14.3	Using mod_plsql Gateway to Map Client Requests to a PL/SQL Web Application	14-4
14.4	Using Embedded PL/SQL Gateway.....	14-4
14.4.1	How Embedded PL/SQL Gateway Processes Client Requests.....	14-5
14.4.2	Installing Embedded PL/SQL Gateway	14-6
14.4.3	Configuring Embedded PL/SQL Gateway	14-6
14.4.3.1	Configuring Embedded PL/SQL Gateway: Overview.....	14-6
14.4.3.2	Configuring User Authentication for Embedded PL/SQL Gateway	14-8
14.4.4	Invoking PL/SQL Stored Subprograms Through Embedded PL/SQL Gateway	14-17
14.4.5	Securing Application Access with Embedded PL/SQL Gateway.....	14-17
14.4.6	Restrictions in Embedded PL/SQL Gateway.....	14-18
14.4.7	Using Embedded PL/SQL Gateway: Scenario.....	14-18
14.5	Generating HTML Output with PL/SQL.....	14-20
14.6	Passing Parameters to PL/SQL Web Applications.....	14-21
14.6.1	Passing List and Dropdown-List Parameters from an HTML Form	14-21
14.6.2	Passing Option and Check Box Parameters from an HTML Form	14-22
14.6.3	Passing Entry-Field Parameters from an HTML Form	14-22
14.6.4	Passing Hidden Parameters from an HTML Form.....	14-24
14.6.5	Uploading a File from an HTML Form	14-24
14.6.6	Submitting a Completed HTML Form	14-24
14.6.7	Handling Missing Input from an HTML Form	14-24
14.6.8	Maintaining State Information Between Web Pages	14-25
14.7	Performing Network Operations in PL/SQL Subprograms	14-25
14.7.1	Sending E-Mail from PL/SQL	14-26
14.7.2	Getting a Host Name or Address from PL/SQL	14-27
14.7.3	Using TCP/IP Connections from PL/SQL	14-27
14.7.4	Retrieving HTTP URL Contents from PL/SQL	14-27
14.7.5	Using Tables, Image Maps, Cookies, and CGI Variables from PL/SQL	14-29

15 Using Continuous Query Notification (CQN)

15.1	About Object Change Notification (OCN)	15-2
15.2	About Query Result Change Notification (QRCN)	15-2
15.2.1	Guaranteed Mode	15-3
15.2.2	Best-Effort Mode	15-3
15.3	Events that Generate Notifications.....	15-4
15.3.1	Committed DML Transactions	15-5
15.3.2	Committed DDL Statements	15-5
15.3.3	Deregistration.....	15-6
15.3.4	Global Events.....	15-6
15.4	Notification Contents	15-7
15.5	Good Candidates for CQN	15-7
15.6	Creating CQN Registrations.....	15-10
15.7	Using PL/SQL to Create CQN Registrations	15-10
15.7.1	PL/SQL CQN Registration Interface	15-10
15.7.2	CQN Registration Options	15-11
15.7.2.1	Notification Type Option	15-11
15.7.2.2	QRCN Mode (QRCN Notification Type Only)	15-12

15.7.2.3	ROWID Option	15-12
15.7.2.4	Operations Filter Option (OCN Notification Type Only)	15-12
15.7.2.5	Transaction Lag Option (OCN Notification Type Only)	15-13
15.7.2.6	Notification Grouping Options	15-13
15.7.2.7	Reliable Option	15-14
15.7.2.8	Purge-on-Notify and Timeout Options	15-14
15.7.3	Prerequisites for Creating CQN Registrations	15-15
15.7.4	Queries that Can Be Registered for Object Change Notification (OCN)	15-15
15.7.5	Queries that Can Be Registered for Query Result Change Notification (QRCN)	15-15
15.7.5.1	Queries that Can Be Registered for QRCN in Guaranteed Mode	15-16
15.7.5.2	Queries that Can Be Registered for QRCN Only in Best-Effort Mode	15-17
15.7.5.3	Queries that Cannot Be Registered for QRCN in Either Mode	15-17
15.7.6	Using PL/SQL to Register Queries for CQN	15-18
15.7.6.1	Creating a PL/SQL Notification Handler	15-18
15.7.6.2	Creating a CQ_NOTIFICATION\$_REG_INFO Object	15-19
15.7.6.3	Identifying Individual Queries in a Notification	15-22
15.7.6.4	Adding Queries to an Existing Registration	15-22
15.7.7	Best Practices for CQN Registrations	15-23
15.7.8	Troubleshooting CQN Registrations	15-23
15.7.9	Deleting Registrations	15-24
15.7.10	Configuring CQN: Scenario	15-24
15.7.10.1	Creating a PL/SQL Notification Handler	15-25
15.7.10.2	Registering the Queries	15-27
15.8	Using OCI to Create CQN Registrations	15-29
15.8.1	Using OCI for Query Result Set Notifications	15-29
15.8.2	Using OCI to Register a Continuous Query Notification	15-29
15.8.3	Using OCI Subscription Handle Attributes for Continuous Query Notification	15-31
15.8.4	Using OCI_ATTR_CQ_QUERYID Attribute	15-32
15.8.5	Using OCI Continuous Query Notification Descriptors	15-33
15.8.5.1	OCI_DTYPE_CHDES	15-33
15.8.6	Demonstrating Continuous Query Notification in an OCI Sample Program	15-34
15.9	Querying CQN Registrations	15-44
15.10	Interpreting Notifications	15-45
15.10.1	Interpreting a CQ_NOTIFICATION\$_DESCRIPTOR Object	15-45
15.10.2	Interpreting a CQ_NOTIFICATION\$_TABLE Object	15-46
15.10.3	Interpreting a CQ_NOTIFICATION\$_QUERY Object	15-46
15.10.4	Interpreting a CQ_NOTIFICATION\$_ROW Object	15-47

Part IV Advanced Topics for Application Developers

16 Using Oracle Flashback Technology

16.1	Overview of Oracle Flashback Technology	16-1
16.1.1	Application Development Features	16-2
16.1.2	Database Administration Features	16-3
16.2	Configuring Your Database for Oracle Flashback Technology	16-3
16.2.1	Configuring Your Database for Automatic Undo Management	16-4

16.2.2	Configuring Your Database for Oracle Flashback Transaction Query	16-4
16.2.3	Configuring Your Database for Flashback Transaction.....	16-4
16.2.4	Enabling Oracle Flashback Operations on Specific LOB Columns	16-5
16.2.5	Granting Necessary Privileges.....	16-5
16.3	Using Oracle Flashback Query (SELECT AS OF).....	16-6
16.3.1	Example of Examining and Restoring Past Data	16-7
16.3.2	Guidelines for Oracle Flashback Query	16-7
16.4	Using Oracle Flashback Version Query.....	16-8
16.5	Using Oracle Flashback Transaction Query.....	16-10
16.6	Using Oracle Flashback Transaction Query with Oracle Flashback Version Query ...	16-11
16.7	Using DBMS_FLASHBACK Package	16-13
16.8	Using Flashback Transaction.....	16-14
16.8.1	Dependent Transactions	16-14
16.8.2	TRANSACTION_BACKOUT Parameters	16-15
16.8.3	TRANSACTION_BACKOUT Reports.....	16-16
16.8.3.1	*FLASHBACK_TXN_STATE	16-16
16.8.3.2	*FLASHBACK_TXN_REPORT	16-16
16.9	Using Flashback Data Archive.....	16-16
16.9.1	Creating a Flashback Data Archive.....	16-18
16.9.2	Altering a Flashback Data Archive	16-18
16.9.3	Dropping a Flashback Data Archive.....	16-19
16.9.4	Specifying the Default Flashback Data Archive.....	16-20
16.9.5	Enabling and Disabling Flashback Data Archive	16-20
16.9.6	DDL Statements on Tables Enabled for Flashback Data Archive.....	16-21
16.9.7	Viewing Flashback Data Archive Data.....	16-22
16.9.8	Flashback Data Archive Scenarios	16-22
16.9.8.1	Scenario: Using Flashback Data Archive to Enforce Digital Shredding.....	16-22
16.9.8.2	Scenario: Using Flashback Data Archive to Access Historical Data	16-23
16.9.8.3	Scenario: Using Flashback Data Archive to Generate Reports	16-23
16.9.8.4	Scenario: Using Flashback Data Archive for Auditing	16-24
16.9.8.5	Scenario: Using Flashback Data Archive to Recover Data	16-24
16.10	General Guidelines for Oracle Flashback Technology	16-25
16.11	Performance Guidelines for Oracle Flashback Technology.....	16-26
16.12	Multitenant Container Database Restrictions for Oracle Flashback Technology.....	16-26

17 Choosing a Programming Environment

17.1	Overview of Application Architecture	17-2
17.1.1	Client/Server Architecture.....	17-2
17.1.2	Server-Side Programming	17-2
17.1.3	Two-Tier and Three-Tier Architecture	17-2
17.2	Overview of the Program Interface.....	17-3
17.2.1	User Interface	17-3
17.2.2	Stateful and Stateless User Interfaces	17-3
17.3	Overview of PL/SQL	17-4
17.4	Overview of Oracle Database Java Support.....	17-4
17.4.1	Overview of Oracle JVM.....	17-4
17.4.2	Overview of Oracle JDBC	17-5

17.4.2.1	Oracle JDBC Drivers	17-6
17.4.2.2	Sample JDBC 2.0 Program.....	17-7
17.4.2.3	Sample Pre-2.0 JDBC Program	17-8
17.4.3	Overview of Oracle SQLJ.....	17-9
17.4.3.1	Benefits of SQLJ	17-10
17.4.3.2	SQLJ Stored Subprograms in the Server	17-10
17.4.4	Comparing Oracle JDBC and Oracle SQLJ	17-10
17.4.5	Overview of Oracle JPublisher	17-11
17.4.6	Overview of Java Stored Subprograms	17-11
17.4.7	Overview of Oracle Database Web Services.....	17-12
17.5	Choosing PL/SQL or Java	17-13
17.5.1	Similarities of PL/SQL and Java	17-13
17.5.2	PL/SQL Advantages Over Java.....	17-14
17.5.3	Java Advantages Over PL/SQL.....	17-14
17.6	Overview of Precompilers	17-15
17.6.1	Overview of the Pro*C/C++ Precompiler	17-15
17.6.2	Overview of the Pro*COBOL Precompiler	17-17
17.7	Overview of OCI and OCCI	17-18
17.7.1	Advantages of OCI and OCCI	17-19
17.7.2	OCI and OCCI Functions.....	17-20
17.7.3	Procedural and Nonprocedural Elements of OCI and OCCI Applications.....	17-20
17.7.4	Building an OCI or OCCI Application	17-21
17.8	Choosing a Precompiler or OCI.....	17-21
17.9	Overview of Oracle Data Provider for .NET (ODP.NET)	17-22
17.10	Overview of OraOLEDB	17-23

18 Developing Applications with Multiple Programming Languages

18.1	Overview of Multilanguage Programs	18-1
18.2	What Is an External Procedure?.....	18-3
18.3	Overview of Call Specification for External Procedures.....	18-3
18.4	Loading External Procedures	18-4
18.4.1	Define the C Procedures	18-5
18.4.2	Set Up the Environment.....	18-5
18.4.3	Identify the DLL.....	18-8
18.4.4	Publish the External Procedures.....	18-9
18.5	Publishing External Procedures.....	18-9
18.5.1	AS LANGUAGE Clause for Java Class Methods.....	18-10
18.5.2	AS LANGUAGE Clause for External C Procedures.....	18-10
18.5.2.1	LIBRARY	18-10
18.5.2.2	NAME	18-10
18.5.2.3	LANGUAGE	18-11
18.5.2.4	CALLING STANDARD.....	18-11
18.5.2.5	WITH CONTEXT	18-11
18.5.2.6	PARAMETERS	18-11
18.5.2.7	AGENT IN	18-11
18.6	Publishing Java Class Methods.....	18-11
18.7	Publishing External C Procedures.....	18-12

18.8	Locations of Call Specifications	18-12
18.8.1	Example: Locating a Call Specification in a PL/SQL Package.....	18-13
18.8.2	Example: Locating a Call Specification in a PL/SQL Package Body	18-13
18.8.3	Example: Locating a Call Specification in an ADT Specification	18-13
18.8.4	Example: Locating a Call Specification in an ADT Body	18-14
18.8.5	Example: Java with AUTHID.....	18-14
18.8.6	Example: C with Optional AUTHID.....	18-14
18.8.7	Example: Mixing Call Specifications in a Package	18-14
18.9	Passing Parameters to External C Procedures with Call Specifications	18-15
18.9.1	Specifying Data Types.....	18-16
18.9.2	External Data Type Mappings	18-17
18.9.3	Passing Parameters BY VALUE or BY REFERENCE	18-19
18.9.4	Declaring Formal Parameters	18-19
18.9.5	Overriding Default Data Type Mapping.....	18-20
18.9.6	Specifying Properties	18-20
18.9.6.1	INDICATOR.....	18-22
18.9.6.2	LENGTH and MAXLEN	18-22
18.9.6.3	CHARSETID and CHARSETFORM	18-22
18.9.6.4	Repositioning Parameters	18-23
18.9.6.5	SELF.....	18-23
18.9.6.6	BY REFERENCE	18-25
18.9.6.7	WITH CONTEXT.....	18-26
18.9.6.8	Interlanguage Parameter Mode Mappings.....	18-26
18.10	Running External Procedures with CALL Statements	18-26
18.10.1	Preconditions for External Procedures	18-27
18.10.1.1	Privileges of External Procedures	18-27
18.10.1.2	Managing Permissions.....	18-28
18.10.1.3	Creating Synonyms for External Procedures	18-28
18.10.2	CALL Statement Syntax.....	18-28
18.10.3	Calling Java Class Methods.....	18-29
18.10.4	Calling External C Procedures.....	18-29
18.11	Handling Errors and Exceptions in Multilanguage Programs.....	18-30
18.12	Using Service Routines with External C Procedures	18-30
18.12.1	OCIExtProcAllocCallMemory	18-30
18.12.2	OCIExtProcRaiseExcp	18-34
18.12.3	OCIExtProcRaiseExcpWithMsg.....	18-35
18.13	Doing Callbacks with External C Procedures.....	18-36
18.13.1	OCIExtProcGetEnv	18-36
18.13.2	Object Support for OCI Callbacks	18-38
18.13.3	Restrictions on Callbacks.....	18-38
18.13.4	Debugging External C Procedures	18-39
18.13.5	Example: Calling an External C Procedure.....	18-39
18.13.6	Global Variables in External C Procedures.....	18-40
18.13.7	Static Variables in External C Procedures	18-40
18.13.8	Restrictions on External C Procedures	18-41

19 Developing Applications with Oracle XA

19.1	X/Open Distributed Transaction Processing (DTP).....	19-2
19.1.1	DTP Terminology	19-2
19.1.2	Required Public Information	19-5
19.2	Oracle XA Library Subprograms.....	19-5
19.2.1	Oracle XA Library Subprograms.....	19-5
19.2.2	Oracle XA Interface Extensions	19-6
19.3	Developing and Installing XA Applications.....	19-7
19.3.1	DBA or System Administrator Responsibilities.....	19-7
19.3.2	Application Developer Responsibilities	19-8
19.3.3	Defining the <code>xa_open</code> String.....	19-8
19.3.3.1	Syntax of the <code>xa_open</code> String	19-8
19.3.3.2	Required Fields for the <code>xa_open</code> String.....	19-9
19.3.3.3	Optional Fields for the <code>xa_open</code> String.....	19-9
19.3.4	Using Oracle XA with Precompilers.....	19-11
19.3.4.1	Using Precompilers with the Default Database	19-11
19.3.4.2	Using Precompilers with a Named Database.....	19-11
19.3.5	Using Oracle XA with OCI.....	19-12
19.3.6	Managing Transaction Control with Oracle XA	19-13
19.3.7	Examples of Precompiler Applications	19-14
19.3.8	Migrating Precompiler or OCI Applications to TPM Applications	19-14
19.3.9	Managing Oracle XA Library Thread Safety	19-15
19.3.9.1	Specifying Threading in the Open String	19-16
19.3.9.2	Restrictions on Threading in Oracle XA	19-16
19.3.10	Using the DBMS_XA Package	19-16
19.4	Troubleshooting XA Applications.....	19-19
19.4.1	Accessing Oracle XA Trace Files	19-19
19.4.1.1	<code>xa_open</code> String DbgFl.....	19-20
19.4.1.2	Trace File Locations.....	19-20
19.4.2	Managing In-Doubt or Pending Oracle XA Transactions.....	19-20
19.4.3	Using SYS Account Tables to Monitor Oracle XA Transactions.....	19-21
19.5	Oracle XA Issues and Restrictions	19-21
19.5.1	Using Database Links in Oracle XA Applications	19-21
19.5.2	Managing Transaction Branches in Oracle XA Applications.....	19-22
19.5.3	Using Oracle XA with Oracle Real Application Clusters (Oracle RAC)	19-22
19.5.3.1	<code>GLOBAL_TXN_PROCESSES</code> Initialization Parameter	19-23
19.5.3.2	Managing Transaction Branches on Oracle RAC	19-23
19.5.3.3	Managing Instance Recovery in Oracle RAC with DTP Services (10.2)	19-24
19.5.3.4	Global Uniqueness of XIDs in Oracle RAC	19-25
19.5.3.5	Tight and Loose Coupling.....	19-25
19.5.4	SQL-Based Oracle XA Restrictions.....	19-26
19.5.4.1	Rollbacks and Commits	19-26
19.5.4.2	DDL Statements	19-26
19.5.4.3	Session State	19-26
19.5.4.4	EXEC SQL.....	19-26
19.5.5	Miscellaneous Restrictions	19-26

20 Developing Applications with the Publish-Subscribe Model

20.1	Introduction to the Publish-Subscribe Model.....	20-1
20.2	Publish-Subscribe Architecture.....	20-2
20.2.1	Database Events	20-2
20.2.2	Oracle Advanced Queuing.....	20-2
20.2.3	Client Notification	20-2
20.3	Publish-Subscribe Concepts	20-3
20.4	Examples of a Publish-Subscribe Mechanism	20-4

21 Using the Oracle ODBC Driver

21.1	About Oracle ODBC Driver.....	21-1
21.2	For All Users	21-2
21.2.1	Oracle ODBC Driver.....	21-2
21.2.1.1	What is the Oracle ODBC Driver	21-3
21.2.1.2	New and Changed Features.....	21-5
21.2.1.3	Features Not Supported	21-8
21.2.1.4	Files Created by the Installation	21-8
21.2.1.5	Driver Conformance Levels	21-9
21.2.1.6	Known Limitations.....	21-10
21.2.2	Configuration Tasks	21-10
21.2.2.1	Configuring Oracle Net Services.....	21-10
21.2.2.2	Configuring the Data Source	21-10
21.2.2.3	Oracle ODBC Driver Configuration Dialog Box.....	21-11
21.2.3	Modifying the oraodbc.ini File	21-19
21.2.3.1	Reducing Lock Timeout.....	21-19
21.2.4	Connecting to a Data Source	21-20
21.2.4.1	Connecting to an Oracle Data Source.....	21-20
21.2.5	Troubleshooting	21-20
21.2.5.1	Using the Oracle ODBC Driver for the First Time.....	21-21
21.2.5.2	Expired Password.....	21-22
21.3	For Advanced Users	21-22
21.3.1	Creating Oracle ODBC Driver TNS Service Names	21-23
21.3.2	SQL Statements	21-23
21.3.3	Data Types	21-23
21.3.4	Implementation of Data Types (Advanced)	21-23
21.3.5	Limitations on Data Types	21-24
21.3.6	Error Messages.....	21-24
21.4	For Programmers	21-26
21.4.1	Format of the Connection String	21-26
21.4.2	SQLDriverConnect Implementation.....	21-28
21.4.3	Reducing Lock Timeout in a Program.....	21-29
21.4.4	Linking with odbc32.lib (Windows) or libodbc.so (UNIX)	21-29
21.4.5	Obtaining Information About rowids	21-29
21.4.6	Rowids in a WHERE Clause	21-29
21.4.7	Enabling Result Sets	21-29
21.4.8	Enabling EXEC Syntax	21-38

21.4.9	Enabling Event Notification for Connection Failures in an Oracle RAC Environment.... 21-38	21-42
21.4.10	Using Implicit Results Feature Through ODBC.....	21-42
21.4.11	Supported Functionality	21-43
21.4.11.1	API Conformance	21-43
21.4.11.2	Implementation of ODBC API Functions	21-44
21.4.11.3	Implementation of the ODBC SQL Syntax	21-44
21.4.11.4	Implementation of Data Types (Programming).....	21-44
21.4.12	Unicode Support.....	21-45
21.4.12.1	Unicode Support Within the ODBC Environment.....	21-45
21.4.12.2	Unicode Support in ODBC API.....	21-45
21.4.12.3	Unicode Functions in the Driver Manager	21-46
21.4.12.4	SQLGetData Performance	21-46
21.4.12.5	Unicode Samples	21-46
21.4.13	Performance and Tuning	21-53
21.4.13.1	General ODBC Programming Tips	21-53
21.4.13.2	Data Source Configuration Options	21-53
21.4.13.3	DATE and TIMESTAMP Data Types	21-55

22 Using the Identity Code Package

22.1	Identity Concepts	22-1
22.2	What is the Identity Code Package?.....	22-4
22.3	Using the Identity Code Package	22-6
22.3.1	Storing RFID Tags in Oracle Database Using MGD_ID ADT	22-6
22.3.1.1	Creating a Table with MGD_ID Column Type and Storing EPC Tag Encodings in the Column 22-6	22-6
22.3.1.2	Constructing MGD_ID Objects to Represent RFID Tags.....	22-7
22.3.1.3	Inserting an MGD_ID Object into a Database Table	22-9
22.3.1.4	Querying MGD_ID Column Type	22-9
22.3.2	Building a Function-Based Index Using the Member Functions of the MGD_ID Column Type 22-10	22-10
22.3.3	Using MGD_ID ADT Functions	22-10
22.3.3.1	Using the get_component Function with the MGD_ID Object.....	22-10
22.3.3.2	Parsing Tag Data from Standard Representations	22-11
22.3.3.3	Reconstructing Tag Representations from Fields	22-12
22.3.3.4	Translating Between Tag Representations.....	22-12
22.3.4	Defining a Category of Identity Codes and Adding Encoding Schemes to an Existing Category 22-13	22-13
22.3.4.1	Creating a Category of Identity Codes.....	22-13
22.3.4.2	Adding Two Metadata Schemes to a Newly Created Category.....	22-13
22.4	Identity Code Package Types.....	22-17
22.5	DBMS_MGD_ID_UTL Package	22-18
22.6	Identity Code Metadata Tables and Views	22-19
22.7	Electronic Product Code (EPC) Concepts.....	22-21
22.7.1	RFID Technology and EPC v1.1 Coding Schemes	22-21
22.7.2	Product Code Concepts and Their Current Use	22-22
22.7.2.1	Electronic Product Code (EPC).....	22-22

22.7.2.2	Global Trade Identification Number (GTIN) and Serializable Global Trade Identification Number (SGTIN) 22-23	
22.7.2.3	Serial Shipping Container Code (SSCC) 22-24	
22.7.2.4	Global Location Number (GLN) and Serializable Global Location Number (SGLN) 22-24	
22.7.2.5	Global Returnable Asset Identifier (GRAI)..... 22-24	
22.7.2.6	Global Individual Asset Identifier (GIAI)..... 22-24	
22.7.2.7	RFID EPC Network 22-24	
22.8	Oracle Database Tag Data Translation Schema..... 22-24	

23 Understanding Schema Object Dependency

23.1	Overview of Schema Object Dependency	23-1
23.2	Querying Object Dependencies	23-4
23.3	Object Status	23-4
23.4	Invalidation of Dependent Objects.....	23-4
23.4.1	Session State and Referenced Packages.....	23-8
23.4.2	Security Authorization.....	23-8
23.5	Guidelines for Reducing Invalidation	23-8
23.5.1	Add Items to End of Package.....	23-8
23.5.2	Reference Each Table Through a View	23-9
23.6	Object Revalidation.....	23-9
23.6.1	Revalidation of Objects that Compiled with Errors	23-9
23.6.2	Revalidation of Unauthorized Objects	23-10
23.6.3	Revalidation of Invalid SQL Objects.....	23-10
23.6.4	Revalidation of Invalid PL/SQL Objects	23-10
23.7	Name Resolution in Schema Scope	23-10
23.8	Local Dependency Management	23-11
23.9	Remote Dependency Management	23-11
23.9.1	Dependencies Among Local and Remote Database Procedures	23-11
23.9.2	Dependencies Among Other Remote Objects	23-12
23.9.3	Dependencies of Applications	23-12
23.10	Remote Procedure Call (RPC) Dependency Management	23-12
23.10.1	Time-Stamp Dependency Mode	23-12
23.10.2	RPC-Signature Dependency Mode	23-13
23.10.2.1	Changing Names and Default Values of Parameters.....	23-15
23.10.2.2	Changing Specification of Parameter Mode IN	23-15
23.10.2.3	Changing Subprogram Body	23-15
23.10.2.4	Changing Data Type Classes of Parameters.....	23-16
23.10.2.5	Changing Package Types	23-17
23.10.3	Controlling Dependency Mode	23-18
23.10.3.1	Dependency Resolution.....	23-19
23.10.3.2	Suggestions for Managing Dependencies	23-19
23.11	Shared SQL Dependency Management.....	23-19

24 Using Edition-Based Redefinition

24.1	Editions.....	24-2
24.1.1	Editioned and Noneditioned Objects	24-2

24.1.1.1	Name Resolution for Editioned and Noneditioned Objects	24-3
24.1.1.2	Noneditioned Objects That Can Depend on Editioned Objects	24-3
24.1.1.3	Editionable and Noneditionable Schema Object Types.....	24-5
24.1.1.4	Enabling Editions for a User	24-6
24.1.1.5	EDITIONABLE and NONEDITIONABLE Properties.....	24-8
24.1.1.6	Rules for Editioned Objects.....	24-9
24.1.2	Creating an Edition.....	24-10
24.1.3	Editioned Objects and Copy-on-Change.....	24-10
24.1.4	Making an Edition Available to Some Users	24-14
24.1.5	Making an Edition Available to All Users	24-14
24.1.6	Current Edition and Session Edition	24-15
24.1.6.1	Your Initial Session Edition.....	24-15
24.1.6.2	Changing Your Session Edition.....	24-16
24.1.6.3	Displaying the Names of the Current and Session Editions	24-17
24.1.6.4	When the Current Edition Might Differ from the Session Edition	24-17
24.1.7	Retiring an Edition.....	24-18
24.1.8	Dropping an Edition.....	24-19
24.2	Editioning Views.....	24-20
24.2.1	Creating an Editioning View.....	24-21
24.2.2	Partition-Extended Editioning View Names	24-22
24.2.3	Changing the Writability of an Editioning View	24-22
24.2.4	Replacing an Editioning View	24-22
24.2.5	Dropping or Renaming the Base Table.....	24-22
24.2.6	Adding Indexes and Constraints to the Base Table.....	24-22
24.2.7	SQL Optimizer Index Hints	24-23
24.3	Crossedition Triggers	24-23
24.3.1	Forward Crossedition Triggers.....	24-23
24.3.2	Reverse Crossedition Triggers.....	24-24
24.3.3	Crossedition Trigger Interaction with Editions.....	24-24
24.3.3.1	Which Triggers Are Visible.....	24-24
24.3.3.2	What Kind of Triggers Can Fire	24-24
24.3.3.3	Firing Order.....	24-26
24.3.3.4	Crossedition Trigger Execution.....	24-27
24.3.4	Creating a Crossedition Trigger	24-27
24.3.4.1	Coding the Forward Crossedition Trigger Body	24-28
24.3.5	Transforming Data from Pre- to Post-Upgrade Representation.....	24-30
24.3.5.1	Preventing Lost Updates	24-31
24.3.6	Dropping the Crossedition Triggers	24-32
24.4	Displaying Information About EBR Features	24-32
24.4.1	Displaying Information About Editions.....	24-33
24.4.2	Displaying Information About Editioning Views.....	24-33
24.4.3	Displaying Information About Crossedition Triggers	24-34
24.5	Using EBR to Upgrade an Application	24-34
24.5.1	Preparing Your Application to Use Editioning Views.....	24-35
24.5.2	Procedure for EBR Using Only Editions	24-37
24.5.3	Procedure for EBR Using Editioning Views	24-38
24.5.4	Procedure for EBR Using Crossedition Triggers.....	24-39

24.5.5	Rolling Back the Application Upgrade.....	24-40
24.5.6	Reclaiming Space Occupied by Unused Table Columns.....	24-41
24.5.7	Example: Using EBR to Upgrade an Application	24-41
24.5.7.1	Existing Application.....	24-41
24.5.7.2	Preparing the Application to Use Editioning Views.....	24-43
24.5.7.3	Using EBR to Upgrade the Application	24-43

25 Using Transaction Guard

25.1	Problem that Transaction Guard Solves.....	25-1
25.2	Solution that Transaction Guard Provides.....	25-2
25.3	Transaction Guard Concepts and Scope.....	25-3
25.3.1	Logical Transaction Identifier (LTXID)	25-3
25.3.2	At-Most-Once Execution	25-4
25.3.3	Transaction Guard Coverage	25-4
25.3.4	Transaction Guard Exclusions	25-5
25.4	Database Configuration for Transaction Guard	25-5
25.4.1	Configuration Checklist.....	25-5
25.4.2	Transaction History Table	25-6
25.4.3	Service Parameters.....	25-6
25.5	Developing Applications that Use Transaction Guard	25-7
25.5.1	Typical Transaction Guard Usage.....	25-8
25.5.2	Details for Using the LTXID.....	25-8
25.5.3	Connection-Pool LTXID Usage.....	25-9
25.5.4	Additional Requirements for Transaction Guard Development.....	25-10
25.6	Transaction Guard and Its Relationship to Application Continuity	25-10

26 Ensuring Application Continuity

26.1	About Application Continuity	26-2
26.1.1	Key Concepts for Application Continuity	26-3
26.2	Application Continuity Operation and Usage	26-4
26.2.1	How Application Continuity Works for Database Sessions	26-4
26.2.2	Actions for Using Application Continuity for Java	26-6
26.2.2.1	When Is Application Continuity Transparent?.....	26-9
26.2.2.2	Configuring Oracle JDBC for Application Continuity for Java	26-9
26.2.2.3	Configuring Oracle Database for Application Continuity for Java	26-10
26.2.2.4	Registering a Connection Initialization Callback in Application Continuity for Java (optional) 26-11	
26.2.2.5	Delaying the Reconnection in Application Continuity for Java	26-12
26.2.2.6	Using Application Continuity for Planned Outages.....	26-13
26.2.2.7	Running Without Application Continuity.....	26-13
26.2.2.8	Disabling Replay in Application Continuity for Java	26-14
26.2.2.9	Killing or Disconnecting a Session Without Replay.....	26-16
26.2.3	Mutable Objects and Application Continuity	26-16
26.2.4	Session State Consistency	26-18
26.2.4.1	Dynamic Session State Consistency	26-18
26.2.4.2	Static Session State Consistency	26-20
26.3	Potential Side Effects of Application Continuity.....	26-22

26.4 Restrictions and Other Considerations for Application Continuity 26-23

Index

List of Examples

1–1	Creating and Using a Table with Valid Time Support.....	1-10
1–2	Data Change Using Temporal Validity Support.....	1-11
2–1	Setting Session Purity for New Session	2-28
2–2	Setting the Connection Class as HRMS	2-29
2–3	Setting the Connection Class as RECMS	2-29
2–4	Enabling DRCP With :POOLED in Easy Connect String	2-30
2–5	Enabling DRCP With SERVER=POOLED in TNS Connect String	2-30
2–6	DRCP Application	2-31
2–7	Connect String for Deployment in Dedicated Server Mode Without DRCP	2-32
2–8	Connect String for Deployment With DRCP	2-32
2–9	Explicitly Marking Sessions Stateful or Stateless	2-34
3–1	Secure Application Role Procedure to Restrict Access to Business Hours.....	3-2
3–2	Creating a Logon Trigger	3-2
3–3	Creating Program Units with Definer's Rights or Invoker's Rights	3-5
3–4	Granting a Program Unit Creating the INHERIT PRIVILEGES Privilege	3-5
3–5	Loading a Java Class with Definer's Rights	3-6
6–1	LOCK TABLE with SHARE MODE	6-15
6–2	How the Pro*COBOL Precompiler Uses Locks	6-24
6–3	Marking a Package Subprogram as Autonomous	6-38
6–4	AFTER SUSPEND Trigger Handles Suspended Storage Allocation.....	6-40
7–1	Performance Comparison of Three Data Types	7-3
7–2	Displaying Current Date and Time	7-13
7–3	Inserting and Displaying Dates	7-14
7–4	Inserting and Displaying Dates and Times.....	7-15
7–5	Accessing Information in a SYS.ANYDATA Column.....	7-20
7–6	Querying the ROWID Pseudocolumn	7-22
8–1	Enforcing a Phone Number Format with Regular Expressions.....	8-9
8–2	Inserting Phone Numbers in Correct and Incorrect Formats	8-10
8–3	Using Back References to Reposition Characters	8-10
9–1	Function-Based Index for Precomputing Arithmetic Expression.....	9-5
9–2	Function-Based Indexes on Object Column	9-6
9–3	Function-Based Index for Faster Case-Insensitive Searches	9-7
9–4	Function-Based Index for Language-Dependent Sorting	9-7
10–1	Enforcing Business Rules with Constraints	10-2
10–2	Enforcing Business Rules with Both Constraints and Application Code	10-3
10–3	Inserting NULL Values into Columns with NOT NULL Constraints	10-5
10–4	Deferring Constraint Checks.....	10-12
10–5	Defining Constraints with the CREATE TABLE Statement	10-16
10–6	Defining Constraints with the ALTER TABLE Statement	10-16
10–7	Creating Enabled Constraints	10-18
10–8	Creating Disabled Constraints	10-19
10–9	Enabling Existing Constraints	10-19
10–10	Disabling Existing Constraints.....	10-20
10–11	Modifying Constraints	10-21
10–12	Renaming a Constraint.....	10-22
10–13	Dropping Constraints.....	10-23
10–14	Viewing Information About Constraints.....	10-25
11–1	Changing PLSQL_OPTIMIZE_LEVEL	11-4
11–2	Parsing Penalty for Cursor Variable	11-12
11–3	Array Fetching Penalty for Cursor Variable	11-13
11–4	DML Statements that Reference Collections.....	11-16
11–5	SELECT Statements that Reference Collections	11-16
11–6	FOR Loops that Reference Collections and Return DML	11-17
11–7	Invoking a Subprogram Interactively with SQL*Plus.....	11-21

11–8	Creating and Using a Session Variable with SQL*Plus.....	11-21
11–9	Invoking a Subprogram from Within Another Subprogram	11-22
11–10	PL/SQL Function in SQL Expression (Follows Rules).....	11-27
11–11	PRAGMA RESTRICT_REFERENCES.....	11-31
11–12	PRAGMA RESTRICT REFERENCES with TRUST on Invoker	11-32
11–13	PRAGMA RESTRICT REFERENCES with TRUST on Invokee	11-32
11–14	Raising ORA-04068	11-35
11–15	Trapping ORA-04068.....	11-36
12–1	STANDARD and DBMS_STANDARD PL/Scope Identifier Data Available?	12-2
12–2	How Much Space is PL/Scope Data Using?.....	12-4
12–3	USAGE_CONTEXT_ID and USAGE_ID.....	12-6
12–4	Program Unit with Two Identifiers Named p	12-7
13–1	Profiling a PL/SQL Procedure.....	13-3
13–2	Invoking DBMS_HPROF.analyze.....	13-8
13–3	DBMSHP_RUNS Table for Sample PL/SQL Procedure	13-10
13–4	DBMSHP_FUNCTION_INFO Table for Sample PL/SQL Procedure	13-11
13–5	DBMSHP_PARENT_CHILD_INFO Table for Sample PL/SQL Procedure.....	13-11
13–6	Invoking DBMS_HPROF.analyze with Options	13-12
14–1	Creating and Configuring DADs.....	14-12
14–2	Authorizing DADs to be Created or Changed Later.....	14-12
14–3	Determining the Authentication Mode for a DAD	14-13
14–4	Showing the Authentication Mode for All DADs.....	14-14
14–5	Showing DAD Authorizations that Are Not in Effect.....	14-14
14–6	epgstat.sql Script Output for Example 14–1	14-15
14–7	Using HTP Functions to Generate HTML Tags	14-20
14–8	Using HTP.PRINT to Generate HTML Tags.....	14-20
14–9	HTML Drop-Down List	14-21
14–10	Passing Entry-Field Parameters from an HTML Form	14-23
14–11	Sending E-Mail from PL/SQL	14-26
14–12	Retrieving HTTP URL Contents from PL/SQL.....	14-27
15–1	Query to be Registered for Change Notification.....	15-2
15–2	Query Too Complex for QRCN in Guaranteed Mode	15-3
15–3	Query Whose Simplified Version Invalidates Objects	15-4
15–4	Creating a CQ_NOTIFICATIONS\$_REG_INFO Object.....	15-22
15–5	Adding a Query to an Existing Registration.....	15-22
15–6	Creating Server-Side PL/SQL Notification Handler.....	15-26
15–7	Registering a Query	15-27
15–8	Program Listing That Demonstrates Continuous Query Notification.....	15-34
16–1	Retrieving a Lost Row with Oracle Flashback Query.....	16-7
16–2	Restoring a Lost Row After Oracle Flashback Query	16-7
17–1	Pro*C/C++ Application.....	17-16
17–2	Pro*COBOL Application.....	17-17
19–1	xa_open String.....	19-8
19–2	Sample Open String Configuration.....	19-12
19–3	Transaction Started by an Application Server.....	19-14
19–4	Transaction Started by an Application Client.....	19-14
19–5	Using the DBMS_XA Package.....	19-16
23–1	Displaying Dependent and Referenced Object Types.....	23-1
23–2	Schema Object Change that Invalidates Some Dependents	23-3
23–3	View that Depends on Multiple Objects.....	23-4
23–4	Changing Body of Procedure <code>get_hire_date</code>	23-15
23–5	Changing Data Type Class of <code>get_hire_date</code> Parameter	23-17
23–6	Changing Names of Fields in Package Record Type.....	23-17
24–1	Editioned Objects and Copy-on-Change.....	24-11
24–2	Dropping an Editioned Object	24-12

24-3	Creating an Object with the Name of a Dropped Inherited Object.....	24-13
24-4	Current Edition Differs from Session Edition.....	24-17
24-5	Crossedition Trigger that Handles Data Transformation Collisions	24-29
24-6	EBR of Very Simple Procedure	24-37
24-7	Creating the Existing Application	24-41
24-8	Viewing Data in Existing Table	24-42
24-9	Creating an Editioning View for the Existing Table	24-43
24-10	Creating Edition in Which to Upgrade the Application	24-43
24-11	Changing the Table and Replacing the Editioning View	24-44
24-12	Creating and Enabling the Crossedition Triggers.....	24-44
24-13	Applying the Transforms.....	24-47
24-14	Viewing Data in Changed Table.....	24-47
25-1	Adding and Modifying a Service for a Server Pool	25-7
25-2	Adding an Administrator-Managed Service	25-7
25-3	Modifying a Service (PL/SQL)	25-7

List of Figures

6-1	Interaction Between Serializable Transaction and Another Transaction.....	6-27
6-2	Referential Integrity Check.....	6-28
6-3	Transaction Control Flow	6-33
6-4	Possible Sequences of Autonomous Transactions	6-34
6-5	Example: A Buy Order	6-35
6-6	Bank Withdrawal—Sufficient Funds	6-36
6-7	Bank Withdrawal—Insufficient Funds with Overdraft Protection	6-37
6-8	Bank Withdrawal—Insufficient Funds Without Overdraft Protection.....	6-38
10-1	Rows That Violate and Satisfy a UNIQUE Constraint	10-8
10-2	Rows That Violate and Satisfy a FOREIGN KEY Constraint	10-9
14-1	PL/SQL Web Application	14-2
14-2	Processing Client Requests with Embedded PL/SQL Gateway.....	14-5
15-1	Middle-Tier Caching	15-8
15-2	Basic Process of Continuous Query Notification (CQN)	15-9
17-1	The OCI or OCCI Development Process	17-21
18-1	Oracle Database and External Procedures	18-27
19-1	Possible DTP Model.....	19-2
20-1	Oracle Publish-Subscribe Functionality.....	20-2
21-1	Components of the ODBC Model.....	21-2
21-2	Oracle ODBC Driver Architecture.....	21-4
21-3	Oracle ODBC Driver Configuration Dialog Box	21-12
21-4	The Application Options Tab of the Oracle ODBC Driver Configuration Dialog Box	21-13
21-5	The Oracle Options Tab of the Oracle ODBC Driver Configuration Dialog Box	21-15
21-6	The Workarounds Options Tab of the Oracle ODBC Driver Configuration Dialog Box	21-17
21-7	The SQL Server Migration Options Tab of the Oracle ODBC Driver Configuration Dialog Box	21-19
22-1	RFID Code Categories and Their Schemes	22-2
22-2	Oracle Database Tag Data Translation Markup Language Schema.....	22-4
26-1	How Application Continuity Works.....	26-5
26-2	Dynamic Mode: Nontransaction State Can Change During Request	26-19
26-3	Static Mode: Nontransactional State Never Changes During Request.....	26-21

List of Tables

2-1	Table Annotation Result Cache Modes	2-15
2-2	Effective Result Cache Table Mode.....	2-15
2-3	Client Configuration Parameters (Optional)	2-19
2-4	Setting Client Result Cache and Server Result Cache	2-21
2-5	Session Purity and Connection Class Defaults	2-29
6-1	COMMIT Statement Options	6-6
6-2	Examples of Concurrency Under Explicit Locking.....	6-18
6-3	Ways to Display Locking Information.....	6-25
6-4	ANSI/ISO SQL Isolation Levels and Possible Transaction Interactions	6-26
6-5	ANSI/ISO SQL Isolation Levels Provided by Oracle Database	6-26
6-6	Comparison of READ COMMITTED and SERIALIZABLE Transactions.....	6-31
6-7	Possible Transaction Outcomes	6-35
7-1	SQL Character Data Types	7-5
7-2	Range and Precision of Floating-Point Data Types	7-8
7-3	Binary Floating-Point Format Components.....	7-9
7-4	Summary of Binary Format Storage Parameters.....	7-9
7-5	Special Values for Native Floating-Point Formats.....	7-9
7-6	Values Resulting from Exceptions.....	7-11
7-7	SQL Datetime Data Types	7-12
7-8	SQL Conversion Functions for Datetime Data Types	7-16
7-9	Large Objects (LOBs).....	7-18
7-10	Display Types of SQL Functions	7-24
7-11	SQL Data Type Families.....	7-24
8-1	Oracle SQL Pattern-Matching Condition and Functions	8-2
8-2	Pattern-Matching Options for Oracle SQL Pattern-Matching Condition and Functions	8-3
8-3	POSIX Operators in Oracle SQL Regular Expressions	8-5
8-4	POSIX Operators and Multilingual Operator Relationships.....	8-7
8-5	PERL-Influenced Operators in Oracle SQL Regular Expressions	8-8
8-6	Explanation of the Regular Expression Elements in Example 8-1	8-9
8-7	Explanation of the Regular Expression Elements in Example 8-3	8-11
12-1	Identifier Types that PL/Scope Collects.....	12-8
12-2	Usages that PL/Scope Reports	12-9
13-1	Raw Profiler Output File Indicators.....	13-5
13-2	Function Names of Operations that the PL/SQL Hierarchical Profiler Tracks.....	13-6
13-3	PL/SQL Hierarchical Profiler Database Tables.....	13-7
13-4	DBMSHP_RUNS Table Columns	13-8
13-5	DBMSHP_FUNCTION_INFO Table Columns.....	13-9
13-6	DBMSHP_PARENT_CHILD_INFO Table Columns.....	13-10
14-1	Commonly Used Packages in the PL/SQL Web Toolkit	14-3
14-2	Mapping Between mod_plsql and Embedded PL/SQL Gateway DAD Attributes....	14-7
14-3	Mapping Between mod_plsql and Embedded PL/SQL Gateway Global Attributes ..	14-8
14-4	Authentication Possibilities for a DAD.....	14-11
15-1	Continuous Query Notification Registration Options	15-11
15-2	Attributes of CQ_NOTIFICATION\$_REG_INFO	15-19
15-3	Quality-of-Service Flags.....	15-21
15-4	Attributes of CQ_NOTIFICATION\$_DESCRIPTOR.....	15-45
15-5	Attributes of CQ_NOTIFICATION\$_TABLE	15-46
15-6	Attributes of CQ_NOTIFICATION\$_QUERY	15-47
15-7	Attributes of CQ_NOTIFICATION\$_ROW	15-47
16-1	Oracle Flashback Version Query Row Data Pseudocolumns	16-9
16-2	Flashback TRANSACTION_BACKOUT Options.....	16-15
16-3	Static Data Dictionary Views for Flashback Data Archive Files	16-22
17-1	PL/SQL Packages and Their Java Equivalents.....	17-13

18–1	Parameter Data Type Mappings.....	18-16
18–2	External Data Type Mappings	18-17
18–3	Properties and Data Types	18-21
19–1	Required XA Features Published by Oracle Database	19-5
19–2	XA Library Subprograms.....	19-6
19–3	Oracle XA Interface Extensions	19-6
19–4	Required Fields of <code>xa_open</code> string.....	19-9
19–5	Optional Fields in the <code>xa_open</code> String.....	19-9
19–6	TX Interface Functions	19-13
19–7	TPM Replacement Statements	19-15
19–8	Sample Trace File Contents	19-19
19–9	Tightly and Loosely Coupled Transaction Branches.....	19-22
21–1	SQL Functions Not Supported by the Oracle ODBC Driver	21-8
21–2	Files Installed by the Oracle ODBC Driver Kit.....	21-8
21–3	Limitations Imposed on Data Types by the Oracle ODBC Driver and Oracle Database.....	
21–24		
21–4	Error Message Values of Prefixes Returned by the Oracle ODBC Driver	21-25
21–5	Keywords that Can Be Included in the Connection String Argument of the <code>SQLDriverConnect</code> Function Call	21-26
21–6	Keywords Required by the <code>SQLDriverConnect</code> Connection String	21-28
21–7	How Oracle ODBC Driver Implements Specific Functions.....	21-44
21–8	Supported SQL Data Types and the Equivalent ODBC SQL Data Type.....	21-45
22–1	General Structure of EPC Encodings	22-2
22–2	Identity Code Package ADTs	22-18
22–3	<code>MGD_ID</code> ADT Subprograms.....	22-18
22–4	<code>DBMS_MGD_ID_UTL</code> Package Utility Subprograms.....	22-18
22–5	Definition and Description of the <code>MGD_ID_CATEGORY</code> Metadata View	22-20
22–6	Definition and Description of the <code>USER_MGD_ID_CATEGORY</code> Metadata View	22-20
22–7	Definition and Description of the <code>MGD_ID_SCHEME</code> Metadata View	22-20
22–8	Definition and Description of the <code>USER_MGD_ID_SCHEME</code> Metadata View	22-21
23–1	Database Object Status	23-4
23–2	Operations that Cause Fine-Grained Invalidations.....	23-5
23–3	Data Type Classes	23-16
24–1	<code>*_Dictionary</code> Views with Edition Information.....	24-33
24–2	<code>*_Dictionary</code> Views with Editioning View Information.....	24-34
25–1	LTXID Condition or Situation, Application Actions, and Next LTXID to Use.....	25-9
26–1	Example Treatment of Mutable Objects by Products During Replay	26-17

Preface

Oracle Database Development Guide explains topics of interest to experienced developers of databases and database applications. 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

This guide is intended primarily for application developers who are either developing applications or converting applications to run in the Oracle Database environment. This guide might also help anyone interested in database or database application development, such as systems analysts and project managers.

This guide assumes that you are familiar with the concepts and techniques in the "2 Day" guides relevant to your job, such as *Oracle Database 2 Day Developer's Guide* and *Oracle Database 2 Day DBA*. To use this guide most effectively, you also need a working knowledge of:

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

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Related Documents

For more information, see these documents in the Oracle Database 12c Release 1 (12.1) 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 Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*
- *Oracle Database Migration 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

This guide uses these text conventions:

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

Also:

- `*_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*.

- Table names not qualified with schema names are in the sample schema `HR`. For information about the sample schemas, see *Oracle Database Sample Schemas*.

Changes in This Release for Oracle Database Development Guide

Topics:

- [Changes in Oracle Database 12c Release 1 \(12.1.0.2\)](#)
- [Changes in Oracle Database 12c Release 1 \(12.1.0.1\)](#)

Changes in Oracle Database 12c Release 1 (12.1.0.2)

Oracle Database Development Guide documents one new feature for Oracle Database 12c Release 1 (12.1.0.2):

Flashback Database Archive Support for Multitenant Container Databases

In Release 12.1.0.1, Flashback Database Archive was not supported for multitenant container databases (CDBs).

As of Release 12.1.0.2, this restriction is removed (see [Section 16.12, "Multitenant Container Database Restrictions for Oracle Flashback Technology"](#)).

Changes in Oracle Database 12c Release 1 (12.1.0.1)

The following are changes in *Oracle Database Development Guide* for Oracle Database 12c Release 1 (12.1.0.1):

- [New Features](#)
- [Desupported Features](#)
- [Other Changes](#)

New Features

The following features are new in this release:

- [Application Continuity](#)
- [Transaction Guard](#)
- [Temporal Validity Support](#)
- [Can Grant Roles to PL/SQL Packages and Standalone Subprograms](#)
- [SET_NULL_COLUMN_VALUES_TO_EXPR Procedure](#)
- [Objects, Not Types, Are Editioned or Noneditioned](#)
- [Implicit Statement Results](#)

- [ACCESSIBLE BY Clause](#)
- [PL/SQL Functions and Their Parameters Can Have PL/SQL Data Types](#)
- [PL/SQL Functions That Run Faster in SQL](#)
- [Oracle ODBC Driver](#)
- [Client Auto-Tuning](#)

Application Continuity

Application Continuity attempts to mask outages from end users and applications by recovering requests following recoverable outages, unplanned and planned.

Application Continuity performs this recovery beneath the application so that the outage appears to the application as a delayed execution.

Application Continuity masks recoverable outages—those in which requests would succeed if they were re-issued. Examples include system failures, network disconnects, foreground failures, and storage failures.

Application Continuity for Java is available with Oracle Database, JDBC Thin driver, and the Oracle Database connection pools: UCP (Universal Connection Pool) and WebLogic Server Active GridLink.

Application Continuity is transparent for Java EE and Java SE applications that use Oracle JDBC, use Oracle Database connection pools (UCP or WLS Active GridLink), and do not have external actions. For applications with external actions (for example, using autonomous transactions or using UTL_HTTP to issue an SOA call), Application Continuity is still transparent if the application's correctness is preserved when these external actions are replayed after a failure.

For details, see [Chapter 26, "Ensuring Application Continuity."](#)

Transaction Guard

Before Release 12.1.0.1, it was difficult for a database application to recover after an outage, because the commit message that Oracle Database (the server) returned to the application (the client) was not persistent. If the connection between Oracle Database and the application broke or if the database session became unavailable, then the application received a disconnection error message, but that message did not enable the application to answer these questions:

- Was the **in-flight transaction** (the transaction that was running when the connection broke) committed?
- If the in-flight transaction included the invocation of a stored subprogram:
 - Did the subprogram end normally, executing all expected commits and session state changes?
 - Was the subprogram aborted?
 - Is the subprogram still running in Oracle Database, disconnected from the application?

The application could try to determine if the in-flight transaction was committed by using exception code to query the outcome at every possible commit point. Each query had to be specific to the transaction whose outcome it sought. This approach was both impractical (especially after the application was in production) and fallible, because:

- The transaction could be committed immediately after the query.
- If the transaction included the invocation of a stored subprogram:

- Oracle Database could still be running the subprogram, which could commit database changes after the query returned its results to the application.
 - If the subprogram was written in PL/SQL or Java, then the application could not determine if the subprogram had ended normally or aborted.
- An aborted subprogram could have committed database changes without making subsequent session state changes.

After an outage, if an application user resubmitted an in-flight transaction that had been committed, then duplicate transactions resulted.

As of Release 12.1.0.1, the Oracle Database feature **Transaction Guard** ensures that each transaction executes at most once. Its PL/SQL interface, the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure, enables an application to determine the outcome of the in-flight transaction after an outage and then recover any work that was lost due to the outage. For details, see [Chapter 25, "Using Transaction Guard."](#)

Temporal Validity Support

Temporal Validity support in Oracle Database enables you to associate a valid time dimension with a table and to have data be visible depending on its time-based validity, as determined by the start and end dates or time stamps of the period for which a given record is considered valid.

Scenarios where Temporal Validity support can be useful include:

- Information Lifecycle Management (ILM) and any other application where it is important to know when certain data became valid (from the application's perspective) and when it became invalid (if ever)
- Data correction where incorrect data must be retained and marked with the period when it was considered valid, and where the correct data must be visible as currently valid

For more information, see [Section 1.9.4, "Temporal Validity Support."](#)

Can Grant Roles to PL/SQL Packages and Standalone Subprograms

Before Release 12.1.0.1, a definer's rights (DR) unit always ran with the privileges of the definer and an invoker's rights (IR) unit always ran with the privileges of the invoker. If you wanted to create a PL/SQL unit that all users could invoke, even if their privileges were lower than yours, then it had to be a DR unit. The DR unit always ran with all your privileges, regardless of which user invoked it.

As of Release 12.1.0.1, you can grant roles to individual PL/SQL packages and standalone subprograms. Instead of a DR unit, you can create an IR unit and then grant it roles. The IR unit runs with the privileges of both the invoker and the roles, but without any additional privileges that you have.

For more information, see [Section 11.8, "Invoking Stored PL/SQL Subprograms."](#)

`SET_NULL_COLUMN_VALUES_TO_EXPR` Procedure

Before Release 12.1.0.1, when using edition-based redefinition (EBR), transforming the application data from its pre-upgrade representation (in the old edition) to its post-upgrade representation (in the new edition) required an `UPDATE` operation on every row—a very expensive and time-consuming operation.

As of Release 12.1.0.1, you can sometimes invoke the procedure `DBMS_EDITIONS_UTILITIES.SET_NULL_COLUMN_VALUES_TO_EXPR` to use a metadata operation to transform the application data. For more information, see [Section 24.3.5, "Transforming Data from Pre- to Post-Upgrade Representation."](#)

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_EDITIONS_UTILITIES.SET_NULL_COLUMN_VALUES_TO_EXPR procedure

Objects, Not Types, Are Editioned or Noneditioned

Before Release 12.1.0.1, a schema object was editionable if its type was editionable in the database and its owner was editions-enabled. An editions-enabled user could not own a noneditioned object of an editionable type.

If a noneditioned object (such as a table) referred to an editioned object (such as a user-defined type in an editions-enabled schema), then an error occurred. The workaround was to create the object to be referenced in a schema that was not editions-enabled.

As of Release 12.1.0.1:

- A schema object is editionable if its type is editionable in the schema that owns the object and the object has the EDITIONABLE property. An editions-enabled user can own a noneditioned object of a type that is editionable in the database if the type is noneditionable in the schema or the object has the NONEDITIONABLE property.
- Ordinarily, if a noneditioned object refers to an editioned object, then the editioned object is invisible during name resolution, and a "no such object" error occurs.
- A noneditioned object that can specify an edition to search for editioned objects during name resolution—an **evaluation edition**—*can* depend on editioned objects.

For more information, see [Section 24.1.1, "Editioned and Noneditioned Objects."](#)

Implicit Statement Results

Before Release 12.1.0.1, a PL/SQL stored subprogram returned result sets from SQL queries explicitly, through OUT REF CURSOR parameters, and the client program that invoked the subprogram had to bind to those parameters explicitly to receive the result sets.

As of Release 12.1.0.1, a PL/SQL stored subprogram can return query results to its client implicitly, using the PL/SQL package DBMS_SQL instead of OUT REF CURSOR parameters. This technique makes it easy to migrate applications that rely on the implicit return of query results from stored subprograms from third-party databases to Oracle Database. For more information, see [Section 11.4.5.3, "Returning Query Results Implicitly."](#)

ACCESSIBLE BY Clause

You might implement a database application as several PL/SQL packages—one package that provides the application programming interface (API) and helper packages to do the work. Ideally, only the API is accessible to clients.

Also, you might create a utility package to provide services to only some other PL/SQL units in the same schema. Ideally, the utility package is accessible only to the intended PL/SQL units.

Before Release 12.1.0.1, PL/SQL could not prevent clients from using items exposed in helper packages. To isolate these items, you had to use relational database management system (RDBMS) security features. Some application deployment schemes made RDBMS security features hard to use.

As of Release 12.1.0.1, each of these statements has an optional ACCESSIBLE BY clause that lets you specify a "white list" of PL/SQL units that can access the PL/SQL unit that you are creating or altering:

- CREATE FUNCTION, described in *Oracle Database PL/SQL Language Reference*
- CREATE PACKAGE, described in *Oracle Database PL/SQL Language Reference*
- CREATE PROCEDURE, described in *Oracle Database PL/SQL Language Reference*
- CREATE TYPE, described in *Oracle Database PL/SQL Language Reference*
- ALTER TYPE, described in *Oracle Database PL/SQL Language Reference*

The ACCESSIBLE BY clause supplements the standard Oracle Database security mechanisms. It cannot authorize an otherwise illegal reference.

Changes to the ACCESSIBLE BY clause cause coarse-grained invalidation (see [Table 23-2](#)).

PL/SQL Functions and Their Parameters Can Have PL/SQL Data Types

Before Release 12.1.0.1, a SQL expression could not invoke a PL/SQL function that had a formal parameter or return type that was not a SQL data type. There was one exception to this rule: A formal parameter could have a PL/SQL data type if the corresponding actual parameter was implicitly converted to the data type of the formal parameter.

As of Release 12.1.0.1, this restriction is removed. For remaining restrictions, see [Section 11.9.3, "When PL/SQL Functions Can Appear in SQL Expressions."](#)

PL/SQL Functions That Run Faster in SQL

As of Release 12.1.0.1, two kinds of PL/SQL functions might run faster in SQL:

- PL/SQL functions that are declared and defined in the WITH clauses of SQL SELECT statements, described in *Oracle Database SQL Language Reference*
- PL/SQL functions that are defined with the UDF pragma, described in *Oracle Database PL/SQL Language Reference*.

Oracle ODBC Driver

New and changed features for Release 12.1.0.1 are described in [New Features for Oracle ODBC Driver Release 12.1.0.1.0](#).

Client Auto-Tuning

Client auto-tuning is a feature that transparently optimizes the configuration parameters of OCI client session features of middle-tier applications to gain higher application performance without the need to reprogram your OCI application.

Oracle provides an oraaccess.xml file, a client-side configuration file, that you can use to configure selected OCI parameters as deployment time settings (some of which are accepted programmatically in various OCI API calls, such as statement caching and statement prefetch), thereby allowing OCI behavior to be changed during deployment without modifying the source code that calls OCI.

These settings are provided as connect-string-based deployment settings in the client oraaccess.xml file that overrides manual settings of the user configuration of OCI features.

For complete information about client auto-tuning see [Section 2.9, "OCI Client Statement Cache Auto-Tuning."](#)

Desupported Features

Some features previously described in this guide may be desupported in Release 12.1.0.1. For a list of desupported features, see *Oracle Database Upgrade Guide*.

Other Changes

- [Title](#)
- [New Chapters](#)
- [Removed Chapter](#)
- [Moved Appendix](#)

Title

The title of this guide changed from *Oracle Database Advanced Application Developer's Guide* to *Oracle Database Development Guide*. This title change reflects an expansion and adjustment of the focus of the guide. For example, the new [Part I, "Database Development Fundamentals"](#) introduces important contents and techniques for database developers (both DBAs and database application developers), referring to other chapters and documents for detailed information.

New Chapters

- [Chapter 21, "Using the Oracle ODBC Driver,"](#) has the same content as the Oracle ODBC Driver help file, except that the chapter has no glossary.
- [Chapter 25, "Using Transaction Guard,"](#) explains the major new feature **Transaction Guard**. Transaction Guard ensures *transaction idempotence*—that is, at-most-once execution of a transaction—and uses the logical transaction ID of the transaction to determine the outcome of the last transaction open.
- [Chapter 26, "Ensuring Application Continuity,"](#) explains the major new feature **Application Continuity**. Application Continuity enables the replay, in a transparent and rapid manner, of a transaction against the database after a recoverable error that makes the database session unavailable.

[Chapter 25](#) and [Chapter 26](#) explain how Transaction Guard and Application Continuity are related, and refer to related documentation for implementation details.

Removed Chapter

The chapter "Developing PL/SQL Server Pages (PSP)" was removed, because PSP is obsolete technology. Instead of PSP, use Oracle Application Express.

Note: If you must refer to the removed chapter, see *Oracle Database Advanced Application Developer's Guide* in the Oracle Database 11g Release 2 (11.2) Documentation Library.

Moved Appendix

The appendix "Multithreaded extproc Agent" moved to *Oracle Call Interface Programmer's Guide*, because it is most relevant to the audience of that document (developers of database applications written in C or C++).

Part I

Database Development Fundamentals

This part presents fundamental information for developers of Oracle Database and database applications. The chapters in this part cover mainly high-level concepts, and refer to other chapters and manuals for detailed feature explanations and implementation specifics.

To benefit from the chapters in this part, you need some knowledge of Oracle Database. You must be familiar with the concepts and techniques in the "2 Day" guides relevant to your job, such as *Oracle Database 2 Day Developer's Guide* and *Oracle Database 2 Day DBA*.

Much of the material in this part is adapted from books and articles by Tom Kyte, whose blog is available at <http://asktom.oracle.com>.

Chapters:

- [Chapter 1, "Design Basics"](#)
- [Chapter 2, "Performance and Scalability"](#)
- [Chapter 3, "Security"](#)
- [Chapter 4, "High Availability"](#)
- [Chapter 5, "Advanced PL/SQL Features"](#)

Design Basics

This chapter explains several important design goals for database developers. This chapter and the others in [Part I, "Database Development Fundamentals"](#) cover mainly high-level concepts, and refer to other chapters and manuals for detailed feature explanations and implementation specifics.

To benefit from these chapters, you need some knowledge of Oracle Database. You must be familiar with the concepts and techniques in the "2 Day" guides relevant to your job, such as *Oracle Database 2 Day Developer's Guide* and *Oracle Database 2 Day DBA*.

Topics:

- [Design for Performance](#)
- [Design for Scalability](#)
- [Design for Extensibility](#)
- [Design for Security](#)
- [Design for Availability](#)
- [Design for Portability](#)
- [Design for Diagnosability](#)
- [Design for Special Environments](#)
- [Features for Special Scenarios](#)

1.1 Design for Performance

The key to database and application performance is design, not tuning. While tuning is quite valuable, it cannot make up for poor design. Your design must start with an efficient data model, well-defined performance goals and metrics, and a sensible benchmarking strategy. Otherwise, you will encounter problems during implementation, when no amount of tuning will produce the results that you could have obtained with good design. You might have to redesign the system later, despite having tuned the original poor design.

For an overview of designing databases and applications for performance, see [Chapter 2, "Performance and Scalability."](#)

See Also: *Oracle Database Performance Tuning Guide* and *Oracle Database SQL Tuning Guide* for more information

1.2 Design for Scalability

Scalability is the ability of a system to perform well as its load increases. Load is a combination of number of data volumes, number of users, and other relevant factors. To design for scalability, you must use an effective benchmarking strategy, appropriate application development techniques (such as bind variables), and appropriate Oracle Database architectural features like shared server connections, clustering, partitioning, and parallel operations.

For more information about scalability, see [Chapter 2, "Performance and Scalability"](#).

See Also: *Oracle Database 2 Day Developer's Guide*, especially the section about building scalable applications

1.3 Design for Extensibility

Extensibility is the ease with which a database or database application accommodates future growth. The more extensible the database or application, the easier it is to add or change functionality with minimal impact on existing functionality.

Note: Extensibility differs from **forward compatibility**, the ability of an application to accept data from a future version of itself and use only the data that it was designed to accept.

For example, suppose that an early version of an application processes only text and a later version of the same application processes both text and graphics. If the early version can accept both text and graphics, and ignore the graphics and process the text, then it is forward-compatible. If the early version can be upgraded to process both text and graphics, then it is extensible. The easier it is to upgrade the application, the more extensible it is.

To maximize extensibility, you must design it into your database and applications by including mechanisms that allow enhancement without major changes to infrastructure. Early versions of the database or application might not use these mechanisms, and perhaps no version will ever use all of them, but they are essential to easy maintenance and avoiding early obsolescence.

Topics:

- [Data Cartridges](#)
- [External Procedures](#)
- [User-Defined Functions and Aggregate Functions](#)
- [Object-Relational Features](#)

1.3.1 Data Cartridges

Data cartridges extend the capabilities of the Oracle Database server by taking advantage of Oracle Extensibility Architecture framework. This framework lets you capture business logic and processes associated with specialized or domain-specific data in user-defined data types. Data cartridges that provide new behavior without using additional attributes can do so with packages rather than user-defined data types. With either user-defined types or packages, you determine how the server interprets, stores, retrieves, and indexes the application data. Data cartridges package

this functionality, creating software components that plug into a server and extend its capabilities into a new domain, making the database itself extensible.

You can customize the indexing and query optimization mechanisms of an extensible database management system and provide specialized services or more efficient processing for user-defined business objects and rich types. When you register your implementations with the server through extensibility interfaces, you direct the server to implement your customized processing instructions instead of its own default processes.

See Also: *Oracle Database Data Cartridge Developer's Guide* for more information about data cartridges

1.3.2 External Procedures

External procedures are highly extensible because you can enhance their functionality without affecting their invokers. The reason is that the call specification of an external procedure, which has all the information needed to invoke it, is separate from the body of the procedure, which has the implementation details. If you change only the body, and not the specification, then invokers are unaffected.

See Also: [Chapter 18, "Developing Applications with Multiple Programming Languages,"](#) for more information about external procedures

1.3.3 User-Defined Functions and Aggregate Functions

User-defined PL/SQL functions that can appear in SQL statements or expressions can extend the functionality of SQL.

User-defined aggregate functions are part of the Oracle Extensibility Architecture framework.

See Also:

- [Section 11.9](#) for information about invoking user-defined PL/SQL functions in SQL statements and expressions
- [Oracle Database Data Cartridge Developer's Guide](#) for information about user-defined aggregate functions

1.3.4 Object-Relational Features

The object relational features of Oracle Database are the user-defined Abstract Data Types (ADTs). ADTs are highly extensible because you can enhance their functionality without affecting their invokers. The reason is that the call specification of an external procedure, which has all the information needed to invoke it, is separate from the body of the procedure, which has the implementation details. If you change only the body, and not the specification, then invokers are unaffected.

See Also:

- [Oracle Database PL/SQL Language Reference](#) for information about the CREATE TYPE statement
- [Oracle Database PL/SQL Language Reference](#) for information about the CREATE TYPE BODY statement
- [Oracle Database Object-Relational Developer's Guide](#) for more information about object relational features

1.4 Design for Security

Database security involves a wide range of potential activities, including:

- Designing and implementing security policies to protect the data of an organization, users, and applications from accidental, inappropriate, or unauthorized actions
- Creating and enforcing policies and practices of auditing and accountability for inappropriate or unauthorized actions
- Creating, maintaining, and terminating user accounts, passwords, roles, and privileges
- Developing applications that provide desired services securely in a variety of computational models, leveraging database and directory services to maximize both efficiency and ease of use

For information about some important considerations and techniques for providing security, see [Chapter 3, "Security."](#)

See Also: *Oracle Database Security Guide* for more information

1.5 Design for Availability

Availability is the degree to which an application, service, or function is accessible on demand. A system designed for high availability provides uninterrupted computing services during essential time periods, during most hours of the day throughout the year, with minimal downtime for operations such as upgrading the system's hardware or software. The main characteristics of a highly available system are:

- Reliability
- Recoverability
- Timely error detection
- Continuous operation

For information about some important considerations and techniques for providing high availability, see [Chapter 4, "High Availability."](#)

See Also: *Oracle Database High Availability Overview* and the other manuals in the High Availability group in the Oracle Database documentation library

1.6 Design for Portability

While PL/SQL is not designed for portability between Oracle Database and third-party databases, it is highly portable across operating systems and languages. Most programming languages can invoke PL/SQL, and PL/SQL is implemented consistently on every platform that supports Oracle Database, including Macintosh, Linux, and Windows. If you develop PL/SQL applications on one platform, you can be highly confident that they will work consistently on all other platforms.

PL/SQL stored procedures provide some application portability across multiple databases. Although using stored procedures written in the language of a given vendor may seem to tie you to that vendor to some extent, stored procedures make the application's visual component (user interface) and application logic portable. The data logic is encoded optimally for the database on which the application runs.

Because the data logic is hidden in stored procedures, you can use the vendor's extensions and features to optimize the data layer.

When developed and deployed on a database, the application can stay deployed on that database forever. If the application is moved to another database, the visual component and application logic can move independently of the data logic in the stored procedures, which simplifies the move. (Reworking the application in combination with the move complicates the move.)

See Also: *Oracle Database PL/SQL Language Reference* for conceptual, usage, and reference information about PL/SQL

1.7 Design for Diagnosability

Oracle Database includes a fault diagnosability infrastructure for preventing, detecting, diagnosing, and resolving database problems. Problems include critical errors such as code bugs, metadata corruption, and customer data corruption. The goals of the diagnosability infrastructure are to detect problems proactively, limit damage and interruptions after a problem is detected, reduce the time required to diagnose and resolve problems, and simplify any possible interaction with Oracle Support.

Automatic Diagnostic Repository (ADR) is a file-based repository that stores database diagnostic data such as trace files, the alert log, and Health Monitor reports. ADR is located outside the database, which enables Oracle Database to access and manage ADR when the physical database is unavailable.

See Also:

- *Oracle Database Concepts* for an overview of diagnostic files
- *Oracle Database Administrator's Guide* for detailed information about the Oracle Database fault diagnosability infrastructure.

1.8 Design for Special Environments

This topic introduces designing for the following special database and application environments:

- [Data Warehousing](#)
- [Online Transaction Processing \(OLTP\)](#)

1.8.1 Data Warehousing

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data, but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources. This strategy helps the organization maintain historical records and analyze the data to better understand and improve its business.

In addition to a relational database, a data warehouse environment can include:

- An extraction, transportation, transformation, and loading (ETL) solution
- Statistical analysis
- Reporting
- Data mining capabilities

- Client analysis tools
- Applications that manage the process of gathering data; transforming it into useful, actionable information; and delivering it to business users

Data warehousing systems typically:

- Use many indexes
- Use some (but not many) joins
- Use denormalized or partially denormalized schemas (such as a star schema) to optimize query and analytical performance
- Use derived data and aggregates
- Have workloads designed to accommodate ad hoc queries and data analysis

Because you might not know the workload of your data warehouse in advance, you must optimize the data warehouse to perform well for a wide variety of possible query and analytical operations.

- Are updated regularly (nightly or weekly) by the ETL process, using bulk data modification techniques

The end users of a data warehouse do not directly update the data warehouse except when using analytical tools (such as data mining) to make predictions with associated probabilities, assign customers to market segments, and develop customer profiles.

See Also: *Oracle Database Data Warehousing Guide* for information about data warehousing, including a comparison with online transaction processing (OLTP)

1.8.2 Online Transaction Processing (OLTP)

Online transaction processing (OLTP) systems are optimized for fast and reliable transaction handling. Compared to data warehouse systems, most OLTP interactions involve a relatively small number of rows, but a larger group of tables. In OLTP systems, performance requirements require that historical data be frequently moved to an archive.

OLTP systems typically:

- Use few indexes.
- Use many joins.
- Use fully normalized schemas to optimize update, insert, and delete performance, and to guarantee data consistency.
- Rarely use derived data and aggregates.
- Have workloads consisting of predefined operations.
- Have users routinely issuing individual data modification statements to the database, so that the OLTP database always reflects the current state of each transaction.

See Also: See *Oracle Database Concepts* for more information, including links to manuals with detailed information

1.9 Features for Special Scenarios

This topic introduces Oracle Database features that are particularly useful in scenarios that involve very large databases and the need for high performance.

Topics:

- [SQL Analytic Functions](#)
- [Materialized Views](#)
- [Partitioning](#)
- [Temporal Validity Support](#)

1.9.1 SQL Analytic Functions

A **SQL analytic function** computes an aggregate value based on a group of rows. A SQL analytic function differs from an aggregate function in that it returns multiple rows for each group. For each row, a window of rows is defined. The window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time.

SQL analytic functions bring to set-oriented SQL the ability to use array semantics on result sets. They provide coding efficiency, because they enable concise, straightforward coding of logic that is otherwise cumbersome or impossible. They also provide processing efficiency, because they are integral to Oracle Database and use internal optimizations.

A typical use of analytic functions is to retrieve the most current information in a table. For example, a query of the following form returns information from the row with the most recent update time for each customer with records in a table:

```
SELECT ... FROM my_table t1
  WHERE upd_time = ( SELECT MAX(UPD _TIME)
                      FROM my_table t2
                     WHERE t2.cust_id = t1.cust_id );
```

The preceding query uses a correlated subquery to find the MAX(UPD _TIME) by cust _id, record by record. Therefore, the correlated subquery could be evaluated once for each row in the table. If the table has very few records, performance may be adequate; if the table has tens of thousands of records, the cumulative cost of repeatedly executing the correlated subquery is high.

The following query makes a single pass on the table and computes the maximum UPD_TIME during that pass. Depending on various factors, such as table size and number of rows returned, the following query may be much more efficient than the preceding query:

```
SELECT ...
  FROM ( SELECT t1.*,
               MAX(UPD_TIME) OVER (PARTITION BY cust _id) max_time
                     FROM my_table t1
               )
 WHERE upd_time = max_time;
```

The available analytic functions are:

AVG
CORR
COUNT

```
COVAR_POP
COVAR_SAMP
CUME_DIST
DENSE_RANK
FIRST
FIRST_VALUE
LAG
LAST
LAST_VALUE
LEAD
LISTAGG
MAX
MIN
NTH_VALUE
NTILE
PERCENT_RANK
PERCENTILE_CONT
PERCENTILE_DISC
RANK
RATIO_TO_REPORT
REGR_ (Linear Regression) Functions
ROW_NUMBER
STDDEV
STDDEV_POP
STDDEV_SAMP
SUM
VAR_POP
VAR_SAMP
VARIANCE
```

See Also:

- *Oracle Database Concepts* for an overview of SQL analytic functions
- *Oracle Database SQL Language Reference* for syntax and reference information
- *Oracle Database Data Warehousing Guide* for an extensive discussion of SQL for analysis and reporting

1.9.2 Materialized Views

Materialized views are query results that have been stored ("materialized") as schema objects. Like tables and views, materialized views can appear in the `FROM` clauses of queries. Like tables, materialized views can be partitioned (see "[Partitioning](#)" on page 1-9).

Materialized views are used to summarize, compute, replicate, and distribute data. They are useful for pre-answering general classes of questions—users can query the materialized views instead of individually aggregating detail records. Some environments where materialized views are useful are data warehousing, replication, and mobile computing.

Materialized views require time to create and update, and disk space for storage, but these costs are offset by dramatically faster queries. In these respects, materialized views are like indexes, and they are called "the indexes of your data warehouse." Unlike indexes, materialized views can be queried directly (with `SELECT` statements) and sometimes updated with `DML` statements (depending on the type of update needed).

A major benefit of creating and maintaining materialized views is the ability to take advantage of **query rewrite**, which transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped like indexes without invalidating the SQL in the application code.

The following statement creates and populates a materialized aggregate view based on three master tables in the SH sample schema:

```
CREATE MATERIALIZED VIEW sales_mv AS
  SELECT t.calendar_year, p.prod_id, SUM(s.amount_sold) AS sum_sales
  FROM   times t, products p, sales s
  WHERE  t.time_id = s.time_id
  AND    p.prod_id = s.prod_id
  GROUP BY t.calendar_year, p.prod_id;
```

See Also:

- *Oracle Database Concepts* for an overview of materialized views
- *Oracle Database Data Warehousing Guide* to learn how to use materialized views, including the use of query rewrite with materialized views, in a data warehouse

1.9.3 Partitioning

Partitioning is the database ability to physically break a very large table, index, or materialized view into smaller pieces that it can manage independently. Partitioning is similar to parallel processing, which breaks a large process into smaller pieces that can be processed independently.

Each partition is an independent object with its own name and, optionally, its own storage characteristics. Partitioning is useful for many different types of database applications, particularly those that manage large volumes of data. Benefits include increased availability, easier administration of schema objects, reduced contention for shared resources in OLTP systems, and enhanced query performance in data warehouses.

To partition a table, specify the `PARTITION BY` clause in the `CREATE TABLE` statement. `SELECT` and `DML` statements do not need special syntax to benefit from the partitioning.

A common strategy is to partition records by date ranges. The following statement creates four partitions, one for records from each of four years of sales data (2008 through 2011):

```
CREATE TABLE time_range_sales
  ( prod_id      NUMBER(6)
  , cust_id      NUMBER
  , time_id      DATE
  , channel_id   CHAR(1)
  , promo_id     NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
PARTITION BY RANGE (time_id)
(PARTITION SALES_2008 VALUES LESS THAN (TO_DATE('01-JAN-2009', 'DD-MON-YYYY')) ,
 PARTITION SALES_2009 VALUES LESS THAN (TO_DATE('01-JAN-2010', 'DD-MON-YYYY')) ,
 PARTITION SALES_2010 VALUES LESS THAN (TO_DATE('01-JAN-2011', 'DD-MON-YYYY')) ,
```

```

    PARTITION SALES_2011 VALUES LESS THAN (MAXVALUE)
);

```

See Also:

- *Oracle Database Concepts* for an overview of partitions
- *Oracle Database VLDB and Partitioning Guide* for detailed explanations and usage information about partitioning with very large databases (VLDB)

1.9.4 Temporal Validity Support

Temporal Validity Support lets you associate one or more valid time dimensions with a table and have data be visible depending on its time-based validity, as determined by the start and end dates or time stamps of the period for which a given record is considered valid. Examples of time-based validity are the hire and termination dates of an employee in a Human Resources application, the effective date of coverage for an insurance policy, and the effective date of a change of address for a customer or client.

Temporal Validity Support is typically used with Oracle Flashback Technology (described in [Chapter 16](#)), for queries that specify the valid time period in AS OF and VERSIONS BETWEEN clauses. You can also use the DBMS_FLASHBACK_ARCHIVE.ENABLE_AT_VALID_TIME procedure (described in *Oracle Database PL/SQL Packages and Types Reference*) to specify an option for the visibility of table data: all table data (the default), data valid at a specified time, or currently valid data within the valid time period at the session level.

See Also: [Section 16.12, "Multitenant Container Database Restrictions for Oracle Flashback Technology"](#)

Temporal Validity Support is useful in Information Lifecycle Management (ILM) and any other application where it is important to know when certain data becomes valid (from the application's perspective) and when it becomes invalid (if ever).

Note: [Example 1-1](#) and [Example 1-2](#) assume that the user has the privileges to create tables and perform data manipulation language (DML) and (data definition language) DDL operations on them.

[Example 1-1](#) creates a table with Temporal Validity Support, inserts some rows, and issues some queries whose results depend on the valid start and end dates for individual rows.

Example 1-1 Creating and Using a Table with Valid Time Support

```

CREATE TABLE my_emp(
    empno NUMBER,
    last_name VARCHAR2(30),
    start_time TIMESTAMP,
    end_time TIMESTAMP,
    PERIOD FOR user_valid_time (start_time, end_time));

INSERT INTO my_emp VALUES (100, 'Ames', '01-Jan-10', '30-Jun-11');
INSERT INTO my_emp VALUES (101, 'Burton', '01-Jan-11', '30-Jun-11');
INSERT INTO my_emp VALUES (102, 'Chen', '01-Jan-12', null);

```

```
-- Valid Time Queries --

-- AS OF PERIOD FOR queries:

-- Returns only Ames.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Jun-10');

-- Returns Ames and Burton, but not Chen.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Jun-11');

-- Returns no one.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Jul-11');

-- Returns only Chen.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Feb-12');

-- VERSIONS PERIOD FOR ... BETWEEN queries:

-- Returns only Ames.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
    TO_TIMESTAMP('01-Jun-10') AND TO_TIMESTAMP('02-Jun-10');

-- Returns Ames and Burton.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
    TO_TIMESTAMP('01-Jun-10') AND TO_TIMESTAMP('01-Mar-11');

-- Returns only Chen.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
    TO_TIMESTAMP('01-Nov-11') AND TO_TIMESTAMP('01-Mar-12');

-- Returns no one.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
    TO_TIMESTAMP('01-Jul-11') AND TO_TIMESTAMP('01-Sep-11');
```

To add Temporal Validity Support to an existing table without explicitly adding columns, use the `ALTER TABLE` statement with the `ADD PERIOD FOR` clause. For example, if the `CREATE TABLE` statement in [Example 1-1](#) did not create the `START_TIME` and `END_TIME` columns, you could use this statement:

```
ALTER TABLE my_emp ADD PERIOD FOR user_valid_time;
```

The preceding statement adds two hidden columns to the table `MY_EMP`: `USER_VALID_TIME_START` and `USER_VALID_TIME_END`. You can insert rows that specify values for these columns, but the columns do not appear in the output of the `SQL*Plus DESCRIBE` statement, and `SELECT` statements show the data in those columns only if the `SELECT` list explicitly includes the column names.

[Example 1-2](#) uses Temporal Validity Support for data change in the table created in [Example 1-1](#). In [Example 1-2](#), the initial record for employee 103 has the last name Davis. Later, the employee changes his or her last name to Smith. The `END_TIME` value in the original row changes from `NULL` to the day before the change is to become valid. A new row is inserted with the new last name, the appropriate `START_TIME` value, and `END_TIME` set to `NULL` to indicate that it is valid until set otherwise.

Example 1-2 Data Change Using Temporal Validity Support

```
-- Add a record for Davis.
INSERT INTO my_emp VALUES (103, 'Davis', '01-Jan-12', null);

-- To change employee 103's last name to Smith,
```

```
-- first set an end date for the record with the old name.
UPDATE my_emp SET end_time = '01-Feb-12' WHERE empno = 103;

-- Then insert another record for employee 103, specifying the new last name,
-- the appropriate valid start date, and null for the valid end date.
-- After the INSERT statement, there are two records for #103 (Davis and Smith).
INSERT INTO my_emp VALUES (103, 'Smith', '02-Feb-12', null);

-- What's the valid information for employee 103 as of today?
SELECT * from my_emp AS OF PERIOD FOR user_valid_time SYSDATE WHERE empno = 103;

-- What was the valid information for employee 103 as of a specified date?

-- First, as of a date after the change to Smith.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Jul-12')
WHERE empno = 103;

-- Next, as of a date before the change to Smith.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('20-Jan-12')
WHERE empno = 103;
```

See Also:

- CREATE TABLE and ALTER TABLE in *Oracle Database SQL Language Reference* for syntax information about the PERIOD FOR clause for specifying the valid time period name and the start and end time columns
- SELECT in *Oracle Database SQL Language Reference* for syntax information about using the AS OF PERIOD FOR and VERSIONS PERIOD FOR ... BETWEEN clauses in queries
- [Chapter 16](#) for information about using Oracle Flashback Technology
- DBMS_FLASHBACK_ARCHIVE.ENABLE_AT_VALID_TIME in *Oracle Database PL/SQL Packages and Types Reference*
- Chapter about Information Lifecycle Management (ILM) in *Oracle Database VLDB and Partitioning Guide*, especially the section about temporal validity

Performance and Scalability

This chapter explains some techniques for designing performance and scalability into the database and database applications.

Topics:

- [Performance Strategies](#)
- [Tools for Performance](#)
- [Monitoring Database Performance](#)
- [Using Instrumentation](#)
- [Testing for Performance](#)
- [Using Bind Variables](#)
- [Using Client Result Cache](#)
- [Statement Caching](#)
- [OCI Client Statement Cache Auto-Tuning](#)
- [Client-Side Deployment Parameters](#)
- [Using Query Change Notification](#)
- [Using Database Resident Connection Pool](#)
- [Using Runtime Connection Load Balancing](#)

2.1 Performance Strategies

Topics:

- [Designing Your Data Model to Perform Well](#)
- [Setting Performance Goals \(Metrics\)](#)
- [Benchmarking Your Application](#)

2.1.1 Designing Your Data Model to Perform Well

This topic briefly describes the important concepts of data modeling. Entire books about this subject are available; refer to them for details and further guidance.

Design your data model for optimal performance of the most important and frequent queries, following this basic procedure:

1. [Analyze the Data Requirements of the Application](#)

2. [Create the Database Design for the Application](#)
3. [Implement the Database Application](#)
4. [Maintain the Database and Database Application](#)

2.1.1.1 Analyze the Data Requirements of the Application

Analyze the data requirements of your application by following this basic procedure:

1. Collect data.

Interview people to learn about the business, the nature of the application, who uses information and how, and the expectations of end users. Collect business documents—personnel forms, invoice forms, order forms, and so on—to learn how the business uses information.

2. Analyze the collected data.

This bottom-up process includes normalization of the data, entity-relationship modeling, and transaction analysis.

3. Do a functional analysis of the data.

The end result of this top-down process is a data flow diagram that identifies the main process blocks and how data flows into and out of them over its life time.

2.1.1.2 Create the Database Design for the Application

Create the database design by following this basic procedure:

1. Create the logical design.
2. Translate the logical design into the physical design.

Topics:

- [Create the Logical Design](#)
- [Create the Physical Design](#)

See Also: *Oracle Database Performance Tuning Guide* for more information about designing and developing for performance

2.1.1.2.1 Create the Logical Design The logical design is a graphical representation of the database. The logical design models both relationships between database objects and transaction activity of the application. Effective logical design considers the requirements of different users who must own, access, and update data.

To model relationships between database objects:

1. Translate the data requirements into data items (columns).
2. Group related columns into tables.
3. Map relationships among columns and tables, determining primary and foreign key attributes for each table.
4. Normalize the tables to minimize redundancy and dependency.

To model transaction activity:

- Know the most common transactions and those that users consider most important.
- Trace transaction paths through the logical model.

- Prototype transactions in SQL and develop a volume table that indicates the size of your database.
- Determine which tables are accessed by which users in which sequences, which transactions read data, and which transactions write data.
- Determine whether the application mostly reads data or mostly writes data.

2.1.1.2.2 Create the Physical Design

The physical design is the implementation of the logical design on the physical database.

Because the logical design integrates the information about tables and columns, the relationships between and among tables, and all known transaction activity, you know how the database stores data in tables and creates other structures, such as indexes.

Using this knowledge, create scripts that use SQL data definition language (DDL) to create the schema definition, define the database objects in their required sequence, define the storage requirements to specification, and so on.

2.1.1.3 Implement the Database Application

Implement the database application by following this basic procedure:

1. Implement the application in a test environment.

In a test environment that is as similar as possible to the production environment, run the scripts that implement the physical database design. Load the data into the physical schema. Select the programming language in which to develop the application, develop the user interface, create and test the transactions, and so on.

2. Ensure that the application runs to specification.

Ensure that all components are exercised, the application is fully operational, and the database features that the application uses are optimally configured. For information about tools for testing performance, see [Section 2.2](#).

3. Run benchmark tests on the application.

Benchmark tests determine whether the application performs as expected under various workloads (including peak activity) with simulated real-time operations, such as adding data and users. Ensure that the application scales well. For information about benchmarking your application, see [Section 2.1.3](#).

If the application does not meet the benchmarks, tune your SQL statements to perform optimally, first with no workload and then with increasing workloads.

See Also:

- *Oracle Database SQL Tuning Guide* for more information about SQL tuning
- *Oracle Database 2 Day + Performance Tuning Guide* for more information about SQL tuning
- *Oracle Database Performance Tuning Guide* for more information about workload testing, modeling, and implementation

4. Implement the application in the production environment.

See Also: *Oracle Database Performance Tuning Guide* for more information about deploying new applications

2.1.1.4 Maintain the Database and Database Application

Maintaining the database, the database application, and the operating system are on-going tasks for the database administrator, the application developer, and the system administrator, respectively. The resources that the business allocates to maintenance depend on the importance of the database and the database application, its growth potential, the need to accommodate more users, and so on.

If you are responsible for maintenance, you must periodically monitor the system, schedule maintenance periods, and inform users of upcoming maintenance periods. If maintenance periods require down time, schedule them for periods with little or no database activity.

Application maintenance includes fixing bugs, applying patches, and releasing upgrades. Test maintenance work in a test environment to catch and resolve any before implemented it on production systems.

2.1.2 Setting Performance Goals (Metrics)

Start your application development project by setting performance goals (metrics), including:

- Expected number of application users
- Expected number of transactions per second at peak load times
- Expected query response times at peak load times
- Expected number of records for each table per unit of time (such as one day, one month, or one year)

Use these metrics to create benchmark tests.

2.1.3 Benchmarking Your Application

Benchmarks are tests that measure aspects of application performance. Benchmark results either validate application design or raise issues that you can resolve before putting the application into production.

Usually, you first run benchmarks on an isolated single-user system to minimize interference from other factors. Results from such benchmarks provide a performance baseline for the application. For meaningful benchmark results, you must test the application in the environment where you expect it to run.

You can create small benchmarks that measure performance of the most important transactions, compare different solutions to performance problems, and help resolve design issues that could affect performance.

You must develop much larger, more complex benchmarks to measure application performance during peak user loads, peak transaction loads, or both. Such benchmarks are especially important if you expect the user or transaction load to increase over time. You must budget and plan for such benchmarks.

After the application is in production, run benchmarks regularly in the production environment and store their results in a database table. After each benchmark run, compare the previous and new records for transactions that cannot afford performance degradation. Using this method, you isolate issues as they arise. If you wait until users complain about performance, you might be unable to determine when the problem started.

See Also: *Oracle Database Performance Tuning Guide* for more information about benchmarking applications

2.2 Tools for Performance

Topics:

Several tools that report runtime performance information about your application are:

- [DBMS_APPLICATION_INFO Package](#)
- [SQL Trace Facility \(SQL_TRACE\)](#)
- [EXPLAIN PLAN Statement](#)

See Also: *Oracle Database Testing Guide* for more information about tools for tuning the database

2.2.1 DBMS_APPLICATION_INFO Package

Use the DBMS_APPLICATION_INFO package with the SQL Trace facility (described in [Section 2.2.2](#)) and Oracle Trace and to record the names of executing modules or transactions in the database. System administrators and performance tuning specialists can use this recorded information to track the performance of individual modules and for debugging. System administrators can also use this information to track resource use by module.

When you register the application with the database, its name and actions are recorded in the views V\$SESSION and V\$SQLAREA.

The DBMS_APPLICATION_INFO package provides subprograms that set the following columns in the V\$SESSION view:

- MODULE (name of application or package)
- ACTION (name of transaction or packaged subprogram)
- CLIENT_INFO (additional information about the client application, such as initial bind variable values for the current session)

The DBMS_APPLICATION_INFO package also provides subprograms that return information from the preceding V\$SESSION columns for the current session.

You can also use the DBMS_APPLICATION_INFO package to track the progress of commands that take many seconds to display results (such as those that create indexes or update many rows). The DBMS_APPLICATION_INFO package provides a subprogram that stores information about the command in the V\$SESSION_LONGOPS view. The V\$SESSION_LONGOPS view shows when the command started, how far it has progressed, and its estimated time to completion.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_APPLICATION_INFO package
- *Oracle Database Reference* for information about the V\$SESSION view
- *Oracle Database Reference* for information about the V\$SESSION_LONGOPS view
- *Oracle Database Reference* for information about the V\$SQLAREA view

2.2.2 SQL Trace Facility (SQL_TRACE)

Use the SQL Trace facility, SQL_TRACE, to trace all SQL statements and PL/SQL blocks that your application executes. By enabling the SQL Trace facility for a single statement or module and then disabling it after the statement or module runs, you can get trace information for only that statement or module.

For best results, use the SQL Trace facility with TKPROF and the EXPLAIN PLAN statement. The SQL Trace facility provides performance information for individual SQL statements. TKPROF formats the trace file contents in a readable file. The EXPLAIN PLAN statement shows the execution plans chosen by the optimizer and the execution plan for the specified SQL statement if it were executed in the current session (for more information, see [Section 2.2.3](#)).

Use the SQL Trace facility while designing your application, so that you know what you want to trace and how the application performs before putting it into production. If you wait until performance problems develop in the production environment, your application structure might make using the SQL Trace facility almost impossible.

See Also: *Oracle Database SQL Tuning Guide* for more information about SQL_TRACE and TKPROF

2.2.3 EXPLAIN PLAN Statement

When you run a SQL statement, the optimizer generates several possible execution plans, calculates the cost of each plan, and uses the plan with the lowest cost.

Plan cost is based on statistics such as the data distribution and storage characteristics of the tables, indexes, and partitions that the SQL statement accesses. The cost of access paths and join orders is based on estimated use of computer resources such as I/O, CPU, and memory.

When you use the statement EXPLAIN PLAN FOR *statement* before running *statement*, the EXPLAIN PLAN statement stores the execution plan for *statement* in a plan table. By querying the plan table, you can examine the execution plan that the optimizer chose.

The plan table presents the execution plan as a row source tree, which shows the steps that Oracle Database uses to execute the SQL statement. The row source tree shows the order in which the statement references tables, the join method for tables affected by join operations, and data operations such as filtering, sorting, and aggregation. The plan table also contains optimization information, such as the cost and cardinality of each operation, partitioning information (such as the set of accessed partitions), and parallel execution information (such as the distribution method of join inputs). This information provides valuable insight into how and why the optimizer chose the execution plan.

If you have information about the data that the optimizer does not have, you can give the optimizer a hint, which might change the execution plan. If tests show that the hint improves performance, keep the hint in your code.

Note: You must regularly test code that contains hints, because changing database conditions and query performance enhancements in subsequent releases can significantly impact how hints affect performance.

The EXPLAIN PLAN statement shows only how Oracle Database would run the SQL statement when the statement was explained. If the execution environment or explain plan environment changes, the optimizer might use a different execution plan. Therefore, Oracle recommends using SQL plan management to build a SQL plan baseline, which is a set of accepted execution plans for a SQL statement.

First, use the EXPLAIN PLAN statement to see the statement's execution plan. Second, test the execution plan to verify that it is optimal, considering the statement's actual resource consumption. Finally, if the plan is optimal, use SQL plan management.

See Also:

- *Oracle Database SQL Tuning Guide* for more information about the query optimizer
- *Oracle Database SQL Tuning Guide* for more information about query execution plans
- *Oracle Database SQL Tuning Guide* for more information about influencing the optimizer with hints
- *Oracle Database SQL Tuning Guide* for more information about SQL plan management

2.3 Monitoring Database Performance

Oracle Database provides advisors and powerful tools to help you manage and tune your database.

Topics:

- [Automatic Database Diagnostic Monitor \(ADDM\)](#)
- [Monitoring Real-Time Database Performance](#)
- [Responding to Performance-Related Alerts](#)
- [SQL Advisors and Memory Advisors](#)

2.3.1 Automatic Database Diagnostic Monitor (ADDM)

Automatic Database Diagnostic Monitor (ADDM) is an advisor that analyzes data captured in Automatic Workload Repository (AWR). ADDM and AWR are part of the Oracle Diagnostic Pack.

ADDM determines where database performance problems might exist and where they do not exist, and recommends corrections for the former.

ADDM performs its analysis after each AWR snapshot and stores the results in the database. By default, the AWR snapshot interval is 1 hour and the ADDM results retention period is 8 days.

Oracle Enterprise Manager Cloud Control (Cloud Control) displays ADDM Findings on the Database home page. This AWR snapshot data confirms ADDM results, but lacks the analysis and recommendations that ADDM provides. (AWR snapshot data is similar to Statspack data that database administrators used for performance analysis.)

See Also: *Oracle Database 2 Day + Performance Tuning Guide* for more information about configuring ADDM, reviewing ADDM analysis, interpreting ADDM findings, implementing ADDM recommendations, and viewing snapshot statistics using Enterprise Manager

2.3.2 Monitoring Real-Time Database Performance

Using Oracle Enterprise Manager Cloud Control (Cloud Control), you can monitor real-time database performance from the Performance page. From the Performance page, you can access pages that identify performance issues and resolve those issues without waiting for the next ADDM analysis.

See Also: *Oracle Database 2 Day + Performance Tuning Guide* for more information about monitoring real-time database performance

2.3.3 Responding to Performance-Related Alerts

The Database home page in Oracle Enterprise Manager Cloud Control (Cloud Control) displays performance-related alerts generated by the database. You can improve database performance by resolving problems indicated by these alerts. Oracle Database by default enables alerts for tablespace usage, snapshot too old, recovery area low on free space, and resumable session suspended. You can view metrics and thresholds, set metric thresholds, respond to alerts, clear alerts, and set up direct alert email notification. Using this built-in alerts infrastructure allows you to be notified for these special performance-related alerts.

See Also:

- *Oracle Database Administrator's Guide* for more information about using alerts to help you monitor and tune the database and managing alerts
- *Oracle Database 2 Day + Performance Tuning Guide* for more information about monitoring performance alerts

2.3.4 SQL Advisors and Memory Advisors

Oracle Database provides SQL advisors and memory advisors that you can use to help improve database performance.

SQL advisors include SQL Tuning Advisor and SQL Access Advisor, which are part of the Oracle Database Tuning Pack.

SQL Tuning Advisor accepts one or more SQL statements as input and returns specific tuning recommendations. Each recommendation includes a rationale and expected benefit. Recommendations are based on object statistics, new index creation, SQL statement restructuring, and SQL profile creation.

SQL Access Advisor enables you to optimize data access paths of SQL queries by recommending a set of materialized views and view logs, indexes, and partitions for a given SQL workload.

Memory advisors include Memory Advisor, SGA Advisor, Shared Pool Advisor, Buffer Cache Advisor, and PGA Advisor. Memory advisors provide graphical analyses of total memory target settings, SGA and PGA target settings, and SGA component size settings. Use these analyses to tune database performance and to plan for possible situations.

See Also:

- *Oracle Database 2 Day DBA* for more information about using advisors to optimize database performance, including a brief description of each performance advisor
- *Oracle Database 2 Day DBA* for more information about optimizing memory usage with the memory advisors
- *Oracle Database 2 Day + Performance Tuning Guide* for more information about tools for tuning the database
- *Oracle Database SQL Tuning Guide* for more information about SQL Tuning Advisor
- *Oracle Database SQL Tuning Guide* for more information about SQL Access Advisor

2.4 Using Instrumentation

To use instrumentation is to add debug code throughout your application. When enabled, this code generates trace files, which contain information that helps you identify and locate problems. Trace files are especially helpful when debugging multitier applications; they help you identify the problematic tier.

See Also: [Section 2.2.2, "SQL Trace Facility \(SQL_TRACE\)"](#) for more information

2.5 Testing for Performance

When testing an application for performance, follow these guidelines:

- Use Automatic Database Diagnostic Monitor (ADDM) and SQL Tuning Advisor for design validation.

ADDM determines where database performance problems might exist and recommends corrections. For example, if ADDM finds high-load SQL statements, then you can use SQL Tuning Advisor to analyze those statements and provide tuning recommendations.

For more information about ADDM and SQL Tuning Advisor, see [Section 2.3.1](#) and [Section 2.3.4](#), respectively.

See Also: *Oracle Database 2 Day + Performance Tuning Guide* for more information about tuning SQL statements using the SQL Tuning Advisor

- Test with realistic data volumes and distributions.

The test database must contain data representative of the production system. Tables must be fully populated and represent the data volume and cardinality between tables found in the production system. All production indexes must be built and the schema statistics must be populated correctly.

- Test with the optimizer mode to be used in production.

Because Oracle Database research and development focuses on the query optimizer, Oracle re commands using the query optimizer in both the test and production environments.
- Test a single user performance first.

Start testing with a single user on an idle or lightly-used database. If a single user cannot achieve acceptable performance under ideal conditions, then multiple users cannot achieve acceptable performance under real conditions.
- Get an execution plan for each SQL statement.

Verify that the optimizer uses optimal execution plans, and that you understand the relative cost of each SQL statement in terms of CPU time and physical I/O. Identify heavily used transactions that would benefit most from tuning.
- Test with multiple users.

This testing is difficult to perform accurately because user workload and profiles might not be fully quantified. However, you must test DML transactions in a multiuser environment to ensure that there are no locking conflicts or serialization problems.
- Test with a hardware configuration as close as possible to the production system.

Testing on a realistic system is particularly important for network latencies, I/O subsystem bandwidth, and processor type and speed. Testing on an unrealistic system can fail to reveal potential performance problems.
- Measure steady state performance.

Each benchmark run must have a ramp-up phase, where users connect to the database and start using the application. This phase lets frequently cached data be initialized into the cache and lets single-execution operations (such as parsing) be completed before reaching the steady state condition. Likewise, each benchmark run must end with a ramp-down period, where resources are freed from the system and users exit the application and disconnect from the database.

See Also:

- *Oracle Database Performance Tuning Guide* for an overview about performance tuning
- *Oracle Database Performance Tuning Guide* for more information about workload testing, modeling, and implementation

2.6 Using Bind Variables

A bind variable placeholder in a SQL statement or PL/SQL block indicates where data must be supplied at runtime.

Suppose that you want your application to insert data into the table created with this statement:

```
CREATE TABLE test (x VARCHAR2(30), y VARCHAR2(30));
```

Because the data is not known until runtime, you must use dynamic SQL.

The following statement inserts a row into table `test`, concatenating string literals for columns `x` and `y`:

```
INSERT INTO test (x,y) VALUES ( ' ' || REPLACE (x, ' ', ' ')) ,
```

```
... || REPLACE (y, '...', '...') || ...);
```

The following statement inserts a row into table test using bind variables :x and :y for columns x and y:

```
INSERT INTO test (x,y) VALUES (:x, :y);
```

The statement that uses bind variable placeholders is easier to code.

Now consider a dynamic bulk load operation that inserts 1,000 rows into table test using each of the preceding methods.

The method that concatenates string literals uses 1,000 `INSERT` statements, each of which must be hard-parsed, qualified, checked for security, optimized, and compiled. Because each statement is hard-parsed, the number of latches greatly increases. Latches are mutual-exclusion locking mechanisms—serialization devices, which inhibit concurrency.

The method that uses bind variable placeholders uses only one `INSERT` statement. The statement is soft-parsed, qualified, checked for security, optimized, compiled, and cached in a shared pool. The compiled statement from the shared pool is used for each of the 1000 inserts. This statement caching is a very important benefit of using bind variables.

An application that uses bind variable placeholders is more scalable, supports more users, requires fewer resources, and runs faster than an application that uses string concatenation—and it is less vulnerable to SQL injection attacks. If a SQL statement uses string concatenation, an end user can modify the statement and use the application to do something harmful.

You can use bind variable placeholders for input variables in `DELETE`, `INSERT`, `SELECT`, and `UPDATE` statements, and anywhere in a PL/SQL block that you can use an expression or literal. In PL/SQL, you can also use bind variable placeholders for output variables. Binding is used for both input and output variables in nonquery operations.

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about using bind variables to protect your application from SQL injection
- *Oracle Call Interface Programmer's Guide* for more information about using bind variable placeholders in OCI

2.7 Using Client Result Cache

Topics:

- [About Client Result Cache](#)
- [Benefits of Client Result Cache](#)
- [Guidelines for Using Client Result Cache](#)
- [Client Result Cache Consistency](#)
- [Deployment-Time Settings for Client Result Cache](#)
- [Client Result Cache Statistics](#)
- [Validation of Client Result Cache](#)

- [Client Result Cache and Server Result Cache](#)
- [Client Result Cache Demo Files](#)
- [Client Result Cache Compatibility with Previous Releases](#)
- [Where to Find More Information About Client Result Cache](#)

2.7.1 About Client Result Cache

Applications that use Oracle Database drivers and adapters built on OCI libraries—including C, C++, Java (JDBC-OCI), PHP, Python, Ruby, and Perl—can use client result cache to improve response times of repetitive queries.

Client result cache enables client-side caching of SQL query (SELECT statement) result sets in client memory. Because retrieving results from a client process is faster than calling the database and rerunning the query, frequently run queries perform significantly faster when their results are cached. Client result cache also reduces the server CPU time that would have been used to process the query, thereby improving server scalability.

OCI statements from multiple sessions can match the same cached result set in the OCI process memory if they have similar schemas, SQL text, bind values, and session settings. If not, the query execution is directed to the server.

Client result cache is transparent to applications, and its cache of result set data is kept consistent with session or database changes that affect its result set data.

Applications that use client result cache benefit from faster performance for queries that have cache hits. These applications use the cached result sets on clients or middle tiers.

Client result cache works with OCI features such as the OCI session pool, the OCI connection pool, DRCP, and OCI transparent application failover (TAF).

When using client result cache, you must also enable OCI statement caching or cache statements at the application level.

See Also:

- *Oracle Call Interface Programmer's Guide* for information about statement caching
- *Oracle Database JDBC Developer's Guide* for information about JDBC statement caching

2.7.2 Benefits of Client Result Cache

The benefits of client result cache are:

- Because client result cache is on the client, a cache hit causes fetch (`OCIStmtFetch2()`) and execute (`OCIStmtExecute()`) calls to be processed locally, eliminating a server round trip. Eliminating server round trips reduces the use of server resources (such as server CPU and server I/O), significantly improving performance.
- Client result cache is transparent and consistent.
- Client result cache is available to every process, so multiple client sessions can simultaneously use matching cached result sets.
- Client result cache minimizes the need for each OCI application to have its own custom result cache.

- Client result cache transparently manages:
 - Concurrent access by multiple threads, multiple statements, and multiple sessions
 - Invalidation of cached result sets that might have been affected by database changes
 - Refreshing of cached result sets
 - Cache memory management
- Client result cache is automatically available to applications and drivers that use OCI libraries, including JDBC OCI, ODP.Net, OCCI, Pro*C/C++, Pro*COBOL, and ODBC.
- Client result cache uses OCI client memory, which might be less expensive than server memory.
- A local cache on the client has better locality of reference for queries executed by that client.

See Also: `OCISStmtExecute()` and `OCISStmtFetch2()` in *Oracle Call Interface Programmer's Guide*

2.7.3 Guidelines for Using Client Result Cache

You can enable or disable client result cache at three levels, which are, in order of descending precedence:

1. Query

For a specific query, you can enable or disable client result cache with a SQL hint (for details, see [Section 2.7.3.1](#)). To add or change a SQL hint, you must change the application.

2. Table

For all queries on a specific table, you can enable or disable client result cache with a table annotation, without changing the application (for details, see [Section 2.7.3.2](#)).

3. Session

For all queries in your database session, you can enable or disable client result cache with a session parameter, without changing the application (for details, see [Section 2.7.3.3](#)).

Higher-level settings take precedence over lower-level ones. For examples, see [Section 2.7.3.6](#).

Oracle recommends enabling client result cache only for queries on read-only and seldom-read tables (tables that are rarely updated). That is, enable client result cache:

- At query level only for queries of read-only and seldom-read tables
- At table level only for read-only and seldom-read tables
- At session level only when running applications that query only read-only or seldom-read tables

Enabling client result cache for queries that have large result sets or many sets of bind values can use a large amount of client result cache memory. Each set of bind values (for the same SQL text) creates a different cached result set.

When client result cache is enabled for a query, its result sets can be cached on the client, server, or both. Client result cache can be enabled even if the server result cache (which is enabled by default) is disabled.

For OCI, the first `OCISqlStatement::execute()` call of every OCI statement handle call always goes to the server even if there might be a valid cached result set. An `OCISqlStatement::execute()` call must be made for each statement handle to be able to match a cached result set. Oracle recommends that applications either have their own statement caching for OCI statement handles or use OCI statement caching so that `OCISqlStatement::prepare2()` can return an OCI statement handle that has been executed once. Multiple OCI statement handles, from the same or different sessions, can simultaneously fetch data from the same cached result set.

For OCI, for a result set to be cached, the `OCISqlStatement::execute()` or `OCISqlStatement::fetch2()` calls that transparently create this cached result set must fetch rows until ORA-01403 (No Data Found) is returned. Subsequent `OCISqlStatement::execute()` or `OCISqlStatement::fetch2()` calls that match a locally cached result set need not fetch to completion.

See Also: `OCISqlStatement::execute()`, `OCISqlStatement::prepare2()`, and `OCISqlStatement::fetch2()` in *Oracle Call Interface Programmer's Guide*

Topics:

- [SQL Hints](#)
- [Table Annotation](#)
- [Session Parameter](#)
- [Effective Table Result Cache Mode](#)
- [Displaying Effective Table Result Cache Mode](#)
- [Result Cache Mode Use Cases](#)
- [Queries Never Result Cached in Client Result Cache](#)

2.7.3.1 SQL Hints

The SQL hint `RESULT_CACHE` or `NO_RESULT_CACHE` applies to a single query, for which it enables or disables client result cache. These hints take precedence over both table annotations and the `RESULT_CACHE_MODE` server initialization parameter.

For OCI, the SQL hint `/*+ result_cache */` or `/*+ no_result_cache */` must be set in SQL text passed to `OCISqlStatement::prepare()` and `OCISqlStatement::prepare2()` calls.

For JDBC OCI, the SQL hint `/*+ result_cache */` or `/*+ no_result_cache */` is included in the query (SELECT statement) as part of the query string.

See Also:

- *Oracle Database SQL Language Reference* for general information about SQL hints
- `OCISqlStatement::prepare()`, `OCISqlStatement::prepare2()` in *Oracle Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide* for information about SQL hints in JDBC

2.7.3.2 Table Annotation

A **table annotation** enables or disables client result cache for all queries on a specific table. A table annotation takes precedence over the `RESULT_CACHE_MODE` server

initialization parameter, but the SQL hints `/*+ RESULT_CACHE */` and `/*+ NO_RESULT_CACHE */` take precedence over a table annotation.

You annotate a table with either the `CREATE TABLE` or `ALTER TABLE` statement, using the `RESULT_CACHE` clause. To enable client result cache, specify `RESULT_CACHE (MODE FORCE)`; to disable it, use `RESULT_CACHE (MODE DEFAULT)`.

[Table 2–1](#) summarizes the table annotation result cache modes.

Table 2–1 Table Annotation Result Cache Modes

Mode	Description
DEFAULT	The default value. Result caching is not determined at the table level. You can use this value to clear any table annotations.
FORCE	If all table names in the query have this setting, then the query result is always considered for caching unless the <code>NO_RESULT_CACHE</code> hint is specified for the query. If one or more tables named in the query are set to <code>DEFAULT</code> , then the effective table annotation for that query is <code>DEFAULT</code> .

See Also:

- *Oracle Database SQL Language Reference* for more information about `RESULT_CACHE` clause of `ALTER TABLE` and `CREATE TABLE`
- *Oracle Database JDBC Developer's Guide* for information about table annotations in JDBC

2.7.3.3 Session Parameter

The session parameter `RESULT_CACHE_MODE` enables or disables client result cache for all queries for your database session. `RESULT_CACHE_MODE` can be overruled for specific tables by table annotations and for specific queries by SQL hints.

You can set `RESULT_CACHE_MODE` in either the server parameter file (`init.ora`) or the `ALTER SESSION` or `ALTER SYSTEM` statement. To enable client result cache, set `RESULT_CACHE_MODE` to `FORCE`; to disable it, set `RESULT_CACHE_MODE` to `MANUAL`.

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

2.7.3.4 Effective Table Result Cache Mode

The effective result cache mode for a table depends on both the table annotation result cache mode and the `RESULT_CACHE_MODE` session parameter setting, as [Table 2–2](#) shows.

Table 2–2 Effective Result Cache Table Mode

Table Annotation Result Cache Mode	<code>RESULT_CACHE_MODE = MANUAL</code>	<code>RESULT_CACHE_MODE = FORCE</code>
FORCE	FORCE	FORCE
DEFAULT	MANUAL	FORCE

When the effective mode is `FORCE`, every query is considered for result caching unless the query has the `NO_RESULT_CACHE` hint, but actual result caching depends on internal restrictions for client and server caches, query potential for reuse, and space available in the client result cache.

The effective table result cache mode `FORCE` is similar to the SQL hint `RESULT_CACHE` in that both are only requests.

2.7.3.5 Displaying Effective Table Result Cache Mode

To display the result cache mode for one or more tables, see the RESULT_CACHE column of a *_TABLES static data dictionary view. For example:

- This query displays the result cache mode for the table T:

```
SELECT result_cache FROM all_tables WHERE table_name = T
```

- This query displays the result cache mode for all relational tables that you own:

```
SELECT table_name, result_cache FROM user_tables
```

If the result cache mode is DEFAULT, then the table is not annotated.

If the result cache mode is FORCE, then the table was annotated with the RESULT_CACHE (MODE FORCE) clause of the CREATE TABLE or ALTER TABLE statement.

If the result cache mode is MANUAL, then the session parameter RESULT_CACHE_MODE was set to MANUAL in either the server parameter file (init.ora) or an ALTER SESSION or ALTER SYSTEM statement.

See Also: *Oracle Database Reference* for more information about *_TABLES static data dictionary views

2.7.3.6 Result Cache Mode Use Cases

The following examples show when SQL hints take precedence over table annotations and the RESULT_CACHE_MODE session parameter.

- If the emp table is annotated with ALTER TABLE emp RESULT_CACHE (MODE FORCE) and the session parameter has its default value, MANUAL, then queries on emp are considered for result caching.

However, results of the query `SELECT /*+ no_result_cache */ empno FROM emp` are not cached, because the SQL hint takes precedence over the table annotation and session parameter.

- If the emp table is not annotated, or is annotated with ALTER TABLE emp RESULT_CACHE (MODE DEFAULT), and the session parameter has its default value, MANUAL, then queries on emp are not result cached.

However, results of the query `SELECT /*+ result_cache */ * FROM emp` are considered for caching, because the SQL hint takes precedence over the table annotation and session parameter.

- If the session parameter RESULT_CACHE_MODE is set to FORCE, and no table annotations or SQL hints override it, then all queries on all tables are considered for query caching.

2.7.3.7 Queries Never Result Cached in Client Result Cache

Results of the following queries are never cached in client result cache (even if the queries include the RESULT_CACHE hint):

- Queries that contain any of the following:
 - Remote object
 - Complex type in the select list
 - PL/SQL function
- Snapshot-based queries

- Flashback queries
 - Queries executed in serializable, read-only transactions
 - Queries of tables on which virtual private database (VPD) policies are enabled
- Such queries might be cached on the database if the server result cache feature is enabled—for more information, see *Oracle Database Concepts*.

2.7.4 Client Result Cache Consistency

Client result cache transparently keeps its result sets consistent with any database or session state changes that affect them.

When a transaction modifies the data or metadata of any database object used to construct a cached result set, invalidation of that cached result set is sent to the client on its next round trip to the server. If the application does no database calls for a period of time, then the client result cache lag setting forces the next `OCIStmtExecute()` call to make a database call to check for such invalidations.

Cached result sets relevant to database validations are immediately invalidated. Any OCI statement handles that are fetching from cached result sets when their invalidations are received can continue fetching from them, but no subsequent `OCIStmtExecute()` calls can match them.

The next `OCIStmtExecute()` call by the process might cache the new result set if the client result cache has space available. Client result cache periodically reclaims unused memory.

If a session has a transaction open, OCI ensures that its queries that reference database objects changed in this transaction go to the server instead of client result cache.

This consistency mechanism ensures that client result cache is always close to committed database changes. If the application has relatively frequent calls involving database round trips due to queries that cannot be cached (DMLs, `OCILob` calls, and so on), then these calls transparently keep client result cache consistent with database changes.

Sometimes, when a table is modified, a trigger causes another table to be modified. Client result cache is sensitive to such changes.

When the session state is altered—for example, when NLS session parameters are modified—query results can change. Client result cache is sensitive to such changes and for subsequent query executions, returns the correct result set. However, client result cache keeps the current cached result sets (and does not invalidate them) so that other sessions in the process can match them. If other processes do not match them, these result sets are "Ruled" after a while. Result sets that correspond to the new session state are cached.

For an application, keeping track of database and session state changes that affect query result sets can be cumbersome and error-prone, but client result cache transparently keeps its result sets consistent with such changes.

Client result cache does not require thread support in the client.

See Also: `OCIStmtExecute()` in *Oracle Call Interface Programmer's Guide*

2.7.5 Deployment-Time Settings for Client Result Cache

When you deploy your application, you can set the following for client result cache (without changing the application):

- Server initialization parameters (see section [Section 2.7.5.1](#))
- Client configuration parameters (see section [Section 2.7.5.2](#))
- Table annotations (see section [Section 2.7.3.2](#))
- RESULT_CACHE_MODE session parameter (see section [Section 2.7.3.3](#))

See Also: [Section 2.10, "Client-Side Deployment Parameters"](#)

2.7.5.1 Server Initialization Parameters

The server initialization parameters to set for client result cache when you deploy your application are:

- [COMPATIBLE](#)
- [CLIENT_RESULT_CACHE_SIZE](#)
- [CLIENT_RESULT_CACHE_LAG](#)

2.7.5.1.1 COMPATIBLE Specifies the release with which Oracle Database must maintain compatibility. To enable client result cache, COMPATIBLE must be at least 11.0.0.0. To enable client result cache for views, COMPATIBLE must be at least 11.2.0.0.

2.7.5.1.2 CLIENT_RESULT_CACHE_SIZE Specifies the maximum size of the client result set cache for each OCI client process. The default value, 0, means that client result cache is disabled. To enable client result cache, set CLIENT_RESULT_CACHE_SIZE to at least 32768 bytes (32 kilobytes (KB)).

If client result cache is enabled on the server by CLIENT_RESULT_CACHE_SIZE, then its value can be overridden by the `sqlnet.ora` configuration parameter `OCI_RESULT_CACHE_MAX_SIZE`. If client result cache is disabled on the server, then `OCI_RESULT_CACHE_MAX_SIZE` is ignored and client result cache cannot be enabled on the client.

Oracle recommends either enabling client result cache for all Oracle Real Application Clusters (Oracle RAC) nodes or disabling client result cache for all Oracle RAC nodes. Otherwise, within a client process, some sessions might have caching enabled and other sessions might have caching disabled (thereby getting the latest results from the server). This combination might present an inconsistent view of the database to the application.

`CLIENT_RESULT_CACHE_SIZE` is a static parameter. Therefore, if you use an `ALTER SYSTEM` statement to change the value of `CLIENT_RESULT_CACHE_SIZE`, you must include the `SCOPE = SPFILE` clause and restart the database before the change will take effect.

The maximum value for `CLIENT_RESULT_CACHE_SIZE` is the least of these values:

- Available client memory
- $((\text{Possible number of result sets to be cached}) * (\text{average size of a row in a result set}) * (\text{average number of rows in a result set}))$
- 2 gigabytes (GB)

If you specify a value greater than 2 GB, then the value is 2 GB.

Note: Do not set the `CLIENT_RESULT_CACHE_SIZE` parameter during database creation, because that can cause errors.

2.7.5.1.3 CLIENT_RESULT_CACHE_LAG Specifies the maximum time in milliseconds that client result cache can lag behind changes in the database that affect its result sets. The default is 3000 milliseconds.

CLIENT_RESULT_CACHE_LAG is a static parameter. Therefore, if you use an ALTER SYSTEM statement to change the value of CLIENT_RESULT_CACHE_LAG, you must include the SCOPE = SPFILE clause and restart the database before the change will take effect.

2.7.5.2 Client Configuration Parameters

Client configuration parameters are optional, but if set, they override the equivalent parameters in the server initialization file `init.ora`.

Client configuration parameters can be set in the `oraaccess.xml` file (see *Oracle Call Interface Programmer's Guide*), the `sqlnet.ora` file, or both. When equivalent parameters are set both files, the `oraaccess.xml` setting takes precedence over the corresponding `sqlnet.ora` setting. When a parameter is not set in `oraaccess.xml`, the process searches for its setting in `sqlnet.ora`.

When a client configuration parameter can be set in both `oraaccess.xml` and `sqlnet.ora`, Oracle recommends setting the parameter in `oraaccess.xml`. However, for a network configuration, `sqlnet.ora` is the primary file, because `oraaccess.xml` does not support network level settings.

Table 2–3 describes the equivalent `oraaccess.xml` and `sqlnet.ora` client configuration parameters.

Table 2–3 Client Configuration Parameters (Optional)

<code>oraaccess.xml</code> Parameter	<code>sqlnet.ora</code> Parameter	Description
<code>max_size</code>	<code>OCI_RESULT_CACHE_MAX_SIZE</code>	Maximum size in bytes for the client result cache for each process. Specifying a size less than 32768 in <code>sqlnet.ora</code> disables client result cache for client processes the read <code>sqlnet.ora</code> .
<code>max_rset_size</code>	<code>OCI_RESULT_CACHE_MAX_RSET_SIZE</code>	Maximum size of any result set in bytes in the client result cache for each process.
<code>max_rset_rows</code>	<code>OCI_RESULT_CACHE_MAX_RSET_ROWS</code>	Maximum size of any result set in rows in the client result cache for each process.

The result cache lag cannot be set on the client.

2.7.6 Client Result Cache Statistics

On round trips to the server from the OCI client, OCI periodically sends client result cache statistics to the server. These statistics, shown in the `CLIENT_RESULT_CACHE_STATS$` view, include number of result sets successfully cached, number of cache hits, and number of cached result sets invalidated. The number of cache misses for queries is at least equal to the number of Create Counts in client result cache statistics. More precisely, the cache miss count equals the number of server executions in server Automatic Workload Repository (AWR) reports.

See Also:

- *Oracle Database Reference* for information about the `CLIENT_RESULT_CACHE_STAT$` view
- *Oracle Database Performance Tuning Guide* to find the client process IDs and session IDs for the sessions doing client result caching

2.7.7 Validation of Client Result Cache

Some ways to validate client result cache are:

- [Measure Execution Times](#)
- [Query V\\$MYSTAT](#)
- [Query V\\$SQLAREA](#)

2.7.7.1 Measure Execution Times

First, measure the execution time of the queries without `RESULT_CACHE` hints. Then add `RESULT_CACHE` hints to the queries and measure the execution time again. The difference in execution times is your performance gain.

2.7.7.2 Query V\$MYSTAT

Note: To query the `V$MYSTAT` view, you must have the `SELECT` privilege on it.

1. Run this query 5 times:

```
SELECT count(*) FROM table_name
```

2. Query `V$MYSTAT`:

```
SELECT * FROM V$MYSTAT
```

3. Run this query 5 times:

```
SELECT /*+ result_cache */ count(*) FROM table_name
```

Because the query results are cached, this step requires fewer round trips between client and server than step 1 did.

4. Query `V$MYSTAT`:

```
SELECT * FROM V$MYSTAT
```

Compare the values of the columns for this query to those for the query in step .2.

Instead of adding the hint to the query in step 3, you can add the table annotation `RESULT_CACHE (MODE FORCE)` to `table_name` at step 3 and then run the query in step 1 a few times.

2.7.7.3 Query V\$SQLAREA

Note: To query the `V$SQLAREA` view, you must have the `SELECT` privilege on it.

1. Run this query 5 times:

```
SELECT count(*) FROM table_name
```

2. Query V\$SQLAREA:

```
SELECT executions, fetches, parse_calls FROM V$SQLAREA
WHERE sql_text LIKE '% FROM table_name'
```

3. Run this query 5 times:

```
SELECT /*+ result_cache */ count(*) FROM table_name
```

4. Query V\$SQLAREA:

```
SELECT executions, fetches, parse_calls FROM V$SQLAREA
WHERE sql_text LIKE '% FROM table_name'
```

Compare the values of the columns executions, fetches, and parse_calls for this query to those for the query in step 2. The difference in execution times is your performance gain.

Instead of adding the hint to the query in step 3, you can add the table annotation RESULT_CACHE (MODE FORCE) to *table_name* at step 3 and then run the query in step 1 a few times.

2.7.8 Client Result Cache and Server Result Cache

Client result cache is different from server result cache. Client result cache caches results of top-level SQL queries in OCI client memory, whereas server result cache caches result sets and query fragments in server SGA memory.

You can enable client result cache independently of server result cache, though they share the result cache SQL hints, table annotations, and session parameter RESULT_CACHE_MODE. [Table 2-4](#) shows the result cache association for result cache parameters, the PL/SQL package DBMS_RESULT_CACHE, and result cache views. For more information about server result cache, see *Oracle Database Concepts*.

Table 2-4 Setting Client Result Cache and Server Result Cache

Parameters, PL/SQL Package, and Database Views	Result Cache Association
CLIENT_RESULT_CACHE_* parameters:	client result cache
<ul style="list-style-type: none"> ■ CLIENT_RESULT_CACHE_SIZE ■ CLIENT_RESULT_CACHE_LAG 	
SQL hints:	client result cache, server result cache
<ul style="list-style-type: none"> ■ RESULT_CACHE ■ NO_RESULT_CACHE 	
sqlnet.ora OCI_RESULT_CACHE* parameters:	client result cache
<ul style="list-style-type: none"> ■ OCI_RESULT_CACHE_MAX_SIZE ■ OCI_RESULT_CACHE_MAX_RSET_SIZE ■ OCI_RESULT_CACHE_MAX_RSET_ROWS 	
CLIENT_RESULT_CACHE_STATS\$ view	client result cache
RESULT_CACHE_MODE parameter	client result cache, server result cache

Table 2–4 (Cont.) Setting Client Result Cache and Server Result Cache

Parameters, PL/SQL Package, and Database Views	Result Cache Association
All other RESULT_CACHE_* parameters (for example, RESULT_CACHE_MAX_SIZE)	server result cache
DBMS_RESULT_CACHE package	server result cache
V\$RESULT_CACHE_* and GV\$RESULT_CACHE_* views (for example, V\$RESULT_CACHE_STATISTICS and GV\$RESULT_CACHE_MEMORY)	server result cache
CREATE TABLE annotation	client result cache, server result cache
ALTER TABLE annotation	client result cache, server result cache

2.7.9 Client Result Cache Demo Files

For OCI applications, demonstration files for client result cache are `cdemoqc.sql`, `cdemoqc.c`, and `cdemoqc2.c` (in the `demo` directory for your operating system).

2.7.10 Client Result Cache Compatibility with Previous Releases

To use client result cache, applications must be relinked with Oracle Database 11g Release 1 (11.1) or later client libraries and be connected to an Oracle Database 11g Release 1 (11.1) or later database server. Client result cache is available to all OCI applications, including JDBC Type II driver, OCCI, Pro*C/C++, and ODP.NET. OCI drivers automatically pass the SQL hint `RESULT_CACHE` to `OCISqlPrepare()` and `OCISqlPrepare2()` calls.

See Also: `OCISqlPrepare()`, `OCISqlPrepare2()` in *Oracle Call Interface Programmer's Guide*

2.7.11 Where to Find More Information About Client Result Cache

More information about client result cache is in the following documentation:

- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle Database Performance Tuning Guide*
- *Oracle Database Concepts*
- *Oracle Database SQL Language Reference* for information about SQL hints and `RESULT_CACHE` clause of `ALTER TABLE`, and `CREATE TABLE`
- *Oracle Database Reference* for information about `RESULT_CACHE_MODE` initialization parameter and `CLIENT_RESULT_CACHE_STAT$` view

2.8 Statement Caching

Statement caching is a feature that establishes and manages a cache of statements for each session. In the server, statement caching lets cursors be used without reparsing the statement, eliminating repetitive statement parsing. You can use statement caching with both connection pooling and session pooling, thereby improving performance and scalability. You can also use statement caching without session pooling in OCI and

without connection pooling in OCCI, in the JDBC interface, and in the ODP.NET interface. You can also use dynamic SQL statement caching in Oracle precompiler applications that rely on dynamic SQL statements, such as Pro*C/C++ and ProCOBOL.

In the JDBC interface, you can enable and disable implicit and explicit statement caching independently of the other—you can use either, neither, or both. Implicit and explicit statement caching share a single cache for each connection. You can also disable implicit caching for a particular statement.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information and guidelines about using statement caching in OCI
- *Oracle C++ Call Interface Programmer's Guide* for more information about statement caching in OCCI
- *Oracle Database JDBC Developer's Guide* for more information about using statement caching
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about using statement caching in ODP.NET applications
- *Oracle Database Programmer's Guide to the Oracle Precompilers, Pro*C/C++ Programmer's Guide, and Pro*COBOL Programmer's Guide* for more information about using dynamic SQL statement caching in precompiler applications that rely on dynamic SQL statements

2.9 OCI Client Statement Cache Auto-Tuning

OCI client statement cache auto-tuning optimizes OCI client session features of middle-tier applications to improve performance without changing your OCI application.

Without auto-tuning, the OCI client statement cache size setting can become suboptimal—for example, when a changing workload causes a different working set of SQL statements. If the size is too low, it causes excess network activity and more parses at the server. If the size is too high, it causes excess memory use. It can be difficult for the client application to keep the cache size optimal.

Auto-tuning solves this potential performance problem by automatically and periodically reconfiguring the OCI statement cache size.

Auto-tuning is achieved by providing a deployment-time setting that provides an option to reconfigure OCI statement caching. These settings are provided as connect-string-based deployment settings in a client `oraaccess.xml` file that overrides programmatic settings to the user configuration of OCI features.

Middle-tier application developers and database administrators (DBAs) can expect reduced time and effort in diagnosing and fixing performance problems with each part of their system using the auto-tuning OCI client statement caching parameter setting.

For information about OCI auto-tuning parameters, implementation, and usage, see *Oracle Call Interface Programmer's Guide*.

2.10 Client-Side Deployment Parameters

Beginning with Oracle Database 12c Release 1 (12.1.0.1), OCI deployment parameters are available in a new configuration file, `oraaccess.xml`, as described in *Oracle Call Interface Programmer's Guide*.

2.11 Using Query Change Notification

Continuous Query Notification (CQN) lets client applications register queries with the database and receive notifications of DML or DDL changes on the objects (object change notification (OCN)) or result set changes associated with the queries (query result change notification (QRCN)). The database publishes notifications when the DML or DDL transaction commits.

A **CQN registration** associates one or more queries with a notification type (OCN or QRCN) and a notification handler. To create a CQN registration, you can use:

- PL/SQL interface

When you use the PL/SQL interface, the notification handler is a server-side PL/SQL stored procedure. PL/SQL registration can be used by nonthreaded languages and systems, such as PHP. For PHP, the PL/SQL listener invokes a PHP callback when it receives the database notification.

- Oracle Call Interface (OCI)

When you use OCI, the notification handler is a client-side C callback procedure.

- Java Database Connectivity (JDBC) interface

When you use the JDBC interface, the JDBC driver creates a registration on the server. The JDBC driver launches a new thread that listens for notifications from the server (through a dedicated channel), converts them to Java events, and then notifies all listeners registered with this registration.

See Also:

- [Chapter 15, "Using Continuous Query Notification \(CQN\)"](#) for a complete discussion of the concepts of this feature and how to use the PL/SQL and OCI interfaces to create CQN registrations
- [Oracle Database JDBC Developer's Guide](#) for information about using JDBC for CQN registration on the server

2.12 Using Database Resident Connection Pool

Database Resident Connection Pool (DRCP) provides a connection pool in the database server for typical web application usage scenarios where the application acquires a database connection, works on the database for a relatively short time, and then releases the connection.

Topics:

- [About Database Resident Connection Pool](#)
- [Configuring DRCP](#)
- [Using JDBC with DRCP](#)
- [Using OCI Session Pool APIs with DRCP](#)
- [Session Purity and Connection Class](#)

- [Starting Database Resident Connection Pool](#)
- [Enabling DRCP](#)
- [Benefiting from the Scalability of DRCP in an OCI Application](#)
- [Benefiting from the Scalability of DRCP in a Java Application](#)
- [Best Practices for Using DRCP](#)
- [Compatibility and Migration](#)
- [DRCP Restrictions](#)
- [Using DRCP with Custom Pools](#)
- [Explicitly Marking Sessions Stateful or Stateless](#)
- [Using DRCP with Oracle Real Application Clusters](#)
- [Using DRCP with Pluggable Databases](#)
- [DRCP with Data Guard](#)

2.12.1 About Database Resident Connection Pool

DRCP pools server processes, each of which is the equivalent of a dedicated server process and database session combined; these are called **pooled servers**. Pooled servers can be shared by multiple applications running on the same or several hosts. A connection broker process manages the pooled servers at the database instance level.

DRCP is a configurable feature, chosen at application runtime. It allows concurrent use of traditional and DRCP-based connection architectures.

DRCP is especially useful for architectures with multiprocess single-threaded application servers (such as PHP and Apache) that cannot do middle-tier connection pooling. DRCP is also very useful in large-scale web deployments where hundreds or thousands of web servers or middle-tiers need database access and client-side pools (even in multithreaded systems and languages such as Java). Using DRCP, the database can scale to tens of thousands of simultaneous connections. If your database web application must scale to large numbers of connections, DRCP is your connection pooling solution.

DRCP complements middle-tier connection pools that share connections between threads in a middle-tier process. DRCP also enables sharing of database connections across middle-tier processes on the same middle-tier host, across multiple middle-tier hosts, and across multiple middle-tiers (web servers, containers) that accommodate applications written in different languages. This sharing significantly reduces the database resources needed to support a large number of client connections, thereby reducing the database tier memory footprint and increasing the scalability of both middle and database tiers. Having a pool of readily available servers also reduces the cost of creating and releasing client connections.

Clients get connections from the DRCP, which is connected to an Oracle Database background process called the connection broker. The connection broker implements the pool functionality and multiplexes pooled servers among persistent inbound connections from the client.

When a client needs database access, the connection broker gets a server process from the pool and gives it to the client. The client is then directly connected to the server. After the server executes the client request, the server process returns to the pool and the connection from the client is restored to the connection broker as a persistent inbound connection from the client process. In DRCP, releasing resources leaves the

session intact, but no longer associated with a connection (server process). Because this session stores its user global area (UGA) in the program global area (PGA), not in the system global area (SGA), a client can reestablish a connection transparently upon detecting activity.

DRCP is typically recommended for applications with a large number of connections. Shared servers are recommended for applications with a medium number of connections and dedicated sessions are recommended for applications with a small number of connections. The threshold sizes depend on the amount of memory available on the database host.

DRCP has these advantages:

- DRCP enables resource sharing among multiple client applications and middle-tier application servers.
- DRCP improves scalability of databases and applications by reducing resource usage on the database host.

Compared to client-side connection pooling and shared servers:

- DRCP provides a direct connection to the database server, furnished by client-side connection pooling (like client-side connection pooling but unlike shared servers).
- DRCP can pool database servers (like client-side connection pooling and shared servers).
- DRCP can pool sessions (like client-side connection pooling but unlike shared servers).
- DRCP can share connections across middle-tier boundaries (unlike client-side connection pooling).

DRCP offers a unique connection pooling solution that addresses scalability requirements in environments requiring large numbers of connections with minimal database resource usage.

See Also: *Oracle Database Concepts* for details about DRCP architecture

2.12.2 Configuring DRCP

DRCP is installed by default, but the DBA must start and configure it, using the `DBMS_CONNECTION_POOL` package. Configuration options include minimum and maximum number of pooled servers, number of connection brokers, maximum number of connections that each connection broker can handle, and so on.

OCI session pool APIs have been extended to interoperate with DRCP.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package
- *Oracle Database Administrator's Guide* for the details of configuring a database for DRCP
- *Oracle Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle Universal Connection Pool for JDBC Developer's Guide*

2.12.3 Using JDBC with DRCP

Beginning with Oracle Database 12c Release 1 (12.1.0.1), Oracle JDBC drivers support DRCP. The DRCP implementation creates a pool on the server side, which is shared across multiple client pools. These client pools use Universal Connection Pool for JDBC. Using Universal Connection Pool significantly lowers memory consumption (because of the reduced number of server processes on the server) and increases the scalability of the Database server.

To track check-in and checkout operations of server-side connections, Java applications must use a client-side pool such as Universal Connection Pool for JDBC or a third-party Java connection pool.

[Section 2.12.6](#) explains how the DBA starts DRCP on the server side.

To enable DRCP on the client side, you must do the following:

- Pass a non-NULL, nonempty `String` value to the connection property `oracle.jdbc.DRCPConnectionClass`.
- Pass `(SERVER=POOLED)` in the long connection string.

You can also specify `(SERVER=POOLED)` in the short URL form as follows:

```
jdbc:oracle:thin:@//<host>:<port>/<service_name>[:POOLED]
```

For example:

```
jdbc:oracle:thin:@//localhost:5221/orcl:POOLED
```

For an example of enabling JDBC on the client side, see the section on enabling DRCP on the client side in *Oracle Database JDBC Developer's Guide*.

By setting the same DRCP Connection class name for all the pooled server processes on the server using the connection property `oracle.jdbc.DRCPConnectionClass`, you can share pooled server processes on the server across multiple connection pools.

In DRCP, you can also apply a tag to a given connection and easily retrieve that tagged connection later. For more information about enabling connection tagging in DRCP, see *Oracle Database JDBC Developer's Guide*.

2.12.4 Using OCI Session Pool APIs with DRCP

The OCI session pool APIs `OCISessionPoolCreate()`, `OCISessionGet()`, and `OCISessionRelease()` interoperate with DRCP.

An OCI application initializes the environment for the OCI session pool for DRCP by invoking `OCISessionPoolCreate()`, which is described in *Oracle Call Interface Programmer's Guide*.

To get a session from the OCI session pool for DRCP, an OCI application invokes `OCISessionGet()`, specifying `OCI_SESSGET_SPOOL` for the mode parameter. For information about `OCISessionGet()`, see *Oracle Call Interface Programmer's Guide*.

To release a session to the OCI session pool for DRCP, an OCI application invokes `OCISessionRelease()`, which is described in *Oracle Call Interface Programmer's Guide*.

To improve performance, the OCI session pool can transparently cache connections to the connection broker. An OCI application can reuse the sessions within which the application leaves sessions of a similar state either by invoking `OCISessionGet()` with the `authInfoOp` parameter set to `OCI_ATTR_CONNECTION_CLASS` and specifying a connection class name or by using the `OCIAuthInfo` handle before invoking `OCISessionGet()`. For details, see [Section 2.12.5](#).

DRCP also supports features offered by the traditional client-side OCI session pool, such as tagging, statement caching, and TAF.

2.12.5 Session Purity and Connection Class

In Oracle Database 11g Release 1 (11.1), OCI introduced two settings that can be specified when obtaining a session using `OCISessionGet()`: session purity and connection class.

Topics:

- [Session Purity](#)
- [Connection Class](#)
- [Session Purity and Connection Class Defaults](#)

2.12.5.1 Session Purity

Session purity specifies whether an OCI application can reuse a pooled session (`OCI_SESSGET_PURITY_SELF`) or must use a new session (`OCI_SESSGET_PURITY_NEW`).

The application can set session purity either on the `OCIAuthInfo` handle before invoking `OCISessionGet()` or in the mode parameter when invoking `OCISessionGet()`.

[Example 2-1](#) shows how a connection pooling application sets up a new session.

Example 2-1 Setting Session Purity for New Session

```
/* OCIAttrSet method */  
  
ub4 purity = OCI_ATTR_PURITY_NEW;  
OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO, &purity, sizeof (purity),  
            OCI_ATTR_PURITY, errhp);  
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,  
               NULL, NULL, OCI_SESSGET_SPOOL);  
/* poolName is the name returned by OCISessionPoolCreate() */  
  
/* OCISessionGet mode method */  
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,  
               NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);  
/* poolName is the name returned by OCISessionPoolCreate() */
```

Note: When reusing a pooled session, the NLS attributes of the server override those of the client.

For example, if the client sets `NLS_LANG` to `french_france.us7ascii` and then is assigned a German session from the pool, the client session becomes German.

To avoid this problem, use connection classes to restrict sharing.

2.12.5.2 Connection Class

Connection class defines a logical name for the type of connection that an OCI application needs. When a pooled session has a connection class, OCI ensures that the session is not shared outside of that connection class.

For example, a connection class can prevent the following from sharing pooled sessions:

- Different users
(A session first created for user HR is assigned only to subsequent requests by user HR.)
- Different sessions of the same user
- Different applications being run by the same user
(Each application can have its own connection class.)

To set the connection class, use the `OCI_ATTR_CONNECTION_CLASS` attribute of the `OCIAuthInfo` handle. A connection class name is a string of at most 1024 bytes, and it cannot include an asterisk (*).

[Example 2–2](#) specifies that an HRMS application needs sessions with the connection class HRMS.

Example 2–2 Setting the Connection Class as HRMS

```
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "HRDB",
    strlen("HRDB"), 0, 10, 1, "HR", strlen("HR"), "HR", strlen("HR"),
    OCI_SPC_HOMOGENEOUS);

OCIAAttrSet (authInfop, OCI_HTYPE_AUTHINFO, "HRMS", strlen ("HRMS"),
    OCI_ATTR_CONNECTION_CLASS, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
    NULL, NULL, NULL, OCI_SESSGET_SPOOL);
```

[Example 2–3](#) specifies that a recruitment application needs sessions with the connection class RECMS.

Example 2–3 Setting the Connection Class as RECMS

```
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "HRDB",
    strlen("HRDB"), 0, 10, 1, "HR", strlen("HR"), "HR", strlen("HR"),
    OCI_SPC_HOMOGENEOUS);

OCIAAttrSet (authInfop, OCI_HTYPE_AUTHINFO, "RECMS", strlen("RECMS"),
    OCI_ATTR_CONNECTION_CLASS, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
    NULL, NULL, NULL, OCI_SESSGET_SPOOL);
```

2.12.5.3 Session Purity and Connection Class Defaults

[Table 2–5](#) shows the defaults for the attributes and settings of connections that an OCI application gets from the OCI session pool (using `OCISessionGet()`) and from other sources.

Table 2–5 Session Purity and Connection Class Defaults

Attribute or Setting	Default Value for Connection From OCI Session Pool	Default Value for Connection Not From OCI Session Pool
<code>OCI_ATTR_PURITY</code>	<code>OCI_ATTR_PURITY_SELF</code>	<code>OCI_ATTR_PURITY_NEW</code>
<code>OCI_ATTR_CONNECTION_CLASS</code>	OCI-generated globally unique name for each client-side session pool, used as the default connection class for all connections in the OCI session pool	SHARED
Sessions shared by ...	Threads that request sessions from the OCI session pool	Connections to a particular database that have the SHARED connection class

2.12.6 Starting Database Resident Connection Pool

The DBA must log on as SYSDBA and start the default pool, SYS_DEFAULT_CONNECTION_POOL, using DBMS_CONNECTION_POOL.START_POOL with the default settings.

For detailed information about configuring the pool, see *Oracle Database Administrator's Guide*.

2.12.7 Enabling DRCP

To enable DRCP in an application, specify either :POOLED in the Easy Connect string (as in [Example 2-4](#)) or (SERVER=POOLED) in the TNS connect string (as in [Example 2-5](#)).

Example 2-4 Enabling DRCP With :POOLED in Easy Connect String

```
oraclehost.company.com:1521/books.company.com:POOLED
```

Example 2-5 Enabling DRCP With SERVER=POOLED in TNS Connect String

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)(PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)(SERVER=POOLED)))
```

2.12.8 Benefiting from the Scalability of DRCP in an OCI Application

Consider the following OCI application scenarios and how they benefit from DRCP:

- An application neither uses the OCI session pool nor specifies a connection class or purity setting (or specifies PURITY=NEW).

The application gets a new session from DRCP. When the application returns a connection to the pool, the session is not shared with other instances of the same application by default. Therefore, the pooled server remains assigned to the client for the life of the client session. (SQL*Plus is an example of a client that does not use the OCI session pool. SQL*Plus keeps connections even when they are idle.)

The application benefits from reusing an existing pooled server.

- An application invokes OCISessionGet() outside of the OCI session pool, or to specify the connection class and PURITY=SELF.

The application can reuse both DRCP pooled servers and sessions. However, after an OCISessionRelease() call, OCI terminates the connection to the connection broker. On the next OCISessionGet() call, the application reconnects to the broker, and then DRCP assigns a pooled server (and session) belonging to the specified connection class. Reconnecting incurs the cost of connection establishment and reauthentication.

The application achieves better sharing of DRCP resources (processes and sessions) but does not benefit from caching connections to the connection broker.

- An application uses OCI session pool APIs, specifies a connection class, and specifies PURITY=SELF.

The application uses all DRCP functionality, reusing both the pooled server and the associated session and benefiting from cached connections to the connection broker. Cached connections do not incur the cost of reauthentication on the OCISessionGet() call.

See Also: *Oracle Call Interface Programmer's Guide* for information about OCISessionPoolCreate(), OCISessionGet(), OCISessionRelease(), and OCISessionPoolDestroy()

2.12.9 Benefiting from the Scalability of DRCP in a Java Application

Java applications benefit from DRCP as OCI applications do (see [Section 2.12.8](#)).

A customer who uses Universal Connection Pool (UCP), or uses ConnectionPoolDataSource as the connection factory, can upgrade to using DRCP by changing only the configuration (not the code).

2.12.10 Best Practices for Using DRCP

The steps for designing an application that can benefit from the full power of DRCP are very similar to those for an application that uses the OCI session pool (for details, see *Oracle Call Interface Programmer's Guide*).

The only additional step is that for best performance, when deployed to run with DRCP, the application must explicitly specify a connection class.

Multiple instances of the same application must specify the same connection class for best performance and enhanced sharing of DRCP resources. Ensure that the different instances of the application can share database sessions.

[Example 2–6](#) shows a DRCP application.

Example 2–6 DRCP Application

```
/* Assume that all necessary handles are allocated. */

/* This middle tier uses a single database user. Create a homogeneous
   client-side session pool */
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "BOOKSDB",
                      strlen("BOOKSDB"), 0, 10, 1, "SCOTT", strlen("SCOTT"), "password",
                      strlen("password"), OCI_SPC_HOMOGENEOUS);

while (1)
{
    /* Process a client request */
    WaitForClientRequest();
    /* Application function */

    /* Set the Connection Class on the OCIAuthInfo handle that is passed as
       argument to OCISessionGet*/
    OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO, "BOOKSTORE", strlen("BOOKSTORE"),
                OCI_ATTR_CONNECTION_CLASS, errhp);

    /* Purity need not be set, as default is OCI_ATTR_PURITY_SELF for
       OCISessionPool
       connections */

    /* You can get a SCOTT session released by Middle-tier 2 */
    OCISessionGet(envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
                  NULL, NULL, NULL, OCI_SESSGET_SPOOL);

    /* Database calls using the svchp obtained above */
    OCIStmtExecute(...)

    /* This releases the pooled server on the database for reuse */
    OCISessionRelease (svchp, errhp, NULL, 0, OCI_DEFAULT);
}

/* Middle tier is done - exiting */
```

```
OCISessionPoolDestroy (spoolhp, errhp, OCI_DEFAULT);
```

[Example 2–7](#) and [Example 2–8](#) show connect strings that deploy code in 10 middle-tier hosts that service the BOOKSTORE application from [Example 2–6](#).

In [Example 2–7](#), assume that the database is Oracle Database 12c (or earlier) in dedicated server mode with DRCP not enabled and that the client has 12c libraries. The application gets dedicated server connections from the database.

Example 2–7 Connect String for Deployment in Dedicated Server Mode Without DRCP

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp) (HOST=oraclehost.company.com)
(PORT=1521)) (CONNECT_DATA = (SERVICE_NAME=books.company.com)))
```

In [Example 2–8](#), assume that DRCP is enabled on the Oracle Database 12c database. All middle-tier processes can benefit from the pooling capability of DRCP. The database resource requirement with DRCP is much less than it would be in dedicated server mode.

Example 2–8 Connect String for Deployment With DRCP

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp) (HOST=oraclehost.company.com)
(PORT=1521)) (CONNECT_DATA = (SERVICE_NAME=books.company.com) (SERVER=POOLED)))
```

2.12.11 Compatibility and Migration

An OCI application linked with Oracle Database 12c client libraries works unaltered with:

- An Oracle Database 12c database with DRCP disabled
- A database server from a release earlier than Oracle Database 12c
- An Oracle Database 12c database server with DRCP enabled, when deployed with the DRCP connect string

Suitable clients benefit from enhanced scalability offered by DRCP if they are appropriately modified to use the OCI session pool APIs with the connection class and purity settings previously described.

As of Oracle Database 12c Release 1 (12.1.0.1), Oracle JDBC drivers support DRCP. For more information, see *Oracle Database JDBC Developer's Guide*.

2.12.12 DRCP Restrictions

The following cannot be performed with pooled servers:

- Shutting down the database
- Stopping DRCP
- Changing the password for the connected user
- Using shared database links to connect to a DRCP that is on a different instance
- Using Advanced Security Options (ASO) with TCPS
- Using Enterprise user security with DRCP
- Using migratable sessions on the server side, either directly (using the OCI_MIGRATE option) or indirectly (invoking OCISessionPoolCreate())
- Using initial client roles

- Using application context attributes (such as `OCI_ATTR_APPCTX_NAME` and `OCI_ATTR_APPCTX_VALUE`)
- Using Transaction Guard (explained in [Chapter 25](#)) with DRCP
- Using Application Continuity (explained in [Chapter 26](#)) with DRCP

Sessions created before DDL statements run can be assigned to clients after DDL statements run. Therefore, be careful when running DDL statements that affect database users in the pool. For example, before dropping a user, ensure that there are no sessions of that user in the pool and no connections to the broker that were authenticated as that user.

If sessions with explicit roles enabled are released to the pool, they can later be assigned to connections (of the same user) that need the default logon role. Therefore, avoid releasing sessions with explicit roles; instead, terminate them.

Note: You can use Oracle Advanced Security features such as encryption and strong authentication with DRCP.

Users can mix data encryption/data integrity combinations. However, users must segregate each such combination by using connection classes. For example, if the user application must specify AES256 as the encryption mechanism for one set of connections and DES for another set of connections, then the application must specify different connection classes for each set.

2.12.13 Using DRCP with Custom Pools

Oracle highly recommends using the OCI session pool, which is already integrated with DRCP, FAN, and RLB. However, an application that does not use the OCI session pool can still use DRCP if either of the following is true:

- The application was built using its own custom connection pool.
- The application uses no pool, but has periods when it does not use its session (and could therefore release it to a pool) and does not depend on getting back the same session

To use DRCP with such an application, the session must be stateful; that is, the session must have the `OCI_ATTR_SESSION_STATE` attribute (described in *Oracle Call Interface Programmer's Guide*). When an application is stateful and DRCP is enabled, OCI transparently assigns it an appropriate session from the DRCP pool. If the application is stateless (has the `OCI_SESSION_STATELESS` attribute) and DRCP is enabled, OCI transparently returns the session to the DRCP pool.

Applications must identify session state as promptly as possible for efficient utilization of underlying database resources. For more information, see [Section 2.12.14](#).

Note: An application that specifies the attribute `OCI_ATTR_SESSION_STATE` or `OCI_SESSION_STATELESS` must also specify session purity and connection class, which are explained in [Section 2.12.5](#).

2.12.14 Explicitly Marking Sessions Stateful or Stateless

An application typically requires a specific database session for the duration of a unit of work. For this duration, the session is **stateful**. After this duration, if the application

does not depend on retaining the specific session for subsequent units of work, then the session is **stateless**.

When an application or caller detects a session's transition from stateful to stateless, or the reverse, the application can explicitly inform OCI of the transition by using the `OCI_ATTR_SESSION_STATE` or `OCI_SESSION_STATELESS` attribute. This information lets OCI and Oracle Database transparently perform scalability optimizations, such as reassigning the session that the application is not using to someone else and then assigning the application a new session when necessary.

See Also: [Section 2.12.13](#)

Example 2–9 shows a code fragment that explicitly marks session states.

Example 2–9 Explicitly Marking Sessions Stateful or Stateless

```
wait_for_transaction_request();
do {

    ub1 state;

    /* mark database session as STATEFUL */
    state = OCI_SESSION_STATEFUL;
    checkerr(errhp, OCIAttrSet(usrhp, OCI_HTYPE_SESSION,
                               &state, 0, OCI_ATTR_SESSION_STATE, errhp));
    /* do database work consisting of one or more related calls to the database */

    ...

    /* done with database work, mark session as stateless */
    state = OCI_SESSION_STATELESS;
    checkerr(errhp, OCIAttrSet(usrhp, OCI_HTYPE_SESSION,
                               &state, 0, OCI_ATTR_SESSION_STATE, errhp));

    wait_for_transaction_request();

} while(not _done);
```

A session obtained from outside the OCI session pool is marked `OCI_SESSION_STATEFUL` and remains `OCI_SESSION_STATEFUL` unless the application explicitly marks it `OCI_SESSION_STATELESS`.

A session obtained from the OCI session pool is marked `OCI_SESSION_STATEFUL` by default when the first call is initiated on that session. When the session is released to the pool, it is marked `OCI_SESSION_STATELESS` by default. Therefore, you need not explicitly mark sessions as stateful or stateless when you use the OCI session pool.

See Also: [Oracle Call Interface Programmer's Guide](#) for more information about `OCI_ATTR_SESSION_STATE`

2.12.15 Using DRCP with Oracle Real Application Clusters

Oracle Real Application Clusters (Oracle RAC) is a database option in which a single database is hosted by multiple instances on multiple nodes. When DRCP is configured in a database in an Oracle RAC environment, the pool configuration is applied to each database instance. Starting or stopping the pool on one instance starts or stops the pool on all instances.

2.12.16 Using DRCP with Pluggable Databases

The DRCP in a multitenant container database (CDB) is configured and managed in the root container. You can configure, start, and stop the pool when you are connected to the root container. The pool maintains the pooled servers of different pluggable databases to which the clients are connected using different service names.

2.12.17 DRCP with Data Guard

When operating DRCP in a Data Guard environment:

- On a physical standby database:
 - You can start the pool only if the pool is running on the primary database.
 - You can stop the pool only if the pool is stopped on the primary database.
 - You cannot configure, restore to defaults, or alter pool parameters.
 The preceding restrictions cease to apply to the physical standby database if it becomes the primary database.
- On a logical standby database, all pool operations are allowed.

2.13 Using Runtime Connection Load Balancing

Topics:

- [About Runtime Connection Load Balancing](#)
- [Enabling and Disabling Runtime Connection Load Balancing](#)
- [Receiving Load Balancing Advisory FAN Events](#)

2.13.1 About Runtime Connection Load Balancing

Oracle Real Application Clusters (Oracle RAC) is a database option in which a single database is hosted by multiple instances on multiple nodes. The Oracle RAC shared disk method of clustering databases increases scalability. The nodes can easily be added or freed to meet current needs and improve availability, because if one node fails, another can assume its workload. Oracle RAC adds high availability and failover capacity to the database, because all instances have access to the whole database.

Work requests are balanced at both connect time (**connect time load balancing**, provided by Oracle Net Services) and runtime (**runtime connection load balancing**). For Oracle RAC environments, session pools use service metrics received from the Oracle RAC load balancing advisory through Fast Application Notification (FAN) events to balance application session requests. The work requests coming into the session pool can be distributed across the instances of Oracle RAC offering a service, using the current service performance.

See Also:

- `cdemosp.c` in the directory `demo`
- [Section 2.12](#)

Connect time load balancing occurs when an application creates a session. Pooled sessions must be well distributed across Oracle RAC instances when the sessions are created to ensure that sessions on each instance can execute work.

Runtime connection load balancing occurs when an application selects a session from an existing session pool (and thus is a very frequent activity). Runtime connection load balancing routes work requests to sessions in a session pool that best serve the work. For session pools that support services at only one instance, the first available session in the pool is adequate. When the pool supports services that span multiple instances, the work must be distributed across instances so that the instances that are providing better service or have greater capacity get more work requests.

OCI, OCCI, JDBC, and ODP.NET client applications all support runtime connection load balancing.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information about runtime load balancing using the OCI interface
- *Oracle C++ Call Interface Programmer's Guide* for more information about runtime load balancing using the OCCI interface
- *Oracle Database JDBC Developer's Guide* for more information about runtime load balancing using the JDBC interface
- *Oracle Universal Connection Pool for JDBC Java API Reference*
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about runtime load balancing using the ODP.NET interface

2.13.2 Enabling and Disabling Runtime Connection Load Balancing

Enabling and disabling runtime connection load balancing on the client depends on the client environment.

Topics:

- [OCI](#)
- [OCCI](#)
- [JDBC](#)
- [ODP.NET](#)

2.13.2.1 OCI

For an OCI client application, runtime connection load balancing is enabled by default in an Oracle Database 11g Release 1 (11.1) or later client communicating with a server of Oracle Database 10g Release 2 (10.2) or later when you perform the following operations to ensure that your application receives service metrics based on service time:

- Link the application with the threads library.
- Create the OCI environment in `OCI_EVENTS` and `OCI_THREADED` modes.
- Configure the load balancing advisory goal and the connection load balancing goal for a service that is used by the session pool.

To disable runtime connection load balancing for an OCI client, set the `mode` parameter to `OCI_SPC_NO_RLB` when calling `OCISessionPoolCreate()`.

FAN HA (FCF) for OCI requires `AQ_HA_NOTIFICATIONS` for the service to be TRUE.

See Also: *Oracle Call Interface Programmer's Guide* for information about `OCISessionPoolCreate()`

2.13.2.2 OCCI

For an OCCI client application, runtime connection load balancing is enabled by default in an Oracle Database 11g Release 1 (11.1) or later client communicating with a server of Oracle Database 10g Release 2 (10.2) or later when you perform the following operations:

- Link the application with the threads library.
- Create the OCCI environment in EVENTS and THREADED_MUTEXED modes.
- Configure the load balancing advisory goal and the connection load balancing goal for a service that is used by the session pool.

To disable runtime connection load balancing for an OCCI client, use the `NO_RLB` option for the `PoolType` attribute of the `StatelessConnectionPool` class.

FAN HA (FCF) for OCCI requires `AQ_HA_NOTIFICATIONS` for the service to be TRUE.

See Also: *Oracle C++ Call Interface Programmer's Guide* for more information about runtime load balancing using the OCCI interface

2.13.2.3 JDBC

In the JDBC environment, runtime connection load balancing is enabled by default in an Oracle Database 11g Release 1 (11.1) or later client communicating with a server of Oracle Database 10g Release 2 (10.2) or later when Fast Connection Failover (FCF) is enabled.

In the JDBC environment, runtime connection load balancing relies on the Oracle Notification Service (ONS) infrastructure, which uses the same out-of-band ONS event mechanism used by FCF processing. No additional setup or configuration of ONS is required to benefit from runtime connection load balancing.

To disable runtime connection load balancing in the JDBC environment, call `setFastConnectionFailoverEnabled()` with a value of false.

See Also: *Oracle Database JDBC Developer's Guide* for more information about runtime load balancing using the JDBC interface

2.13.2.4 ODP.NET

In an ODP.NET client application, runtime connection load balancing is disabled by default. To enable runtime connection load balancing, include "Load_Balancing=true" in the connection string.

FAN HA (FCF) for ODP.NET requires `AQ_HA_NOTIFICATIONS` for the service to be TRUE.

See Also: *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about runtime load balancing

2.13.3 Receiving Load Balancing Advisory FAN Events

Your application can receive load balancing advisory FAN events only if all of these requirements are met:

- Oracle RAC environment with Oracle Clusterware is set up and enabled.
- The server is configured to issue event notifications.

- The application is linked with the threads library.
- The OCI environment is created in OCI_EVENTS and OCI_THREADED modes.
- The OCCI environment is created in THREADED_MUTEXED and EVENTS modes.
- You configured or modified the Oracle RAC environment using the DBMS_SERVICE package.

You must modify the service to set up its goal and the connection load balancing goal as follows:

```
EXEC DBMS_SERVICE.MODIFY_SERVICE("myService",
    DBMS_SERVICE.GOAL_SERVICE_TIME,
    clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
```

The constant GOAL_SERVICE_TIME specifies that Load Balancing Advisory is based on elapsed time for work done in the service plus bandwidth available to the service.

The constant CLB_GOAL_SHORT specifies that connection load balancing uses Load Balancing Advisory, when Load Balancing Advisory is enabled.

You can set the connection balancing goal to CLB_GOAL_LONG. However, CLB_GOAL_LONG is typically useful for closed workloads (that is, when the rate of completing work is equal to the rate of starting new work).

See Also:

- *Oracle Real Application Clusters Administration and Deployment Guide* for information about enabling OCI clients to receive FAN events
- *Oracle Database PL/SQL Packages and Types Reference* for information about DBMS_SERVICE
- *Oracle Call Interface Programmer's Guide* for information about OCISessionPoolCreate()
- *Oracle Database JDBC Developer's Guide* for more information about runtime load balancing using the JDBC interface

3

Security

This chapter explains some fundamentals of designing security into the database and database applications.

Topics:

- [Enabling User Access with Grants, Roles, and Least Privilege](#)
- [Automating Database Logins](#)
- [Controlling User Access with Fine-Grained Access Control](#)
- [Using Invoker's and Definer's Rights for Procedures and Functions](#)
- [Managing External Procedures for Your Applications](#)
- [Auditing User Activity](#)

3.1 Enabling User Access with Grants, Roles, and Least Privilege

This topic explains how you can grant privileges and roles to users to restrict access to data. It also explains the importance of the concept of least privilege, introduces secure application roles as a way to automatically filter out users who attempt to log in to your applications.

A user **privilege** is the right to perform an action, such as updating or deleting data. You can grant users privileges to perform these actions. A **role** is named collection of privileges that are grouped together, usually to enable users to perform a set of tasks related to their jobs. For example, a role called `clerk` can enable clerks to do things like create, update, and delete files. The `clerk_mgr` role can include the `clerk` role, plus some additional privileges such as approving the clerks' expense reports or managing their performance appraisals.

When you grant privileges to users, apply the principle of least privilege: *Grant users only the privileges that they need*. If possible, do not directly grant the user a privilege. Instead, create a role that defines the set of privileges the user needs and then grant the user this role. For example, grant user `fred` the `CREATE SESSION` privilege so that he can log in to a database session. But for the privileges that he needs for his job, such as the `UPDATE TABLE` privilege, grant him a role that has those privileges.

You can design your applications to automatically grant a role to the user who is trying to log in, provided the user meets criteria that you specify. To do so, you create a **secure application role**, which is a role that is associated with a PL/SQL procedure (or PL/SQL package that contains multiple procedures). The procedure validates the user: if the user fails the validation, then the user cannot log in. If the user passes the validation, then the procedure grants the user a role so that he or she can use the

application. The user has this role only while he or she is logged in to the application. When the user logs out, the role is revoked.

[Example 3-1](#) shows a secure application role procedure that allows the user to log in during business hours (8 a.m. to 5 p.m.) from a specific set of work stations. If the user passes these checks, then the user is granted the `hr_admin` role and then is able to log in to the application.

[Example 3-1 Secure Application Role Procedure to Restrict Access to Business Hours](#)

```
CREATE OR REPLACE PROCEDURE hr_admin_role_check
  AUTHID CURRENT_USER
  AS
  BEGIN
    IF (SYS_CONTEXT ('userenv', 'ip_address')
        BETWEEN '192.0.2.10' and '192.0.2.20'
        AND
        TO_CHAR (SYSDATE, 'HH24') BETWEEN 8 AND 17)
    THEN
      EXECUTE IMMEDIATE 'SET ROLE hr_admin';
    END IF;
  END;
/
```

See Also:

- [Oracle Database Security Guide](#) more information about privilege and role authorization
- [Oracle Database Security Guide](#) more information about secure application roles

3.2 Automating Database Logins

You can automate database logins. To do so, you create a logon trigger to run a PL/SQL procedure that can validate a user who is attempting to log in to an application. When the user logs in, the trigger executes. Logon triggers can perform multiple actions, such as generating an alert if the user fails the validation, displaying error messages, and so on.

[Example 3-2](#) shows a simple logon trigger that executes a PL/SQL procedure.

[Example 3-2 Creating a Logon Trigger](#)

```
CREATE OR REPLACE TRIGGER run_logon_trig AFTER LOGON ON DATABASE
BEGIN
  sec_mgr.check_user_proc;
END;
```

See Also:

- [Oracle Database PL/SQL Language Reference](#) for detailed information about the `CREATE TRIGGER` statement
- [Oracle Database Security Guide](#) for information about how to create a logon trigger that runs a database session application context package

3.3 Controlling User Access with Fine-Grained Access Control

There are several ways that you can control the level of access users have to data from your applications.

- **Oracle Virtual Private Database (VPD):** VPD enables you to create policies that can restrict database access at the row and column level. Essentially, VPD adds a dynamic WHERE clause to a SQL statement that is issued against the table, view, or synonym to which an VPD security policy was applied.

For example, suppose the user enters the `SELECT * FROM OE.ORDERS` statement. A VPD policy can transform this statement to the following statement instead, so that only the user who has logged in can view this data:

```
SELECT * FROM OE.ORDERS
WHERE SALES_REP_ID = SYS_CONTEXT('USERENV', 'SESSION_USER');
```

- **Oracle Data Redaction:** Oracle Data Redaction masks data at run time, at the moment the user attempts to access the data (that is, at query-execution time). This solution works well in a dynamic production system in which data is constantly changing. During the time that the data is being redacted, all data processing is performed normally, and the back-end referential integrity constraints are preserved. You typically redact sensitive data, such as credit card or Social Security numbers.

You can mask the data in the following ways:

- **Full redaction, in which the entire data is masked.** For example, the number 37828224 can be displayed as a zero.
 - **Partial redaction, in which only a portion of the data is redacted.** With this type, the number 37828224 can be displayed as ****224.
 - **Random redaction, in which the data is displayed as randomized data.** Here, the number 37828224 can appear as 93204857.
 - **Regular expressions, in which you redact data based on a search pattern.** You can use regular expressions in both full and partial redaction. For example, you can redact the user name of email addresses, so that only the domain shows: jsmith in the email address jsmith@example.com can be replaced with [redacted] so that the email address appears as [redacted]@example.com.
 - **No redaction, which enables you to test the internal operation of your redaction policies, with no effect on the results of queries against tables with policies defined on them.** You can use this option to test the redaction policy definitions before applying them to a production environment.
 - **Oracle Label Security:** Oracle Label Security secures your database tables at the row level, and assigns these rows different levels of security based on the needs of your site. Rows that contain highly sensitive data can be assigned a label entitled HIGHLY SENSITIVE; rows that are less sensitive can be labeled as SENSITIVE, and so on. Rows that all users can have access to can be labeled PUBLIC. You can create as many labels as you need, to fit your site's security requirements.
- For example, when user fred, who is a low-level employee, logs in, he would see only data that is available to all users, as defined by the PUBLIC label. Yet when his director, hortensia, logs in, she can see all the sensitive data that has been assigned the HIGHLY SENSITIVE label.
- **Oracle Database Vault:** Oracle Database Vault enables you to restrict administrative access to your data. By default, administrators (such as user SYS

with the SYSDBA privilege) have access to all data in the database. Administrators typically must perform tasks such performance tuning, backup and recovery, and so on. However, they do not need access to your salary records. Database Vault enables you to create policies that restrict the administrator's actions yet not prevent him or her from performing normal administrative activities.

A typical Database Vault policy could, for example, prevent an administrator from accessing and modifying the HR.EMPLOYEES table. You can create fine-tuned policies that impose restrictions such as limiting the hours the administrator can log in, which computers he can use, whether he can log in to the database using dynamic tools, and so on. Furthermore, you can create policies that generate alerts if the administrator tries to violate a policy.

See Also:

- *Oracle Database Security Guide* for more information about Oracle Virtual Private Database
- *Oracle Database Advanced Security Guide* for more information about Oracle Data Redaction
- *Oracle Label Security Administrator's Guide* for more information about Oracle Label Security
- *Oracle Database Vault Administrator's Guide* for more information about Oracle Database Vault
- *Oracle Database 2 Day + Security Guide* for tutorials on all of these fine-grained access control features

3.4 Using Invoker's and Definer's Rights for Procedures and Functions

Topics:

- [What Are Invoker's Rights and Definer's Rights?](#)
- [Protecting Users Who Run Invoker's Rights Procedures and Functions](#)
- [How Default Rights Are Handled for Java Stored Procedures](#)

3.4.1 What Are Invoker's Rights and Definer's Rights?

When you create a procedure or function (that is, a program unit), you can design it so that it runs with either the privileges of the owner (you) or the privileges of the person who is invoking it. Definer's rights run the program unit using the owner's privileges and invoker's rights run the program unit using the privileges of the person who runs it. For example, suppose user `harold` creates a procedure that updates the table `orders`. User `harold` then grants the `EXECUTE` privilege on this procedure to user `hazel`. If `harold` had created the procedure with definer's rights, then the procedure would expect the `orders` table to be in `harold`'s schema. If he created it with invoker's rights, then the procedure would look for the `orders` table in `hazel`'s schema.

To designate a program unit as definer's rights or invokers rights, use the `AUTHID` clause in the creation statement. If you omit the `AUTHID` clause, then the program unit is created with definer's rights.

[Example 3–3](#) shows how to use the `AUTHID` clause in a `CREATE PROCEDURE` statement to specify definer's rights or invoker's rights.

Example 3-3 Creating Program Units with Definer's Rights or Invoker's Rights

```
CREATE PROCEDURE my_def_proc AUTHID DEFINER -- Definer's rights procedure
AS ...

CREATE PROCEDURE my_inv_proc AUTHID CURRENT_USER -- Invoker's rights procedure
AS ...
```

See Also:

- *Oracle Database PL/SQL Language Reference* for details about definer's rights and invoker's rights procedures and functions
- *Oracle Database PL/SQL Language Reference* for details about the CREATE PROCEDURE statement
- *Oracle Database PL/SQL Language Reference* for details about the CREATE FUNCTION statement

3.4.2 Protecting Users Who Run Invoker's Rights Procedures and Functions

An important consideration when you create an invoker's rights program unit is the level of privilege that the invoking users have. Suppose user harold is a low-ranking employee who has few privileges and hazel is an executive with many privileges. When hazel runs harold's invoker's rights procedure, the procedure temporarily inherits hazel's privileges (all of them). But because harold owns this procedure, he can modify it without her knowing it to behave in ways that take advantage of hazel's privileges, such as giving harold a raise. To help safeguard against this type of scenario, after she has ensured that harold is trustworthy, user hazel can grant harold permission so that his invoker's rights procedures and functions have access to hazel's privileges when she runs them. To do so, she must grant him the INHERIT PRIVILEGES privilege.

[Example 3-4](#) shows how invoking user hazel can grant user harold the INHERIT PRIVILEGES privilege

Example 3-4 Granting a Program Unit Creating the INHERIT PRIVILEGES Privilege

```
GRANT INHERIT PRIVILEGES ON hazel TO harold;
```

If harold proves untrustworthy, hazel can revoke the INHERIT PRIVILEGES privilege from him.

Administrators such as user SYS and SYSTEM have the INHERIT ANY PRIVILEGES privilege, which enable these users' invoker's rights procedures to have access to the privileges of any invoking users. As with all ANY privileges, grant this privilege only to trusted users.

See Also: *Oracle Database Security Guide* for more information about managing security for definer's rights and invoker's rights procedures and functions

3.4.3 How Default Rights Are Handled for Java Stored Procedures

By default, Java class schema objects run with the privileges of their invoker, not with definer's rights. If you want your Java schema objects to run with definer's rights, then when you load them by using the loadjava tool, specify the -definer option.

[Example 3-5](#) shows how to use the -definer option in a loadjava command.

Example 3–5 Loading a Java Class with Definer's Rights

```
loadjava -u joe -resolve -schema TEST -definer ServerObjects.jar  
Password: password
```

You can use the `-definer` option on individual classes. Be aware that different definers may have different privileges. Apply the `-definer` option with care, so that you can achieve the desired results. Doing so helps to ensure that your classes run with no more than the privileges that they need.

See Also:

- *Oracle Database Java Developer's Guide* for detailed information about the `loadjava` tool
- *Oracle Database Java Developer's Guide* for more information about controlling the current user in Java applications

3.5 Managing External Procedures for Your Applications

For security reasons, Oracle external procedures run in a process that is physically separate from Oracle Database. When you invoke an external procedure, Oracle Database creates the `extproc` operating system process (or agent), by using the operating system privileges of the user that started the listener for the database instance.

You can configure the `extproc` agent to run as a designated operating system credential. To use this functionality, you define a credential to be associated with the `extproc` process, which then can authenticate impersonate (that is, run on behalf of the supplied user credential) before loading a user-defined shared library and executing a function. To configure the `extproc` user credential, you use the PL/SQL package `DBMS_CREDENTIAL` and the PL/SQL statement `CREATE LIBRARY`.

See Also: *Oracle Database Security Guide* for more information about securing external procedures

3.6 Auditing User Activity

You can create audit policies to audit specific actions in the database. Oracle Database then records these actions in an audit trail. The database mandatorily audits some actions and writes these to the audit trail as well. The audit policies that you create can be simple, such as auditing all actions by a specific user, or complex, such as testing for specific conditions and sending email alerts if these conditions are violated.

When you install an Oracle Database, you can choose how your database is audited.

- **Unified auditing:** In unified auditing, all audit trails are written to a single audit trail, viewable by the `V$UNIFIED_AUDIT_TRAIL` and `GV$UNIFIED_AUDIT_TRAIL` dynamic views. This audit trail encompasses not only Oracle Database-specific actions, but also actions performed in Oracle Real Application Security, Oracle Recovery Manager, Oracle Database Vault, and Oracle Label Security environments. The audit records from these sources are presented in a consistent, uniform format. You can create named audit policies and enable and disable them as necessary. If you want to use unified auditing, then you must migrate your databases to it.
- **Mixture of unified auditing and pre-Release 12c auditing:** For the most part, this option enables you to use the pre-Release 12c auditing, in which audit records are written to different locations using their own formats. However, Release 12c

functionality, such as using auditing in a multitenant environment, is available. This type of auditing is the default for both new and upgraded databases.

In both cases, when you upgrade your databases to Oracle Database 12c Release 1 (12.1.0.1), the audit records from the previous release are preserved. If you decide to migrate to use unified auditing fully, you can archive these earlier records and then purge the audit trail. After you complete the migration, the new audit records are written to the unified audit trail.

See Also:

- *Oracle Database Security Guide* for more information about creating and managing unified auditing policies
- *Oracle Database Security Guide* to find a detailed comparison of unified auditing and pre-Release 12c auditing
- *Oracle Database Upgrade Guide* for information about migrating to unified auditing

High Availability

This chapter explains how to design high availability into the database and database applications.

Topics:

- [Failover and Query Replay](#)
- [Fast Application Notification \(FAN\) and Fast Connection Failover \(FCF\)](#)
- [Application Continuity and Transaction Guard](#)
- [Service and Load Management for Database Clouds](#)

4.1 Failover and Query Replay

This topic describes the following topics:

- [Transparent Application Failover \(TAF\)](#)
- [Application Continuity](#)

4.1.1 Transparent Application Failover (TAF)

This section describes what Transparent Application Failover (TAF) is, how to configure TAF, and using TAF callbacks to notify the application of events as they are generated.

Topics:

- [About Transparent Application Failover](#)
- [Configuring Transparent Application Failover](#)
- [Using Transparent Application Failover Callbacks](#)

4.1.1.1 About Transparent Application Failover

Transparent Application Failover (TAF) is a client-side feature of OCI, OCCI, Java Database Connectivity (JDBC) OCI driver, and ODP.NET designed to minimize disruptions to end-user applications that occur when database connectivity fails because of instance or network failure. TAF can be implemented on a variety of system configurations including Oracle Real Application Clusters (Oracle RAC), Oracle Data Guard physical standby databases, and on a single instance system after it restarts (for example, when repairs are made).

TAF enables client applications to automatically (transparently) reconnect to a preconfigured secondary instance, creating a fresh connection, but identical to the

connection that was established on the first original instance. That is, the connection properties are the same as that of the earlier connection, regardless of how the connection was lost. In this case, the active transactions roll back. Also, all statements that an application attempts to use after a failure attempt also failover.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information about OCI TAF
- *Oracle C++ Call Interface Programmer's Guide* for more information about OCCI TAF
- *Oracle Database JDBC Developer's Guide* for more information about JDBC TAF
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about ODP.NET TAF
- *Oracle Database Net Services Reference* for more information about client-side configuration of TAF (Connect Data Section)
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the server-side configuration of TAF (DBMS_SERVICE)

4.1.1.2 Configuring Transparent Application Failover

TAF can be configured on both the client side and server side with the server side taking precedence if both client and server sides are configured. On the client side, you configure TAF by including the FAILOVER_MODE parameter in the CONNECT_DATA portion of a connect descriptor. On the server side, you configure TAF by modifying the target service with the DBMS_SERVICE.MODIFY_SERVICE packaged procedure.

See the references in [Section 4.1.1.1](#) for more configuration information.

4.1.1.3 Using Transparent Application Failover Callbacks

TAF callbacks are callbacks that are registered in case of failover and called during failover to notify the application of events as they are generated. They are called several times while reestablishing the user's session.

As the application developer you may want to inform the user that failover is in progress because there is a slight delay as failover proceeds. The first call to the callback carries out that function. Also, when failover is successful and the connection is reestablished, you may want to inform the user that failover has happened and then you may want to replay ALTER SESSION commands because these commands are not automatically replayed on the second instance. A subsequent call to the callback performs that function. Also, if failover is unsuccessful, then you want to inform the application that failover cannot occur. A third call to the callback performs this function as well.

Using TAF callbacks makes possible:

- Notifying users of the status of failover throughout the failover process; when failover is underway, when failover is successful, and when failover is unsuccessful
- Replying of ALTER SESSION commands when that is needed

- Reauthenticating a user handle besides the primary handle for each time a session begins on the new connection. Because each user handle represents a server-side session, the client may want to replay `ALTER SESSION` commands for that session.

See the references in [Section 4.1.1.2](#) for specific callback registration information for each interface; such as, the structure and parameters of the callback for each interface, and for failover callback examples.

4.1.2 Application Continuity

Beginning with Oracle Database 12c Release 1 (12.1.0.1), Oracle Database supports Application Continuity for Java. This feature supports the concept of commit outcome and recovery of active requests allowing JDBC-based applications in high availability (HA) infrastructures to avoid application errors from system failures by providing an immediate implicit recovery and resubmission of all active requests as soon as failover occurs and connectivity is restored to database services. See [Section 4.3, "Application Continuity and Transaction Guard,"](#) for more information.

See Also: [Chapter 26, "Ensuring Application Continuity,"](#) for more information about Application Continuity

4.2 Fast Application Notification (FAN) and Fast Connection Failover (FCF)

This section describes what Fast Application Notification (FAN) and Fast Connection Failover (FCF) are and how applications can respond to FAN events in a high availability environment and use FCF to relocate connections after a failover.

Topics:

- [About Fast Application Notification \(FAN\)](#)
- [About Fast Connection Failover \(FCF\)](#)

4.2.1 About Fast Application Notification (FAN)

An important component of high availability is a notification mechanism called Fast Application Notification (FAN). FAN notifies other processes about configuration and service level information that includes service status changes, such as UP or DOWN events. Applications can respond to FAN events and take immediate action. FAN UP and DOWN events can apply to instances, services, and nodes.

FAN provides the ability to immediately terminate an active transaction when an instance or server fails. FAN integrated Oracle clients receive the events and respond. Applications can respond either by propagating the error to the user or by resubmitting the transactions and masking the error from the application user. When a DOWN event occurs, FAN integrated clients immediately clean up connections to the terminated database. When an UP event occurs, the FAN integrated clients create new connections to the new primary database instance.

Oracle has integrated FAN with many of the common Oracle client drivers. Therefore, the easiest way to use FAN is to use an integrated Oracle client. You can use the Oracle Connection Manager (CMAN) session pools, OCI, Universal Connection Pool for Java, JDBC `simplefan` API, and ODP.NET connection pools. The overall goal is to enable applications to consistently obtain connections to the available primary database at anytime.

FAN events are published using Oracle Notification Service and Oracle Streams Advanced Queuing. The publication mechanisms are automatically configured as part of an Oracle RAC installation. Here, an Oracle RAC installation means any installation of Oracle Clusterware with Oracle RAC, Oracle RAC One Node, Oracle Data Guard (fast-start-failover), or Oracle Data Guard single instance with Oracle Clusterware. Beginning with Oracle Database 12c Release 1 (12.1), ONS is the primary notification mechanism for a new client (Oracle Database 12c Release 1 (12.1)) and a new server (Oracle Database 12c Release 1 (12.1)), while the AQ HA notification feature is deprecated and maintained only for backward compatibility when there is an old client (pre-Oracle Database 12c Release 1 (12.1)) or old server (pre-Oracle Database 12c Release 1 (12.1)).

Applications can use FAN programmatically by using the JDBC and Oracle RAC FAN application programming interface (API) or by using callbacks with OCI to subscribe to FAN events and to execute event handling actions upon the receipt of an event.

When you use JDBC or Oracle Database 12c Release 1 (12.1.0.1) OCI or ODP.NET clients, you must create an Oracle Notification Services (ONS) network that is running on the server. When you use Oracle Database 10g or Oracle Database 11g OCI or ODP.NET clients, you must enable Oracle Advanced Queuing (AQ) HA notifications for your services.

See Also:

- *Oracle Real Application Clusters Administration and Deployment Guide* for more information about FAN
- *Oracle Database Administrator's Guide* for information about enabling FAN events in an Oracle Restart environment
- *Oracle Call Interface Programmer's Guide* for more information about receiving runtime connection load balancing advisory FAN events to balance application session requests in an Oracle RAC environment with Oracle Clusterware set up and enabled
- *Oracle C++ Call Interface Programmer's Guide* for more information about runtime load balancing of the stateless connection pool by use of service metrics distributed by the Oracle RAC load-balancing advisory
- *Oracle Database JDBC Developer's Guide* for more information about fast connection failover

4.2.2 About Fast Connection Failover (FCF)

In a configuration with a standby database, after you have added Oracle Notification Services (ONS) to your Oracle Restart configurations and enabled Oracle Advanced Queuing (AQ) HA notifications for your services, you can enable clients for Fast Connection Failover (FCF). The clients then receive FAN events and can relocate connections to the current primary database after an Oracle Data Guard failover. Beginning with Oracle Database 12c Release 1 (12.1), ONS is the primary notification mechanism for a new client (Oracle Database 12c Release 1 (12.1)) and a new server (Oracle Database 12c Release 1 (12.1)), while the AQ HA notification feature is deprecated and maintained only for backward compatibility when there is an old client (pre-Oracle Database 12c Release 1 (12.1)) or old server (pre-Oracle Database 12c Release 1 (12.1)).

For databases with no standby database configured, you can still configure the client FAN events. When there is an outage (planned or unplanned), you can configure the

client to retry the connection to the database. Because Oracle Restart restarts the failed database, the client can reconnect when the database restarts.

You must enable FAN events to provide FAN integrated clients support for FCF in an Oracle Data Guard or standalone environment with no standby database.

FCF offers a driver-independent way for your Java Database Connectivity (JDBC) application to take advantage of the connection failover facilities offered by Oracle Database. FCF is integrated with implicit connection cache and Oracle RAC to provide high availability event notification.

OCI clients can enable FCF by registering to receive notifications about Oracle Restart high availability FAN events and respond when events occur. This improves the session failover response time in OCI and removes terminated connections from connection and session pools. This feature works on OCI applications, including those that use Transparent Application Failover (TAF), connection pools, or session pools.

See Also:

- *Oracle C++ Call Interface Programmer's Guide* for more information about runtime load balancing of the stateless connection pool by use of service metrics distributed by the Oracle RAC load-balancing advisory
- *Oracle Database JDBC Developer's Guide* for more information about fast connection failover
- *Oracle Universal Connection Pool for JDBC Java API Reference*
- *Oracle Database Administrator's Guide* for information about enabling FCF for JDBC clients
- *Oracle Database Administrator's Guide* for information about enabling FCF for OCI clients
- *Oracle Database Administrator's Guide* for information about enabling FCF for ODP.NET clients

4.3 Application Continuity and Transaction Guard

In Oracle Database 10g Release 2 (10.2) HA framework, JDBC clients, OCI clients, and ODP.NET clients support fast application notification (FAN) messages. FAN is designed to quickly notify an application of outages at the node, database, instance, service, and public network levels. After being notified of the failure, an application can reestablish the failed connection on a surviving instance. However, for releases before Oracle Database 12c Release 1 (12.1.0.1) there was no ability to reliably determine the outcome of the failed transaction and the ability to recover the active transaction after database connectivity had been restored.

Beginning with Oracle Database 12c Release 1 (12.1.0.1), the DBA can configure server-side settings for the database services used by the applications to support two new features Application Continuity for Java and Transaction Guard. See [Section 26.2.2.1, "When Is Application Continuity Transparent?"](#) for information about when Application Continuity for Java is transparent (performed automatically) and when it is not transparent and what infrastructure changes may be needed when it is not transparent.

See Also:

- [Chapter 26, "Ensuring Application Continuity"](#) and [Chapter 25, "Using Transaction Guard"](#) for more information about Application Continuity for Java and Transaction Guard
- *Oracle Call Interface Programmer's Guide* for more information about OCI and Transaction Guard
- *Oracle Database JDBC Developer's Guide* for more information about JDBC, Transaction Guard, and Application Continuity for Java

4.3.1 Transaction Guard

Transaction Guard is a protocol and developer API supported for JDBC Type 4 (Oracle Thin), OCI, OCCI, and ODP.Net drivers.

Transaction Guard introduces the concept of *at-most-once transaction semantics*, also referred to as transaction idempotence. When an application opens a connection to the database using this service, the logical transaction ID (LTXID) is generated at authentication and stored in the session handle at the database and a copy at the client driver. This is a globally unique ID that identifies the database transaction from the application perspective. Applications use the Transaction Guard APIs to obtain a known outcome following a recoverable error.

When there is an outage, an application using Transaction Guard can retrieve the LTXID from the previous failed session's handle and use it to determine the outcome of the transaction that was active prior to the session failure. If the LTXID is determined to be `UNCOMMITTED`, then the application can return the `UNCOMMITTED` outcome to the user to decide what action to take, or optionally, the application can replay an uncommitted transaction. If the LTXID is determined to be `COMMITTED`, then the transaction is committed and the application can return this outcome to the end user and might be able to take a new connection and continue. Transaction Guard also reports whether the last user call not only `COMMITTED`, but also whether it completed changing needed nontransactional states - see `USER_CALL_COMPLETED`. See [Chapter 25](#) for more information about Transaction Guard.

4.3.2 Application Continuity for Java

Application Continuity for Java masks planned or unplanned outages (that cause database session unavailability) by attempting to rebuild the database session transactional and nontransactional states, so the outage appears to the user as no more than a delayed execution.

Application Continuity for Java works with the Oracle Database 12c Release 1 (12.1.0.1) server to determine if the database session can be replayed.

When a recoverable error occurs that makes the database session unavailable, an error message is sent back to the application. A driver receives the FAN message (down or interrupt), aborts the dead session, initiates reconnect and reauthenticates, checks out a new session and uses the LTXID of the dead session to determine the last outcome of the session at the outage. If an optional callback has been registered, the JDBC replay driver initializes the connection using this callback to restore initial nontransactional session state (NTSS), then continues replaying the request.

Application Continuity prepares replay by using Transaction Guard to determine the outcome of the last operation submitted by the session that received the error. If the submission committed and completed, the new session returns this result to the

application and continues with the nontransactional state established if the SESSION_STATE_CONSISTENCY mode is STATIC, or exits if the SESSION_STATE_CONSISTENCY mode is DYNAMIC. DYNAMIC session state consistency is appropriate for most applications.

If the last submission has replay enabled, the replay driver prepares to replay the submission and replays the saved statements for the request. When replaying, preserved mutable data are restored if permission has been granted. Validation is performed at the server to ensure that the client-visible results are identical to the original submission. When replay is complete, the application proceeds with its application logic returning to runtime mode as if all that occurred was a delay in execution similar to that which happens under load.

In some cases, replay cannot restore the data that the client potentially made decisions upon. The replay then returns the original error to the application and appears like a delayed error.

Application Continuity supports recovering any outage that is due to database unavailability against a copy of a database with the same DBID (forward in time) or within an Active Data Guard farm. This may be Oracle RAC One, Oracle Real Application Clusters, Data Guard, or Active Data Guard.

See [Section 26.2.2.1, "When Is Application Continuity Transparent?"](#) for information about when Application Continuity for Java is transparent (performed automatically) and when it is not transparent and what infrastructure changes may be needed when it is not transparent. See [Section 26.3, "Potential Side Effects of Application Continuity,"](#) for some replay side effects that you may need to mitigate. See [Section 26.4, "Restrictions and Other Considerations for Application Continuity,"](#) for replay restrictions.

4.4 Service and Load Management for Database Clouds

This section describes the service and load management framework. It also explains how OCI based applications in a service and load management framework environment utilize fast application notification (FAN) messages to respond to outages at the node, database, instance, service, and public network levels. After being notified of the outage, an application can reestablish the failed connection on a surviving instance in this framework.

Topics:

- [About Service and Load Management for Database Clouds](#)

4.4.1 About Service and Load Management for Database Clouds

The database cloud is a self-contained system of databases integrated by the service and load management framework that ensures high performance, availability and optimal utilization of resources. This framework provides effective balancing of processing workload across distributed databases that maintain multiple synchronized replicas both locally and in geographically disparate regional data centers. Replicas may be instances in an Oracle RAC environment, or single instances interconnected using Oracle Data Guard, Oracle Golden Gate, or any combination that supports replication technology. Thus, the service and load management framework provides dynamic load balancing, failover, and centralized service management for these replicated databases.

A global service is a database service provided by multiple databases synchronized through some form of data replication that satisfies quality of service requirements for

the service. This allows a client request for a service to be forwarded to any database that provides that service.

A database pool within a database cloud consists of all databases that provide the same global service that belong to the same administrative domain. The database cloud is partitioned into multiple database pools to simplify service management and to provide higher levels of security by allowing each pool to be administered by a different administrator.

A global service manager (GSM) is a software component that provides service-level load balancing and centralized management of services within the database cloud. Load balancing is done at connection and runtime. Other capabilities provided by GSM include creation, configuration, starting, stopping, and relocation of services and maintaining global service properties such as cardinality and region locality. A region is a logical boundary known as a data center that contains database clients and servers that are considered close enough to each other so as to reduce network latency to levels required by applications accessing these data centers.

The GSM can run on a separate host, or can be colocated with a database instance. Every region must have at least one GSM instance running. For high availability, Oracle recommends that you deploy multiple GSM instances in a region. A GSM instance belongs to only one particular region; however, it manages global services in all database pools associated with this region.

From an application developer's perspective, a client program connects to a regional global service manager (GSM) and requests a connection to a global service. The client need not specify which database or instance it requires. GSM forwards the client's request to the optimal instance within a database pool in the database cloud that offers the service.

Beginning with Oracle Database 12c Release 1 (12.1.0.1), the DBA can configure client-side connect strings for database services in a Global Data Services (GDS) framework using an Oracle Net string.

Introduced in Oracle Database 12c Release 1 (12.1.0.1), the logical transaction ID (LTXID) is initially generated at authentication and stored in the session handle and used to identify a database transaction from the application perspective. The logical transaction ID is globally unique and identifies the transaction within a highly available (HA) infrastructure.

Using the HA Framework, a client application (JDBC, OCI, and ODP.NET) supports fast application notification (FAN) messages. FAN is designed to quickly notify an application of outages at the node, database, instance, service, and public network levels. After being notified of the outage, an application can reestablish the failed connection on a surviving instance.

Beginning with Oracle Database 12c Release 1 (12.1.0.1), the DBA can configure server-side settings for the database services used by the applications to support Application Continuity for Java and Transaction Guard.

See Also:

- *Oracle Database Concepts* for an overview of global service management and description of the physical and logical components of the service and load management framework
- *Oracle Database Global Data Services Concepts and Administration Guide* for more information about global service management in a database cloud
- *Oracle Database Global Data Services Concepts and Administration Guide* for more information about configuring database clients for connectivity to the Global Data Services (GDS) framework.
- [Chapter 25, "Using Transaction Guard"](#) and [Chapter 26, "Ensuring Application Continuity,"](#) for more information about Transaction Guard and Application Continuity
- *Oracle Call Interface Programmer's Guide* for more information about OCI, Application Continuity, and Transaction Guard
- *Oracle Database JDBC Developer's Guide* for more information about JDBC, Application Continuity, and Transaction Guard

Advanced PL/SQL Features

This chapter introduces the advanced PL/SQL features and refers to other chapters or documents for more information.

Topics:

- [PL/SQL Data Types](#)
- [Dynamic SQL](#)
- [PL/SQL Optimize Level](#)
- [Compiling PL/SQL Units for Native Execution](#)
- [Exception Handling](#)
- [Conditional Compilation](#)
- [Bulk Binding](#)

See Also:

- [Part III, "PL/SQL for Application Developers"](#)
- [Oracle Database PL/SQL Language Reference](#) for a complete description of PL/SQL

5.1 PL/SQL Data Types

The PL/SQL data types include the SQL data types, additional scalar data types, and composite data types. You define the composite data types. You can also define subtypes of the scalar data types. For more information, see [Section 11.4.4, "PL/SQL Data Types."](#)

5.2 Dynamic SQL

Dynamic SQL is a programming methodology for generating and running SQL statements at runtime. It is useful when writing general-purpose and flexible programs like dynamic query systems, when writing programs that must run database definition language (DDL) statements, or when you do not know at compile time the full text of a SQL statement or the number or data types of its input and output variables. For more information, see [Section 11.4.9, "PL/SQL Dynamic SQL."](#)

5.3 PL/SQL Optimize Level

The PL/SQL optimize level determines how much the PL/SQL optimizer can rearrange code for better performance. This level is set with the compilation parameter

`PLSQL_OPTIMIZE_LEVEL`, which is described in *Oracle Database Reference*. For more information, see [Section 11.3.1, "PLSQL_OPTIMIZE_LEVEL Compilation Parameter."](#)

5.4 Compiling PL/SQL Units for Native Execution

You can usually speed up PL/SQL units by compiling them into native code (processor-dependent system code), which is stored in the SYSTEM tablespace. For more information about compiling PL/SQL units for native execution, see [Section 11.7, "Compiling PL/SQL Units for Native Execution."](#)

5.5 Exception Handling

Exceptions (PL/SQL runtime errors) can arise from design faults, coding mistakes, hardware failures, and many other sources. You cannot anticipate all possible exceptions, but you can write exception handlers that let your program to continue to operate in their presence. For more information, see *Oracle Database PL/SQL Language Reference*.

5.6 Conditional Compilation

Conditional compilation lets you customize the functionality of a PL/SQL application without removing source text. For example, you can:

- Use new features with the latest database release and disable them when running the application in an older database release.
- Activate debugging or tracing statements in the development environment and hide them when running the application at a production site.

However:

- Oracle recommends against using conditional compilation to change the attribute structure of a type, which can cause dependent objects to "go out of sync" or dependent tables to become inaccessible.

To change the attribute structure of a type, Oracle recommends using the `ALTER TYPE` statement (described in *Oracle Database SQL Language Reference*), which propagates changes to dependent objects.

- Conditional compilation is subject to restrictions.

For more information about conditional compilation, see *Oracle Database PL/SQL Language Reference*.

5.7 Bulk Binding

Bulk binding minimizes the performance overhead of the communication between PL/SQL and SQL, which can greatly improve performance. For more information, see [Section 11.4.8, "Overview of Bulk Binding."](#)

Part II

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:

- [Chapter 6, "SQL Processing for Application Developers"](#)
- [Chapter 7, "Using SQL Data Types in Database Applications"](#)
- [Chapter 8, "Using Regular Expressions in Database Applications"](#)
- [Chapter 9, "Using Indexes in Database Applications"](#)
- [Chapter 10, "Maintaining Data Integrity in Database Applications"](#)

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

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 statements and SQL processing in *Oracle Database Concepts*.

Topics:

- [Description of SQL Statement Processing](#)
- [Grouping Operations into Transactions](#)
- [Ensuring Repeatable Reads with Read-Only Transactions](#)
- [Locking Tables Explicitly](#)
- [Using Oracle Lock Management Services \(User Locks\)](#)
- [Using Serializable Transactions for Concurrency Control](#)
- [Nonblocking and Blocking DDL Statements](#)
- [Autonomous Transactions](#)
- [Resuming Execution After Storage Allocation Errors](#)

6.1 Description of SQL Statement Processing

This topic explains what happens during each stage of processing the execution of a SQL statement, using a data manipulation language (DML) statement as an example.

Assume that you are using a Pro*C program to increase the salary for all employees in a department. The program has connected to Oracle Database and you are connected to the HR schema, which owns the `employees` table. You can embed this SQL statement in your program:

```
EXEC SQL UPDATE employees SET salary = 1.10 * salary
    WHERE department_id = :department_id;
```

The program provides a value for the bind variable placeholder `:department_id`, which the SQL statement uses when it runs.

Topics:

- [Stages of SQL Statement Processing](#)
- [Shared SQL Areas](#)

6.1.1 Stages of SQL Statement Processing

Note: DML statements use all stages. Transaction management, session management, and system management SQL statements use only stages 2 and 8.

1. Open or create a cursor.

A program interface call opens or creates a cursor, in expectation of a SQL statement. Most applications create the cursor implicitly (automatically). Precompiler programs can create the cursor either implicitly or explicitly.

2. Parse the statement.

The user process passes the SQL statement to Oracle Database, which loads a parsed representation of the statement into the shared SQL area. Oracle Database can catch many errors during parsing.

Note: For a data definition language (DDL) statement, parsing includes data dictionary lookup and execution.

See Also:

- *Oracle Database Concepts* for information about parsing
- [Section 6.1.2, "Shared SQL Areas"](#)

3. Determine if the statement is a query.

4. If the statement is a query, describe its results.

Note: This stage is necessary only if the characteristics of the result are unknown; for example, when a user enters the query interactively.

Oracle Database determines the characteristics (data types, lengths, and names) of the result.

5. If the statement is a query, define its output.

You specify the location, size, and data type of variables defined to receive each fetched value. These variables are called **define variables**. Oracle Database performs data type conversion if necessary.

See Also: *Oracle Database Concepts* for information about the **DEFINE** stage

6. Bind any variables.

Oracle Database has determined the meaning of the SQL statement but does not have enough information to run it. Oracle Database needs values for any bind variable placeholders in the statement. In the example, Oracle Database needs a value for `:department_id`. The process of obtaining these values is called **binding variables**.

A program must specify the location (memory address) of the value. End users of applications may be unaware that they are specifying values for bind variable placeholders, because the Oracle Database utility can prompt them for the values.

Because the program specifies the location of the value (that is, binds by reference), it need not rebind the variable before rerunning the statement, even if the value changes. Each time Oracle Database runs the statement, it gets the value of the variable from its address.

You must also specify a data type and length for each value (unless they are implied or defaulted) if Oracle Database must perform data type conversion.

See Also: For more information about specifying a data type and length for a value:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*

7. (Optional) Parallelize the statement.

Oracle Database can parallelize queries and some data definition language (DDL) operations (for example, index creation, creating a table with a subquery, and operations on partitions). Parallelization causes multiple server processes to perform the work of the SQL statement so that it can complete faster.

8. Run the statement.

Oracle Database runs the statement. If the statement is a query or an `INSERT` statement, the database locks no rows, because no data is changing. If the statement is an `UPDATE` or `DELETE` statement, the database locks all rows that the statement affects, until the next `COMMIT`, `ROLLBACK`, or `SAVEPOINT` for the transaction, thereby ensuring data integrity.

For some statements, you can specify multiple executions to be performed. This is called **array processing**. Given n number of executions, the bind and define locations are assumed to be the beginning of an array of size n .

9. If the statement is a query, fetch its rows.

Oracle Database selects rows and, if the query has an `ORDER BY` clause, orders the rows. Each successive fetch retrieves another row of the result set, until the last row has been fetched.

10. Close the cursor.

Oracle Database closes the cursor.

Note: To rerun a transaction management, session management, or system management SQL statement, use another `EXECUTE` statement.

6.1.2 Shared SQL Areas

Oracle Database automatically detects when applications send similar SQL statements to the database. The SQL area used to process the first occurrence of the statement is *shared*—that is, used for processing subsequent occurrences of that same statement. Therefore, only one shared SQL area exists for a unique statement. Because shared SQL areas are shared memory areas, any Oracle Database process can use a shared SQL area. The sharing of SQL areas reduces memory use on the database server, thereby increasing system throughput.

In determining whether statements are similar or identical, Oracle Database compares both SQL statements issued directly by users and applications and recursive SQL statements issued internally by DDL statements.

See Also:

- *Oracle Database Concepts* for more information about shared SQL areas
- *Oracle Database SQL Tuning Guide* for more information about shared SQL

6.2 Grouping Operations into Transactions

Topics:

- [Deciding How to Group Operations in Transactions](#)
- [Improving Transaction Performance](#)
- [Managing Commit Redo Action](#)
- [Determining Transaction Outcome After a Recoverable Outage](#)

See Also: *Oracle Database Concepts* for basic information about transactions

6.2.1 Deciding How to Group Operations in Transactions

Typically, deciding how to group operations in transactions is the concern of application developers who use 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 lets users transfer funds between accounts. The transaction must include the debit to one account, executed by one SQL statement, and the credit to another account, executed by another SQL statement. Both statements must fail or succeed as a unit of work; one statement must not be committed without the other. Do not include unrelated actions, such as a deposit to one account, in the transaction.

6.2.2 Improving Transaction Performance

As an application developer, you must try to improve performance. Consider using these performance enhancement techniques when designing and writing your application:

- For each transaction:
 1. If you can use a single SQL statement, then do so.
 2. If you cannot use a single SQL statement but you can use PL/SQL, then use as little PL/SQL as possible.

For information about PL/SQL, see [Part III, "PL/SQL for Application Developers"](#).

3. If you cannot use PL/SQL (because it cannot do what you must do; for example, read a directory), then use Java.
4. If you cannot use Java (for example, if it is too slow) or you have existing third-generation language (3GL) code, then use an external C subprogram.

For information about using Java and C in your application, see [Chapter 18, "Developing Applications with Multiple Programming Languages."](#)

- Establish standards for writing SQL statements so that you can take advantage of shared SQL areas.

Oracle Database recognizes identical SQL statements and lets them share memory areas, reducing memory usage on the database server and increasing system throughput.

- Collect statistics that Oracle Database can use to implement a cost-based approach to SQL statement optimization, and use additional hints to the optimizer as needed.

To collect most statistics, use the `DBMS_STATS` package, which lets you collect statistics in parallel, collect global statistics for partitioned objects, and tune your statistics collection in other ways. For more information about this package, see *Oracle Database PL/SQL Packages and Types Reference*.

To collect statistics unrelated to the cost-based optimizer (such as information about free list blocks), use the SQL statement `ANALYZE`. For more information about this statement, see *Oracle Database SQL Language Reference*.

For more information about hints, see *Oracle Database SQL Language Reference*.

- Before beginning a transaction, invoke `DBMS_APPLICATION_INFO` procedures to record the name of the transaction in the database for later use when tracking its performance with Oracle Trace and the SQL trace facility. For information about the `DBMS_APPLICATION_INFO` package, see *Oracle Database PL/SQL Packages and Types Reference*.
- Increase user productivity and query efficiency by including user-written PL/SQL functions in SQL expressions. For details, see [Section 11.9, "Invoking Stored PL/SQL Functions from SQL Statements."](#)

See Also: *Oracle Database Concepts* for more information about transaction management

6.2.3 Managing Commit Redo Action

When a transaction updates Oracle Database, it generates a corresponding redo entry. Oracle Database buffers the redo entry to the redo log until the transaction completes. When the transaction commits, the log writer process (LGWR) writes redo records to disk for the buffered redo entries of all changes in the transaction. By default, Oracle Database writes the redo entries to disk before the call returns to the client. This action causes a latency in the commit, because the application must wait for the redo entries to be persistent on disk.

Oracle Database lets you change the handling of commit redo to fit the needs of your application. If your application requires very high transaction throughput and you are willing to trade commit durability for lower commit latency, then you can change the default `COMMIT` options so that the application need not wait for the database to write data to the online redo logs.

Table 6–1 describes the COMMIT options.

Caution: With the NOWAIT option, a failure that occurs after the commit message is received, but before the redo log records are written, can falsely indicate to a transaction that its changes are persistent.

Table 6–1 COMMIT Statement Options

Option	Effect
WAIT (default)	Ensures that the COMMIT statement 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 the transaction committed.
NOWAIT (alternative to WAIT)	The COMMIT statement returns to the client regardless of whether the write to the redo log has completed. This behavior can increase transaction throughput.
BATCH (alternative to IMMEDIATE)	Buffers the redo information to the redo log with concurrently running transactions. After collecting sufficient redo information, initiates a disk write to the redo log. This behavior is called group commit , because it writes redo information for multiple transactions to the log in a single I/O operation.
IMMEDIATE (default)	LGWR writes the transaction redo information to the log. Because this operation option forces a disk I/O, it can reduce transaction throughput.

To change the COMMIT options, use either the COMMIT statement (described in *Oracle Database SQL Language Reference*) or the appropriate initialization parameter. For information about initialization parameters, see *Oracle Database Reference*.

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

If your application uses Oracle Call Interface (OCI), then you can modify redo action by setting these flags in the OCITransCommit function in your application:

- OCI_TRANS_WRITEWAIT
- OCI_TRANS_WITENOWAIT
- OCI_TRANS_WRITEBATCH
- OCI_TRANS_WRITEIMMED

Caution: OCI_TRANS_WITENOWAIT can cause silent transaction loss with shutdown termination, startup force, and any instance or node failure. On an Oracle RAC system, asynchronously committed changes might not be immediately available to read on other instances.

See Also: *Oracle Call Interface Programmer's Guide* for information about the OCITransCommit function

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

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

6.2.4 Determining Transaction Outcome After a Recoverable Outage

A **recoverable outage** is a system, hardware, communication, or storage failure that breaks the connection between your application (the client) and Oracle Database (the server). After an outage, your application receives a disconnection error message. The transaction that was running when the connection broke is the **in-flight transaction**, which may or may not have been committed or run to completion.

To recover from the outage, your application must determine the outcome of the in-flight transaction—whether it was committed and whether it made its intended session state changes. If the transaction was not committed, then the application can either resubmit the transaction or return the uncommitted status to the end user. If the transaction was committed, then the application can return the committed status, rather than the disconnection error, to the end user. If the transaction was both committed and completed, then the application may be able to continue by taking a new session and re-establishing the session state.

The Oracle Database feature that provides your application with the outcome of the in-flight transaction and can be used to ensure that it is not duplicated is Transaction Guard, and its application program interface (API) is the PL/SQL procedure DBMS_APP_CONT.GET_LTXID_OUTCOME.

Topics:

- [Understanding Transaction Guard](#)
- [Understanding DBMS_APP_CONT.GET_LTXID_OUTCOME](#)
- [Using Transaction Guard](#)

6.2.4.1 Understanding Transaction Guard

Transaction Guard is an Oracle Database tool that you can use to provide your application with the outcome of the in-flight transaction after an outage. The application can use Transaction Guard to provide the end user with a known outcome after an outage—committed or not committed—and, optionally, to replay the transaction if it did not commit and the states are correct.

Transaction Guard provides the transaction outcome through its API, the PL/SQL procedure DBMS_APP_CONT.GET_LTXID_OUTCOME, described in [Section 6.2.4.2](#).

Transaction Guard relies on the **logical transaction identifier (LTXID)**, a globally unique identifier that identifies the last in-flight transaction on a session that failed. The database records the LTXID when the transaction is committed, and returns a new LTXID to the client with the commit message (for each client round trip). The client driver always holds the LTXID that will be used at the next COMMIT.

Note:

- Use Transaction Guard only to find the outcome of a session that failed due to a recoverable error, to replace the communication error with the real outcome.
- Do not use Transaction Guard on your own session.
- Do not use Transaction Guard on a live session.

To stop a live session, use `ALTER SYSTEM KILL SESSION IMMEDIATE` at the local or remote instance.

6.2.4.1.1 How Transaction Guard Uses the LTXID

Transaction Guard uses the LTXID as follows:

- While a transaction is running, both Oracle Database (the server) and your application (the client) hold the LTXID to be used at the next `COMMIT`.
- When the transaction is committed, Oracle Database records the LTXID with the transaction. If the LTXID has already been committed or has been blocked, then the database raises error, preventing duplication of the transaction.
- The LTXID persists in Oracle Database for the time specified by the `RETENTION_TIMEOUT` parameter. The default is 24 hours. To change this value:
 - When running Real Application Clusters, use Server Control Utility (SRVCTL), described in *Oracle Real Application Clusters Administration and Deployment Guide*.
 - When not using Real Application Clusters, use the `DBMS_SERVICE` package, described in *Oracle Database PL/SQL Packages and Types Reference*.

If the transaction is remote or distributed, then its LTXID persists in the local database.

The LTXID is transferred to Data Guard and Active Data Guard in the standard redo apply.

- After a recoverable error:
 - If the transaction has not been committed, then Oracle Database blocks its LTXID to ensure that an earlier in-flight transaction with the same LTXID cannot be committed.

This behavior allows the application to return the uncommitted result to the user, who can then decide what to do, and also allows the application to safely replay the application if desirable.

 - If the transaction has been committed, then the application can return this result to the end user, and if the state is correct, the application may be able to continue.
- If the transaction is rolled back, then Oracle Database reuses its LTXID.

See Also:

- [Section 25.3.3, "Transaction Guard Coverage,"](#) for a list of the sources whose commits Transaction Guard supports
- [Section 25.3.4, "Transaction Guard Exclusions,"](#) for a list of the sources whose commits Transaction Guard does not support

6.2.4.2 Understanding DBMS_APP_CONT.GET_LTXID_OUTCOME

The PL/SQL procedure DBMS_APP_CONT.GET_LTXID_OUTCOME is the API of Transaction Guard. After an outage, your application can reconnect to Oracle Database and then invoke this procedure to determine the outcome of the in-flight transaction.

DBMS_APP_CONT.GET_LTXID_OUTCOME has these parameters:

Parameter Name	Data Type	Parameter Mode	Value
CLIENT_LTXID	RAW	IN	LTXID of the in-flight transaction
COMMITTED	BOOLEAN	OUT	TRUE if the in-flight transaction was committed, FALSE otherwise
USER_CALL_COMPLETED	BOOLEAN	OUT	TRUE if the in-flight transaction completed, FALSE otherwise

Topics:

- [CLIENT_LTXID Parameter](#)
- [COMMITTED Parameter](#)
- [USER_CALL_COMPLETED Parameter](#)
- [Exceptions](#)

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about DBMS_APP_CONT.GET_LTXID_OUTCOME

6.2.4.2.1 CLIENT_LTXID Parameter Before your application (the client) can invoke DBMS_APP_CONT.GET_LTXID_OUTCOME with the LTXID of the in-flight transaction, it must get the last LTXID in use at the client driver by using a client driver. The client driver holds the LTXID of the transaction next to be committed. In this case, the LTXID is for the in-flight transaction at the time of the outage. Use `getLTXID` for JDBC-Thin driver, `LogicalTransactionId` for ODP.NET, and `OCI_ATTR_GET` with `LTXID` for OCI and OCCI.

The JDBC-Thin driver also provides a callback that is triggered each time the LTXID at the client driver changes. The callback can be used to maintain the current LTXID to be used. The callback is particularly useful for application servers and applications that must block repeated executions.

Note: Your application must get the LTXID immediately before passing it to DBMS_APP_CONT.GET_LTXID_OUTCOME. Getting the LTXID in advance could lead to passing an earlier LTXID to DBMS_APP_CONT.GET_LTXID_OUTCOME, causing the request to be rejected.

For information about the client driver for your client driver, see the documentation for your client driver. For example:

- For JDBC type 2 and type 4 drivers, see *Oracle Database JDBC Developer's Guide*.
- For the OCI driver (including OCCI), see *Oracle Call Interface Programmer's Guide*.
- For the ODP.NET driver, see *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*.

6.2.4.2.2 COMMITTED Parameter After `DBMS_APP_CONT.GET_LTXID_OUTCOME` returns control to your application, your application can check the value of the actual parameter that corresponds to the formal parameter `COMMITTED` to determine whether the in-flight transaction was committed.

If the value of the actual parameter is `TRUE`, then the transaction was committed.

If the value of the actual parameter is `FALSE`, then the transaction was not committed. Therefore, it is safe for the application to return the code `UNCOMMITTED` to the end user or use it to replay the transaction.

To ensure that an earlier session does not commit the transaction after the application returns `UNCOMMITTED`, `DBMS_APP_CONT.GET_LTXID_OUTCOME` blocks the LTXID. Blocking the LTXID allows the end user to make a decision based on the uncommitted status, or the application to replay the transaction, and prevents duplicate transactions.

6.2.4.2.3 USER_CALL_COMPLETED Parameter Some transactions return information upon completion. For example: A transaction that uses commit on success (auto-commit) might return the number of affected rows, or for a `SELECT` statement, the rows themselves; a transaction that invokes a PL/SQL subprogram that has `OUT` parameters returns the values of those parameters; and a transaction that invokes a PL/SQL function returns the function value. Also, a transaction that invokes a PL/SQL subprogram might execute a `COMMIT` statement and then do more work.

If your application needs information that the in-flight transaction returns upon completion, or session state changes that the transaction does after committing its database changes, then your application must determine whether the in-flight transaction completed, which it can do by checking the value of the actual parameter that corresponds to the formal parameter `USER_CALL_COMPLETED`.

If the value of the actual parameter is `TRUE`, then the transaction completed, and your application has the information and work that it must continue.

If the value of the actual parameter is `FALSE`, then the call from the client may not have completed. Therefore, your application might not have the information and work that it must continue.

6.2.4.2.4 Exceptions If your application (the client) and Oracle Database (the server) are no longer synchronized, then the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure raises one of these exceptions:

Exception	Explanation
ORA-14950 – SERVER_AHEAD	The server is ahead of the client; that is, the LTXID that your application passed to <code>DBMS_APP_CONT.GET_LTXID_OUTCOME</code> identifies a transaction that is older than the in-flight transaction. Your application must get the LTXID immediately before passing it to <code>DBMS_APP_CONT.GET_LTXID_OUTCOME</code> .
ORA-14951 – CLIENT_AHEAD	The client is ahead of the server. Either the server was "flashed back" to an earlier state (see Chapter 16, "Using Oracle Flashback Technology"), was recovered using media recovery, or is a standby database that was opened earlier and has lost data (for information about media recovery and standby databases, see <i>Oracle Data Guard Concepts and Administration</i>).
ORA-14906 – SAME_SESSION	Executing <code>GET_LTXID_OUTCOME</code> is not supported on the session that owns the LTXID, because the execution would block further processing on that session.

Exception	Explanation
ORA-14909 - COMMIT_BLOCKED	Your session has been blocked from committing by another user with the same username using GET_LTXID_OUTCOME. GET_LTXID_OUTCOME should be called only on dead sessions. Blocking a live session is better achieved using ALTER SYSTEM KILL SESSION IMMEDIATE. For help, contact your application administrator.
ORA-14952 GENERAL_ERROR	DBMS_APP_CONT.GET_LTXID_OUTCOME cannot determine the outcome of the in-flight transaction. An error occurred during transaction processing, and the error stack shows the error detail.

6.2.4.3 Using Transaction Guard

After your application (the client) receives an error message, it must follow these steps to use Transaction Guard:

1. Determine if the error is due to an outage (**recoverable**).

For instructions, see the documentation for your client driver—OCI_ATTRIBUTE for OCI, OCCI, and ODP.NET; isRecoverable for JDBC.

2. If the error is recoverable, then use the API of the client driver to get the logical transaction identifier (LTXID) of the in-flight transaction.

For instructions, see the documentation for your client driver.

3. Reconnect to the database.

The session that your application acquires can be either new or pooled.

4. Invoke DBMS_APP_CONT.GET_LTXID_OUTCOME with the LTXID from step 2.

5. Check the value of the actual parameter that corresponds to the formal parameter COMMITTED.

If the value is TRUE, then tell the application that the in-flight transaction was committed. The application can return this result to the user, or continue if the state is correct.

If the value is FALSE, then the application can return UNCOMMITTED or a similar message to the user so that the user can choose the next step. Optionally, the application can replay the transaction for the user. For example:

1. If necessary, clean up state changes on the client side.

2. Resubmit the in-flight transaction.

If you do not resubmit the in-flight transaction, and the application needs neither information that the in-flight transaction returns upon completion nor work that the transaction does after committing its database changes, then continue.

Otherwise, check the value of the actual parameter that corresponds to the formal parameter USER_CALL_COMPLETED.

If the value is TRUE, then continue.

If the value is FALSE, then tell the application user that the application cannot continue.

6.3 Ensuring Repeatable Reads with Read-Only Transactions

By default, Oracle Database guarantees statement-level read consistency, but not transaction-level read consistency. With **statement-level read consistency**, queries in a statement produce consistent data for the duration of the statement, not reflecting

changes by other statements. With **transaction-level read consistency (repeatable reads)**, queries in the transaction produce consistent data for the duration of the transaction, not reflecting changes by other transactions.

To ensure transaction-level read consistency for a transaction that does not include DML statements, specify that the transaction is read-only. The queries in a read-only transaction see only changes committed before the transaction began, so query results are consistent for the duration of the transaction.

A read-only transaction provides transaction-level read consistency without acquiring additional data locks. Therefore, while the read-only transaction is querying data, other transactions can query and update the same data.

A read-only transaction begins with this statement:

```
SET TRANSACTION READ ONLY [ NAME string ];
```

Only DDL statements can precede the SET TRANSACTION READ ONLY statement. After the SET TRANSACTION READ ONLY statement successfully runs, the transaction can include only SELECT (without FOR UPDATE), COMMIT, ROLLBACK, or non-DML statements (such as SET ROLE, ALTER SYSTEM, and LOCK TABLE). A COMMIT, ROLLBACK, or DDL statement ends the read-only transaction.

See Also: *Oracle Database SQL Language Reference* for more information about the SET TRANSACTION statement

Long-running queries sometimes fail because undo information required for consistent read (CR) operations is no longer available. This situation occurs when active transactions overwrite committed undo blocks.

Automatic undo management lets your database administrator (DBA) explicitly control how long the database retains undo information, using the parameter UNDO_RETENTION. For example, if UNDO_RETENTION is 30 minutes, then the database retains all committed undo information for at least 30 minutes, ensuring that all queries running for 30 minutes or less do not encounter the OER error "snapshot too old."

See Also:

- *Oracle Database Concepts* for more information about read consistency
- *Oracle Database Administrator's Guide* for information about long-running queries and resumable space allocation

6.4 Locking Tables Explicitly

Oracle Database has default locking mechanisms that ensure data concurrency, data integrity, and statement-level read consistency (for information about these, see *Oracle Database Concepts*). However, you can override these mechanisms by locking tables explicitly. Locking tables explicitly is useful in situations such as these:

- A transaction in your application needs exclusive access to a resource, so that the transaction does not have to wait for other transactions to complete.
- Your application needs transaction-level read consistency (repeatable reads).

For other ways to ensure transaction-level read consistency, see [Section 6.3, "Ensuring Repeatable Reads with Read-Only Transactions,"](#) and [Section 6.6, "Using Serializable Transactions for Concurrency Control."](#)

To override default locking at the transaction level, use any of these SQL statements:

- `LOCK TABLE` (described in *Oracle Database SQL Language Reference*)
- `SELECT` with the `FOR UPDATE` clause (described in *Oracle Database SQL Language Reference*)
- `SET TRANSACTION` with the `READ ONLY` or `ISOLATION LEVEL SERIALIZABLE` option (described in *Oracle Database SQL Language Reference*)

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

See Also:

- *Oracle Database SQL Language Reference* for information about the `ISOLATION_LEVEL` parameter of the `ALTER SESSION` statement

The initialization parameter `DML_LOCKS` (described in *Oracle Database Reference*) determines the maximum number of DML locks. Although its default value is usually enough, you might need to increase it if you use explicit locks.

Caution: If you override the default locking of Oracle Database at any level, ensure that data integrity is guaranteed, data concurrency is acceptable, and deadlocks are either impossible or appropriately handled.

Topics:

- [Privileges Required to Acquire Table Locks](#)
- [Choosing a Locking Strategy](#)
- [Letting Oracle Database Control Table Locking](#)
- [Explicitly Acquiring Row Locks](#)
- [Examples of Concurrency Under Explicit Locking](#)

6.4.1 Privileges Required to Acquire Table Locks

No special privileges are required to acquire any type of table lock on a table in your own schema. To acquire a table lock on a table in another schema, you must have either the `LOCK ANY TABLE` system privilege or any object privilege (for example, `SELECT` or `UPDATE`) for the table.

6.4.2 Choosing a Locking Strategy

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

```
LOCK TABLE employees, departments 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. For information about locking individual rows, see [Section 6.4.4](#).

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 unavailable now.
- 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 unavailable now.
- 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 running DDL statements concurrently with DML statements, a timeout or deadlock can sometimes result. The database detects such timeouts and deadlocks and returns an error.

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](#)

See Also: *Oracle Database SQL Language Reference* for LOCK TABLE statement syntax

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

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

- Your transaction must prevent another transaction from acquiring an intervening share, share row, or exclusive table lock for a table before your transaction can update that table.
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 must prevent a table from being altered or dropped before your transaction can modify that table.

6.4.2.2 When to Lock with SHARE MODE

SHARE MODE 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 MODE locks on the table either commit or roll back.
- Other transactions might acquire concurrent SHARE MODE table locks on the same table, also giving them the option of transaction-level read consistency.

Caution: Your transaction 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.

Scenario: Tables employees and budget_tab require a consistent set of data in a third table, departments. For a given department number, you want to update the information in employees and budget_tab, and ensure that no members are added to the department between these two transactions.

Solution: Lock the departments table in SHARE MODE, as shown in [Example 6-1](#). Because the departments table is rarely updated, locking it probably does not cause many other transactions to wait long.

Example 6-1 LOCK TABLE with SHARE MODE

```
-- Create and populate table:
DROP TABLE budget_tab;
CREATE TABLE budget_tab (
    sal      NUMBER(8,2),
    deptno  NUMBER(4)
);

INSERT INTO budget_tab (sal, deptno)
    SELECT salary, department_id
    FROM employees;

-- Lock departments and update employees and budget_tab:
LOCK TABLE departments IN SHARE MODE;

UPDATE employees
    SET salary = salary * 1.1
    WHERE department_id IN
        (SELECT department_id FROM departments WHERE location_id = 1700);

UPDATE budget_tab
    SET sal = sal * 1.1
    WHERE deptno IN
        (SELECT department_id FROM departments WHERE location_id = 1700);

COMMIT; -- COMMIT releases lock
```

6.4.2.3 When to Lock with SHARE ROW EXCLUSIVE MODE

You might use a SHARE ROW EXCLUSIVE MODE 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 want only a single transaction to have this action.

6.4.2.4 When to Lock with EXCLUSIVE MODE

You might use an EXCLUSIVE MODE 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.

6.4.3 Letting Oracle Database Control Table Locking

If you let Oracle Database control table locking, 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 gives concurrent access to the table while providing ANSI serializability. Getting table locks greatly reduces concurrency.

See Also:

- *Oracle Database SQL Language Reference* for information about the `SET TRANSACTION` statement
- *Oracle Database SQL Language Reference* for information about 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.

6.4.4 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 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 let a user 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 released only 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 much 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 read-consistent snapshot of the table using these locks, and then restarts the query to acquire the remaining locks.

If your application uses the `SELECT FOR UPDATE` statement and cannot guarantee that a conflicting locking request will not result in user-caused deadlocks—for example, through ensuring that concurrent DML statements on a table never affect the return set of the query of a `SELECT FOR UPDATE` statement—then code the application always to handle such a deadlock (ORA-00060) in an appropriate manner.

By default, the `SELECT FOR UPDATE` statement waits until the requested row lock is acquired. To change this behavior, use the `NOWAIT`, `WAIT`, or `SKIP LOCKED` clause of the `SELECT FOR UPDATE` statement. For information about these clauses, see *Oracle Database SQL Language Reference*.

6.4.5 Examples of Concurrency Under Explicit Locking

Table 6-2 shows how Oracle Database maintains data concurrency, integrity, and consistency when the `LOCK TABLE` statement and the `SELECT` statement with the `FOR UPDATE` clause are used. For brevity, the message text for ORA-00054 ("resource busy and acquire with `NOWAIT` specified") is not included. User-entered text is **bold**.

Note: In tables compressed with Hybrid Columnar Compression (HCC), DML statements lock compression units rather than rows. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

Table 6–2 Examples of Concurrency Under Explicit Locking

Transaction 1	Time Point	Transaction 2
<code>LOCK TABLE hr.departments IN ROW SHARE MODE;</code>	1	
Statement processed.		
	2	<code>DROP TABLE hr.departments;</code>
		<code>DROP TABLE hr.departments *</code>
		ORA-00054
		(Exclusive DDL lock not possible because Transaction 1 has table locked.)
	3	<code>LOCK TABLE hr.departments IN EXCLUSIVE MODE NOWAIT;</code>
		ORA-00054
	4	<code>SELECT location_id FROM hr.departments WHERE department_id = 20 FOR UPDATE OF location_id;</code>
		<code>LOCATION_ID ----- DALLAS</code>
		1 row selected.
<code>UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 20;</code>	5	
(Waits because Transaction 2 locked same rows.)		
	6	<code>ROLLBACK;</code>
		(Releases row locks.)
1 row processed.	7	
<code>ROLLBACK;</code>		
<code>LOCK TABLE hr.departments IN ROW EXCLUSIVE MODE;</code>	8	
Statement processed.		
	9	<code>LOCK TABLE hr.departments IN EXCLUSIVE MODE NOWAIT;</code>
		ORA-00054

Table 6–2 (Cont.) Examples of Concurrency Under Explicit Locking

Transaction 1	Time Point	Transaction 2
	10	LOCK TABLE hr.departments IN SHARE ROW EXCLUSIVE MODE NOWAIT;
		ORA-00054
	11	LOCK TABLE hr.departments IN SHARE ROW EXCLUSIVE MODE NOWAIT;
		ORA-00054
	12	UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 20;
		1 row processed.
	13	ROLLBACK;
SELECT location_id FROM hr.departments WHERE department_id = 20 FOR UPDATE OF location_id;	14	
LOCATION_ID		

DALLAS		
1 row selected.		
	15	UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 20;
		1 row processed.
		(Waits because Transaction 1 locked same rows.)
ROLLBACK;	16	
	17	1 row processed.
		(Conflicting locks were released.)
		ROLLBACK;
LOCK TABLE hr.departments IN ROW SHARE MODE	18	
Statement processed.		
	19	LOCK TABLE hr.departments IN EXCLUSIVE MODE NOWAIT;
		ORA-00054

Table 6–2 (Cont.) Examples of Concurrency Under Explicit Locking

Transaction 1	Time Point	Transaction 2
	20	LOCK TABLE hr.departments IN SHARE ROW EXCLUSIVE MODE NOWAIT;
		ORA-00054
	21	LOCK TABLE hr.departments IN SHARE MODE;
		Statement processed.
	22	SELECT location_id FROM hr.departments WHERE department_id = 20;
		LOCATION_ID ----- DALLAS
		1 row selected.
	23	SELECT location_id FROM hr.departments WHERE department_id = 20 FOR UPDATE OF location_id;
		LOCATION_ID ----- DALLAS
		1 row selected.
	24	UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 20;
		(Waits because Transaction 1 has conflicting table lock.)
ROLLBACK;	25	
	26	1 row processed.
		(Conflicting table lock released.)
		ROLLBACK;
LOCK TABLE hr.departments IN SHARE ROW EXCLUSIVE MODE;	27	
Statement processed.		
	28	LOCK TABLE hr.departments IN EXCLUSIVE MODE NOWAIT;
		ORA-00054

Table 6–2 (Cont.) Examples of Concurrency Under Explicit Locking

Transaction 1	Time Point	Transaction 2
	29	<code>LOCK TABLE hr.departments IN SHARE ROW EXCLUSIVE MODE NOWAIT;</code>
		ORA-00054
	30	<code>LOCK TABLE hr.departments IN SHARE MODE NOWAIT;</code>
		ORA-00054
	31	<code>LOCK TABLE hr.departments IN ROW EXCLUSIVE MODE NOWAIT;</code>
		ORA-00054
	32	<code>LOCK TABLE hr.departments IN SHARE MODE NOWAIT;</code>
		ORA-00054
	33	<code>SELECT location_id FROM hr.departments WHERE department_id = 20;</code>
		LOCATION_ID ----- DALLAS
		1 row selected.
	34	<code>SELECT location_id FROM hr.departments WHERE department_id = 20 FOR UPDATE OF location_id;</code>
		LOCATION_ID ----- DALLAS
		1 row selected.
	35	<code>UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 20;</code>
		(Waits because Transaction 1 has conflicting table lock.)
<code>UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 20;</code>	36	(Deadlock.)
		(Waits because Transaction 2 locked same rows.)

Table 6–2 (Cont.) Examples of Concurrency Under Explicit Locking

Transaction 1	Time Point	Transaction 2
Cancel operation.	37	
ROLLBACK;	38	1 row processed.
LOCK TABLE hr.departments IN EXCLUSIVE MODE;	39	
	40	LOCK TABLE hr.departments IN EXCLUSIVE MODE;
		ORA-00054
	41	LOCK TABLE hr.departments IN ROW EXCLUSIVE MODE NOWAIT;
		ORA-00054
	42	LOCK TABLE hr.departments IN SHARE MODE;
		ORA-00054
	43	LOCK TABLE hr.departments IN ROW EXCLUSIVE MODE NOWAIT;
		ORA-00054
	44	LOCK TABLE hr.departments IN ROW SHARE MODE NOWAIT;
		ORA-00054
	45	SELECT location_id FROM hr.departments WHERE department_id = 20;
		LOCATION_ID ----- DALLAS
		1 row selected.
	46	SELECT location_id FROM hr.departments WHERE department_id = 20 FOR UPDATE OF location_id;
		(Waits because Transaction 1 has conflicting table lock.)
UPDATE hr.departments SET department_id = 30 WHERE department_id = 20;	47	
1 row processed.		

Table 6–2 (Cont.) Examples of Concurrency Under Explicit Locking

Transaction 1	Time Point	Transaction 2
COMMIT;	48	
	49	0 rows selected.
		(Transaction 1 released conflicting lock.)
SET TRANSACTION READ ONLY;	50	
SELECT location_id FROM hr.departments WHERE department_id = 10;	51	
LOCATION_ID		

BOSTON		
	52	UPDATE hr.departments SET location_id = 'NEW YORK' WHERE department_id = 10;
		1 row processed.
SELECT location_id FROM hr.departments WHERE department_id = 10;	53	
LOCATION_ID		

BOSTON		
(Transaction 1 does not see uncommitted data.)		
	54	COMMIT;
SELECT location_id FROM hr.departments WHERE department_id = 10;	55	
LOCATION_ID		

BOSTON		
(Same result even after Transaction 2 commits.)		
COMMIT;	56	
SELECT location_id FROM hr.departments WHERE department_id = 10;	57	
LOCATION_ID		

NEW YORK		
(Sees committed data.)		

6.5 Using Oracle Lock Management Services (User Locks)

Your applications can use Oracle Lock Management services (user locks) by invoking subprograms the DBMS_LOCK package. An application can 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 an Oracle Database lock, it has all the features of a database lock, such as deadlock detection. Ensure that any user locks used in distributed transactions are released upon COMMIT, otherwise an undetected deadlock can occur.

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

Topics:

- [When to Use User Locks](#)
- [Viewing and Monitoring Locks](#)

6.5.1 When to Use User Locks

User locks can help:

- Provide exclusive access to a device, such as a terminal
- Provide application-level enforcement of read locks
- Detect when a lock is released and clean up after the application
- Synchronize applications and enforce sequential processing

[Example 6–2](#) shows how the Pro*COBOL precompiler uses locks to ensure that there are no conflicts when multiple people must access a single device.

Example 6–2 How the Pro*COBOL Precompiler Uses Locks

```
*****
* 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. One printer is opened by all *
* the cashiers to avoid the overhead of opening and closing it *
* for every check, meaning that lines of output from multiple *
* cashiers can become interleaved if you 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.
*      You 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.
```

6.5.2 Viewing and Monitoring Locks

Table 6-3 describes the Oracle Database facilities that display locking information for ongoing transactions within an instance.

Table 6-3 Ways to Display Locking Information

Tool	Description
Performance Monitoring Data Dictionary Views	See <i>Oracle Database Administrator's Guide</i> .
UTLLOCKT.SQL	The UTLLOCKT.SQL script displays a simple character lock wait-for graph in tree structured fashion. Using any SQL tool (such as SQL*Plus) to run 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.)

6.6 Using Serializable Transactions for Concurrency Control

By default, Oracle Database permits concurrently running transactions to modify, add, or delete rows in the same table, and in the same data block. When transaction A changes a table, the changes are invisible to concurrently running transactions until transaction A commits them. If transaction A tries to update or delete a row that transaction B has locked (by issuing a DML or SELECT FOR UPDATE statement), then the DML statement that A issued waits until B either commits or rolls back the transaction. This concurrency model, which provides higher concurrency and thus better performance, is appropriate for most applications.

However, some rare applications require serializable transactions. **Serializable transactions** run concurrently in serialized mode. In **serialized mode**, concurrent transactions can make only the database changes that they could make if they were running serially (that is, one at a time). If a serialized transaction tries to change data that another transaction changed after the serialized transaction began, then error ORA-08177 occurs.

When a serializable transaction fails with ORA-08177, the application can take any of these actions:

- Commit the work executed to that point.
- Run additional, different, statements, perhaps after rolling back to a prior savepoint in the transaction.
- Roll back the transaction and then rerun it.

The transaction gets a transaction snapshot and the operation is likely to succeed.

Tip: To minimize the performance overhead of rolling back and re-running transactions, put DML statements that might conflict with concurrent transactions near the beginning of the transaction.

Note: Serializable transactions do not work with deferred segment creation or interval partitioning. Trying to insert data into an empty table with no segment created, or into a partition of an interval partitioned table that does not yet have a segment, causes an error.

Topics:

- [Transaction Interaction and Isolation Level](#)
- [Setting Isolation Levels](#)
- [Serializable Transactions and Referential Integrity](#)
- [READ COMMITTED and SERIALIZABLE Isolation Levels](#)

6.6.1 Transaction Interaction and Isolation Level

The ANSI/ISO SQL standard defines three kinds of transaction interaction:

Transaction Interaction	Definition
Dirty read	Transaction A reads uncommitted changes made by transaction B.
Unrepeatable read	Transaction A reads data, transaction B changes the data and commits the changes, and transaction A rereads the data and sees the changes.
Phantom read	Transaction A runs a query, transaction B inserts new rows and commits the change, and transaction A repeats the query and sees the new rows.

The kinds of interactions that a transaction can have is determined by its isolation level. The ANSI/ISO SQL standard defines four transaction isolation levels. [Table 6-4](#) shows what kind of interactions are possible at each isolation level.

Table 6-4 ANSI/ISO SQL Isolation Levels and Possible Transaction Interactions

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
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

[Table 6-5](#) shows which ANSI/ISO SQL transaction isolation levels Oracle Database provides.

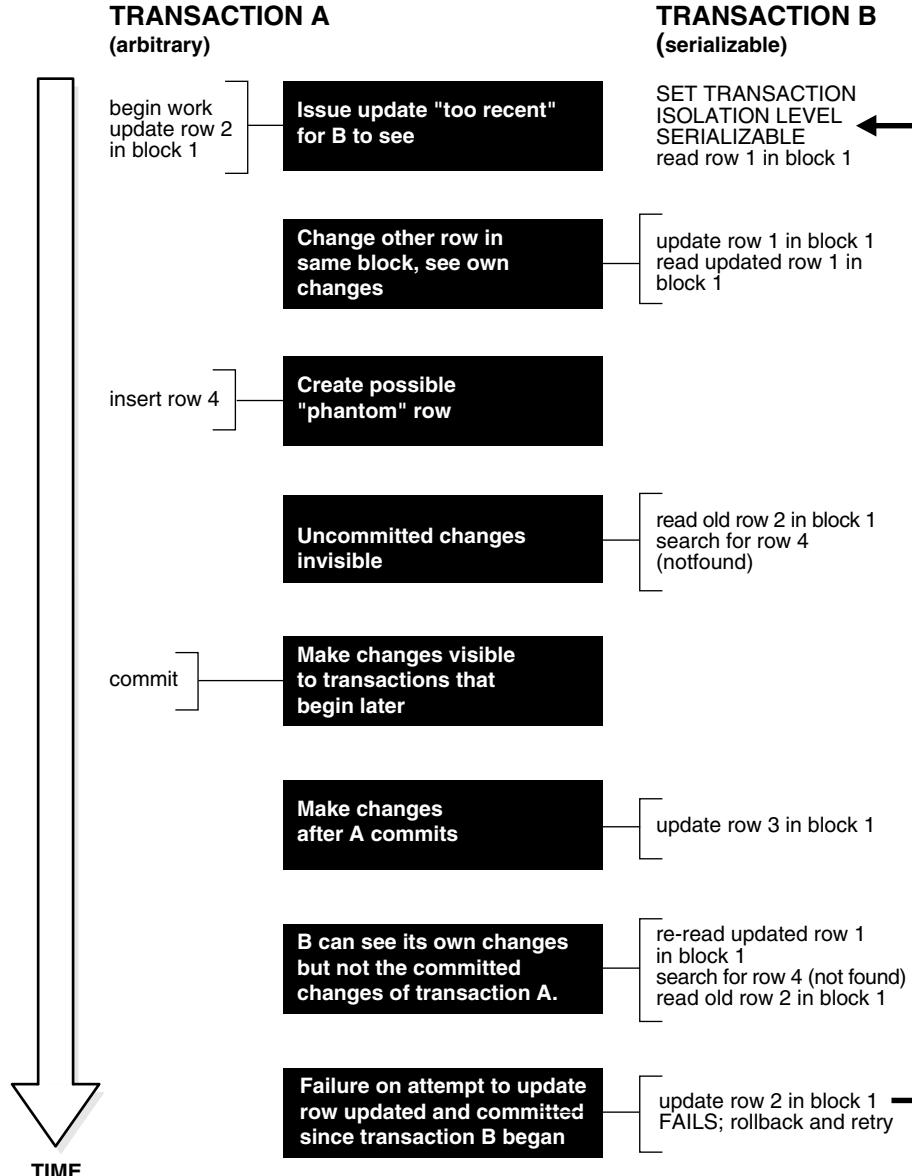
Table 6-5 ANSI/ISO SQL Isolation Levels Provided by Oracle Database

Isolation Level	Provided by Oracle Database
READ UNCOMMITTED	No. Oracle Database never permits "dirty reads." Some other database products use this undesirable technique to improve throughput, but it is not required for high throughput with Oracle Database.
READ COMMITTED	Yes, by default. In fact, because an Oracle Database query sees only data that was committed at the beginning of the query (the snapshot time), Oracle Database offers more consistency than the ANSI/ISO SQL standard for READ COMMITTED isolation requires.

Table 6–5 (Cont.) ANSI/ISO SQL Isolation Levels Provided by Oracle Database

Isolation Level	Provided by Oracle Database
REPEATABLE READ	Yes, if you set the transaction isolation level to SERIALIZABLE.
SERIALIZABLE	Yes, if you set the transaction isolation level to SERIALIZABLE.

Figure 6–1 shows how an arbitrary transaction (that is, one that is either SERIALIZABLE or READ COMMITTED) interacts with a serializable transaction.

Figure 6–1 Interaction Between Serializable Transaction and Another Transaction

6.6.2 Setting Isolation Levels

To set the transaction isolation level for every transaction in your session, use the `ALTER SESSION` statement, described in *Oracle Database SQL Language Reference*.

To set the transaction isolation level for a specific transaction, use the `ISOLATION LEVEL` clause of the `SET TRANSACTION` statement. The `SET TRANSACTION` statement, described in *Oracle Database SQL Language Reference*, must be the first statement in the transaction.

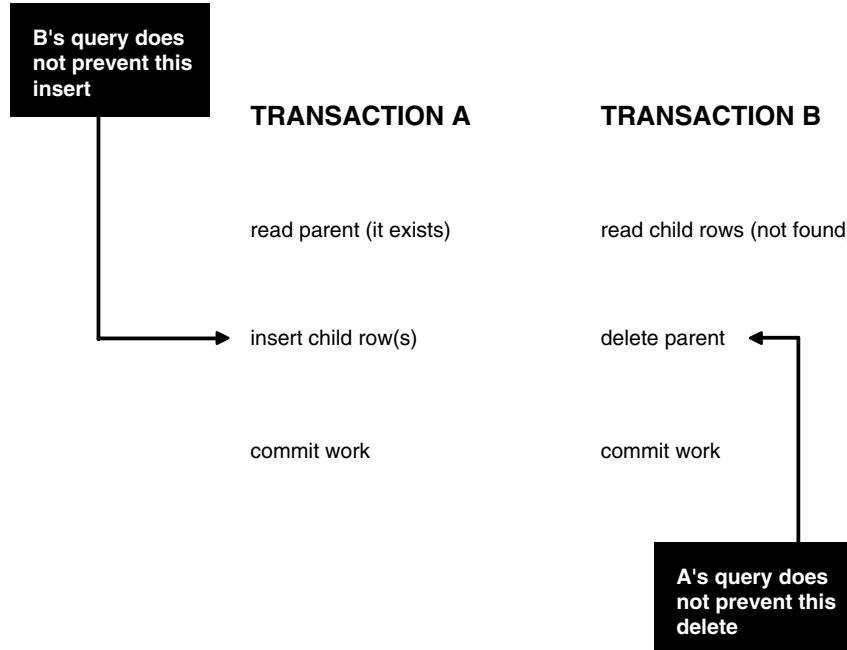
Note: If you set the transaction isolation level to `SERIALIZABLE`, then you must use the `ALTER TABLE` statement to set the `INITTRANS` parameter to at least 3. Use higher values for tables for which many transactions update the same blocks. For more information about `INITTRANS`, see *Oracle Database SQL Language Reference*.

6.6.3 Serializable Transactions and Referential Integrity

Because Oracle Database does not use read locks, even in `SERIALIZABLE` transactions, data read by one transaction can be overwritten by another. Therefore, transactions that perform database consistency checks at the application level must not assume that the data they read does not change during the transaction (even though such changes are invisible to the transaction). Code your application-level consistency checks carefully, even when using `SERIALIZABLE` transactions.

In [Figure 6–2](#), transactions A and B (which are either `READ COMMITTED` or `SERIALIZABLE`) perform application-level checks to maintain the referential integrity of the parent/child relationship between two tables. Transaction A queries the parent table to check that it has a row with a specific primary key value before inserting corresponding child rows into the child table. Transaction B queries the child table to check that no child rows exist for a specific primary key value before deleting the corresponding parent row from the parent table. Both transactions assume (but do not ensure) that the data they read does not change before the transaction completes.

Figure 6–2 Referential Integrity Check



The query by transaction A does not prevent transaction B from deleting the parent row, and the query by transaction B does not prevent transaction A from inserting child rows. Therefore, this can happen:

1. Transaction A queries the parent table and finds the specified parent row.
2. Transaction B queries the child table and finds no child rows for the specified parent row.
3. Having found the specified parent row, transaction A inserts the corresponding child rows into the child table.
4. Having found no child rows for the specified parent row, transaction B deletes the specified parent row from the parent table.

Now the child rows that transaction A inserted in step 3 have no parent row.

The preceding result can occur even if both A and B are `SERIALIZABLE` transactions, because neither transaction prevents the other from changing the data that it reads to check consistency.

Ensuring that data queried by one transaction is not concurrently changed or deleted by another requires more transaction isolation than the ANSI/ISO SQL standard `SERIALIZABLE` isolation level provides. However, in Oracle Database:

- Transaction A can use a `SELECT FOR UPDATE` statement to query and lock the parent row, thereby preventing transaction B from deleting it.
For information about the `FOR UPDATE` clause of the `SELECT` statement, see *Oracle Database SQL Language Reference*.
- Transaction B can prevent transaction A from finding the parent row (thereby preventing A from inserting the child rows) by reversing the order of its processing steps. That is, transaction B can:
 1. Delete the parent row.
 2. Query the child table.
 3. If the deleted parent row has child rows in the child table, then roll back the deletion of the parent row.

Alternatively, you can enforce referential integrity with a trigger. Instead of having transaction A query the parent table, define on the child table a row-level `BEFORE INSERT` trigger that does this:

- Queries the parent table with a `SELECT FOR UPDATE` statement, thereby ensuring that if the parent row exists, then it remains in the database for the duration of the transaction that inserts the child rows.
- Rejects the insertion of the child rows if the parent row does not exist.

See Also: *Oracle Database PL/SQL Language Reference* for more information about using triggers to maintain referential integrity between parent and child tables

A trigger runs SQL statements in the context of the triggering statement (that is, the triggering and triggered statements see the database in the same state). Therefore, if a `READ COMMITTED` transaction runs the triggering statement, then the triggered statements see the database as it was when the triggering statement began to execute. If a `SERIALIZABLE` transaction runs the triggering statement, then the triggered statements see the database as it was at the beginning of the transaction. In either case, using `SELECT FOR UPDATE` in the trigger correctly enforces referential integrity.

6.6.4 READ COMMITTED and SERIALIZABLE Isolation Levels

Oracle Database provides two transaction isolation levels, READ COMMITTED and SERIALIZABLE. Both levels provide a high degree of consistency and concurrency, reduce contention, and are designed for real-world applications. This topic compares them and explains how to choose between them.

Topics:

- [Transaction Set Consistency Differences](#)
- [Choosing Transaction Isolation Levels](#)

6.6.4.1 Transaction Set Consistency Differences

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

Topics:

- [Oracle Database](#)
- [Other Database Systems](#)

6.6.4.1.1 Oracle Database Oracle Database transactions with READ COMMITTED isolation level are transaction set consistent on an individual-statement basis, because all rows that a query reads must be committed before the query begins.

Oracle Database transactions with SERIALIZABLE isolation level are transaction set consistent on an individual-transaction basis, because all statements in a SERIALIZABLE transaction run on an image of the database as it was at the beginning of the transaction.

6.6.4.1.2 Other Database Systems In other database systems, a single query with READ UNCOMMITTED isolation level 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 the reverse). READ COMMITTED isolation level avoids this problem, providing more consistency than read-locking systems do.

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

6.6.4.2 Choosing Transaction Isolation Levels

The choice of transaction isolation level depends on performance and consistency needs and application coding requirements. There is a trade-off between concurrency (transaction throughput) and consistency. Consider the application and workload when choosing isolation levels for its transactions. Different transactions can have different isolation levels.

For environments with many concurrent users rapidly submitting transactions, consider expected transaction arrival rate, response time demands, and required degree of consistency.

READ COMMITTED isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results (from unrepeatable and phantom reads) for some transactions.

SERIALIZABLE isolation provides somewhat more consistency (by protecting against phantoms and unrepeatable reads), which might be important where a read/write transaction runs a query more than once. However, SERIALIZABLE isolation requires applications to check for the "cannot serialize access" error, and this checking can significantly reduce throughput in an environment with many concurrent transactions accessing the same data for update.

As explained in [Section 6.6.3](#) reads do not block writes in either READ COMMITTED or SERIALIZABLE transactions.

Table 6–6 summarizes the similarities and differences between READ COMMITTED and SERIALIZABLE transactions.

Table 6–6 Comparison of 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
Phantom read	Possible	Not Possible
Compliant with ANSI/ISO SQL 92	Yes	Yes
Read snapshot time	Statement	Transaction
Transaction set consistency	Statement level	Transaction level
Row-level locking	Yes	Yes
Readers block writers	No	No
Writers block readers	No	No
Different-row writers block writers	No	No
Same-row writers block writers	Yes	Yes
Waits for blocking transaction	Yes	Yes
Subject to "cannot serialize access" error	No	Yes
Error after blocking transaction terminates	No	No
Error after blocking transaction commits	No	Yes

6.7 Nonblocking and Blocking DDL Statements

The distinction between nonblocking and blocking DDL statements matters only for DDL statements that change either tables or indexes (which depend on tables).

When a session issues a DDL statement that affects object X, the session waits until every concurrent DML statement that references X is either committed or rolled back.

While the session waits, concurrent sessions might issue new DML statements. If the DDL statement is nonblocking, then the new DML statements execute immediately. If

the DDL statement is blocking, then the new DML statements execute after the DDL statement completes, either successfully or with an error.

The `DDL_LOCK_TIMOUT` parameter affects blocking DDL statements (but not nonblocking DDL statements). Therefore, a blocking DDL statement can complete with error ORA-00054 (resource busy and acquire with `NOWAIT` specified or timeout expired). For information about the `DDL_LOCK_TIMOUT` parameter, see *Oracle Database Reference*.

A DDL statement that applies to a partition of a table is blocking for that partition but nonblocking for other partitions of the same table.

For a list of nonblocking DDL statements, see *Oracle Database SQL Language Reference*.

Note: If supplemental logging is enabled at database level (for a multitenant container database or pluggable database), then the database treats nonblocking DDL statements like blocking DDL statements. For information about enabling and disabling supplemental logging, see *Oracle Database SQL Language Reference*.

Caution: Do not issue a nonblocking DDL statement in an autonomous transaction. For information about autonomous transactions, see "[Autonomous Transactions](#)" on page 6-32.

6.8 Autonomous Transactions

Caution: Do not issue a nonblocking DDL statement in an autonomous transaction. For information about nonblocking DDL statements, see "[Nonblocking and Blocking DDL Statements](#)" on page 6-31.

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

For example, in a stock purchase transaction, you might want to commit customer information regardless of whether the purchase succeeds. Or, you might want to log error messages to a debug table even if the transaction rolls back. Autonomous transactions let you do such tasks.

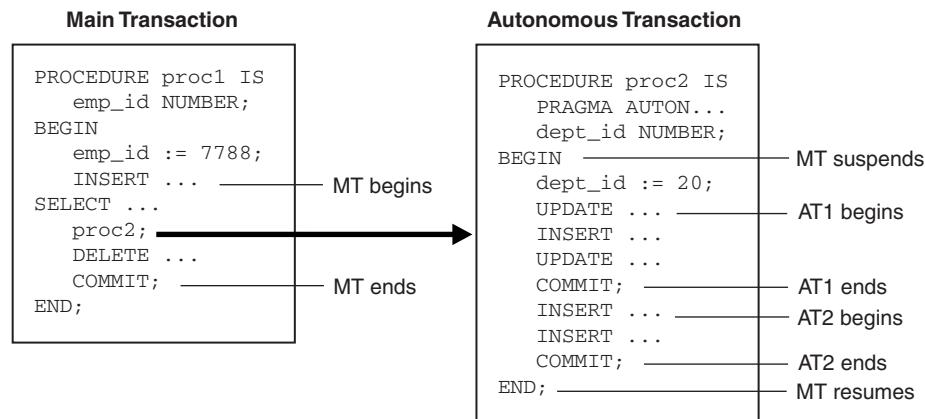
An autonomous transaction runs within an **autonomous scope**; that is, within the scope of an **autonomous routine**—a routine that you mark with the `AUTONOMOUS_TRANSACTION` pragma. In this context, a **routine** is one of these:

- Schema-level (not nested) anonymous PL/SQL block
- Standalone, package, or nested subprogram
- Method of an ADT
- Noncompound trigger

An autonomous routine can commit multiple autonomous transactions.

Figure 6–3 shows how control flows from the main transaction (proc1) to an autonomous routine (proc2) and back again. The autonomous routine commits two transactions (AT1 and AT2) before control returns to the main transaction.

Figure 6–3 Transaction Control Flow



When you enter the executable section of an autonomous transaction, the main transaction suspends. When you exit the transaction, the main transaction resumes. `COMMIT` and `ROLLBACK` end the active autonomous transaction but do not exit the autonomous transaction. As Figure 6–3 shows, when one transaction ends, the next SQL statement begins another transaction.

More characteristics of autonomous transactions:

- The changes an autonomous transaction effects do not depend on the state or the eventual disposition of the main transaction. For example:
 - An autonomous transaction does not see 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. Therefore, users can access the updated information without having to wait for the main transaction to commit.
- Autonomous transactions can start other autonomous transactions.

Figure 6–4 shows some possible sequences that autonomous transactions can follow.

Figure 6–4 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, then 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.

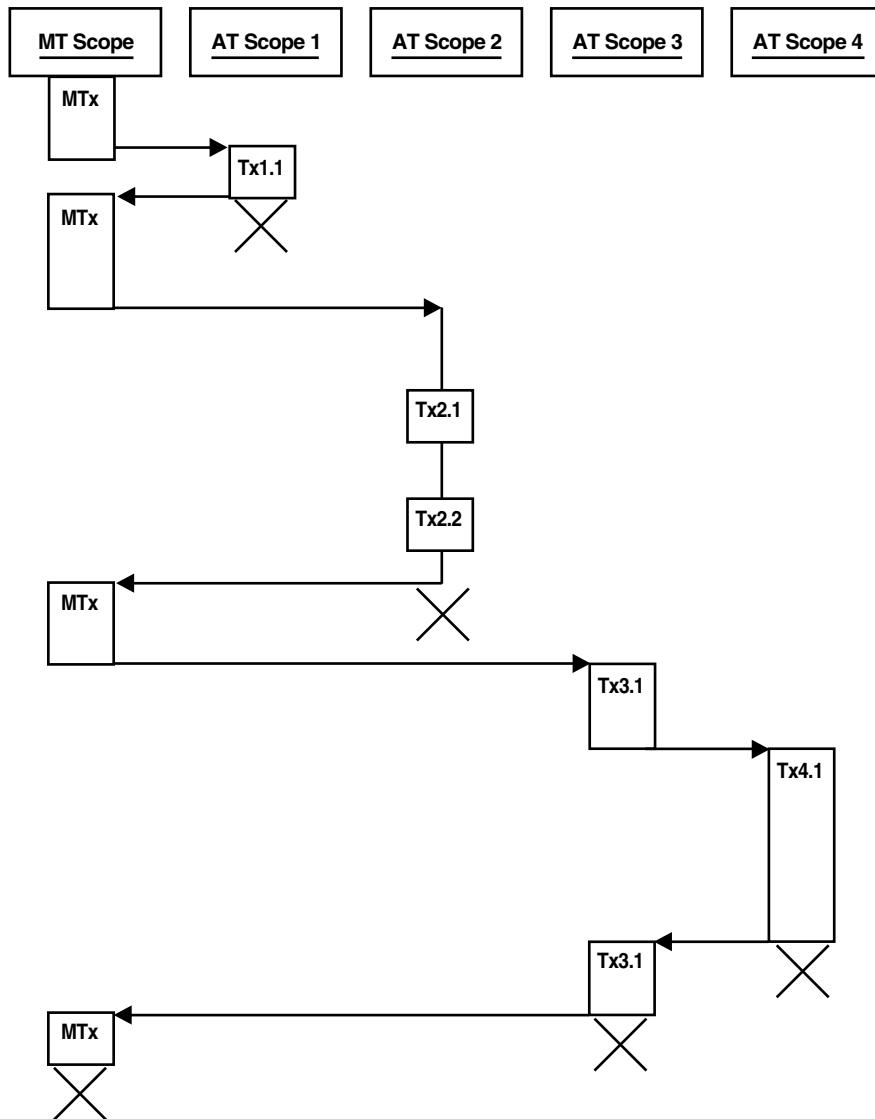
MTx invokes AT Scope 3. MTx suspends, AT Scope 3 begins.

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.



Topics:

- Examples of Autonomous Transactions
- Declaring Autonomous Routines

See Also: *Oracle Database PL/SQL Language Reference* for detailed information about autonomous transactions

6.8.1 Examples of Autonomous Transactions

This section shows examples of autonomous transactions.

Topics:

- Ordering a Product
- Withdrawing Money from a Bank Account

As these examples show, there are four possible outcomes when you use autonomous and main transactions (see [Table 6-7](#)). There is no dependency between the outcome of an autonomous transaction and that of a main transaction.

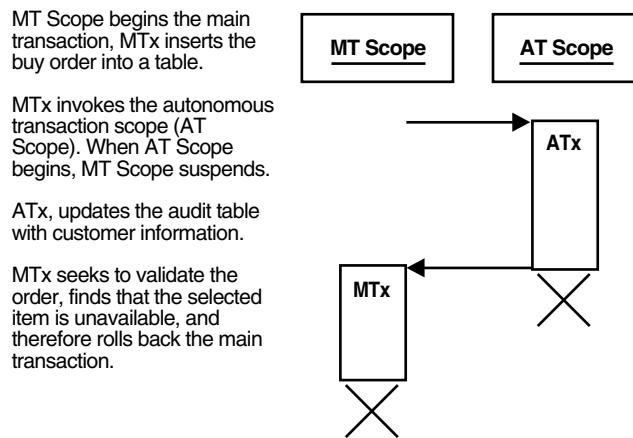
Table 6-7 Possible Transaction Outcomes

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

6.8.1.1 Ordering a Product

[Figure 6-5](#) shows an example of a customer ordering a product. The customer information (such as name, address, phone) is committed to a customer information table—even though the sale does not go through.

Figure 6-5 Example: A Buy Order



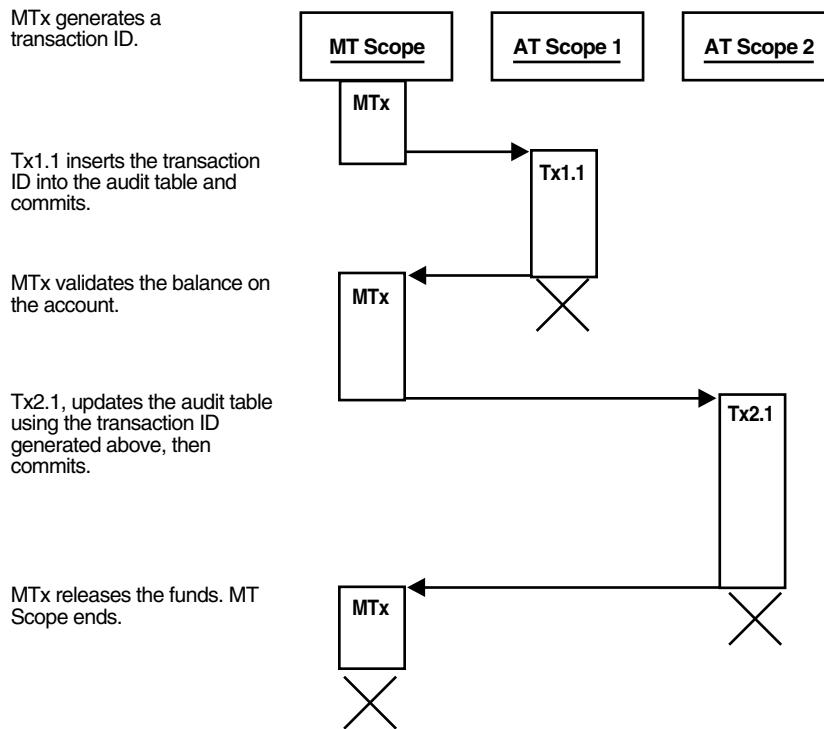
6.8.1.2 Withdrawing Money from a Bank Account

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).

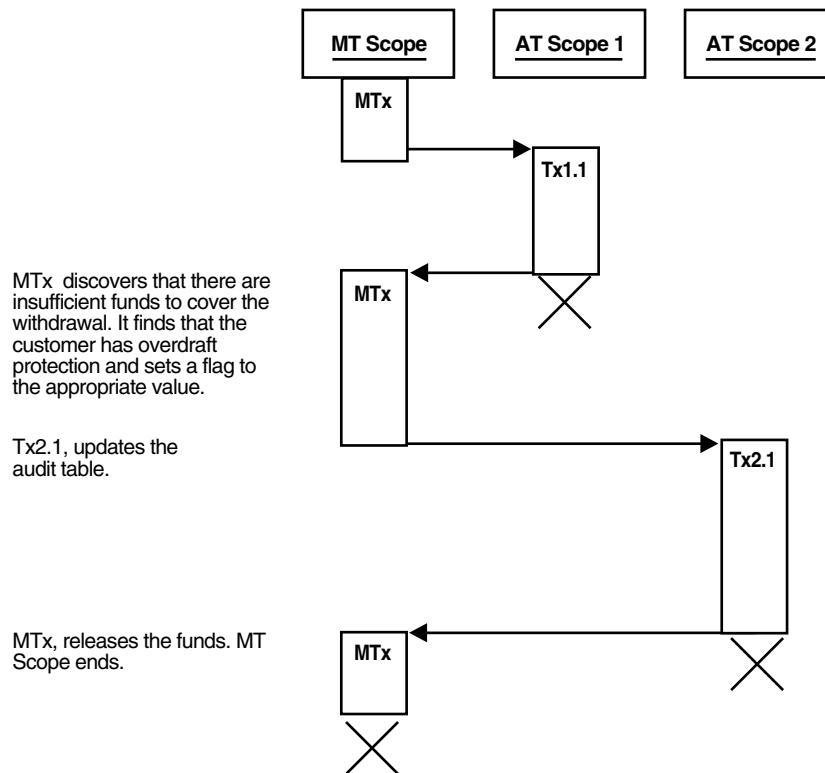
The possible scenarios for this transaction are:

- [Section 6.8.1.2.1, "Scenario 1: Sufficient Funds"](#)
- [Section 6.8.1.2.2, "Scenario 2: Insufficient Funds with Overdraft Protection"](#)
- [Section 6.8.1.2.3, "Scenario 3: Insufficient Funds Without Overdraft Protection"](#)

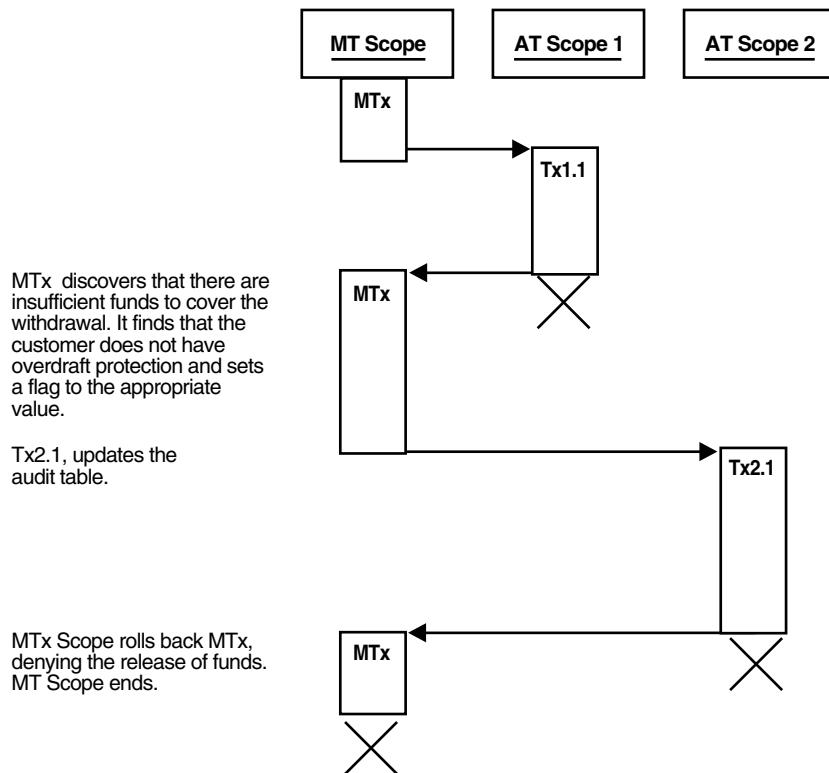
6.8.1.2.1 Scenario 1: Sufficient Funds There are sufficient funds to cover the withdrawal, so the bank releases the funds (see [Figure 6-6](#)).

Figure 6–6 Bank Withdrawal—Sufficient Funds

6.8.1.2.2 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 6–7](#)).

Figure 6-7 Bank Withdrawal—Insufficient Funds with Overdraft Protection

6.8.1.2.3 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 6-8](#)).

Figure 6–8 Bank Withdrawal—Insufficient Funds Without Overdraft Protection

6.8.2 Declaring Autonomous Routines

To declare an autonomous routine, use `PRAGMA AUTONOMOUS_TRANSACTION`, which instructs the PL/SQL compiler to mark the routine as autonomous.

In [Example 6–3](#), the function `balance` is autonomous.

Example 6–3 Marking a Package Subprogram as Autonomous

```
-- Create table for package to use:
DROP TABLE accounts;
CREATE TABLE accounts (account INTEGER, balance REAL);

-- Create package:
CREATE OR REPLACE PACKAGE banking AS
  FUNCTION balance (acct_id INTEGER) RETURN REAL;
  -- 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
      SELECT balance INTO my_bal FROM accounts WHERE account=acct_id;
      RETURN my_bal;
    END;
    -- Additional functions and packages
  END banking;
```

/

See Also: *Oracle Database PL/SQL Language Reference* for more information about PRAGMA AUTONOMOUS_TRANSACTION

6.9 Resuming Execution After Storage Allocation Errors

When a long-running transaction is interrupted by a storage allocation error, the application can suspend the statement that encountered the problem, correct the problem, and then resume executing the statement. This capability, called **resumable storage allocation**, avoids time-consuming rollbacks. It also makes it unnecessary to split the operation into smaller pieces and write code to track its progress.

See Also: *Oracle Database Administrator's Guide* for more information about resumable storage allocation

Topics:

- [What Operations Have Resumable Storage Allocation?](#)
- [Handling Suspended Storage Allocation](#)

6.9.1 What Operations Have Resumable Storage Allocation?

Queries, DML statements, and some DDL statements have resumable storage allocation after these kinds of errors:

- Out-of-space errors, such as ORA-01653.
- Space-limit errors, such as ORA-01628.
- Space-quota errors, such as ORA-01536.

Resumable storage allocation is possible whether the operation is performed directly by a SQL statement or within SQL*Loader, a stored subprogram, an anonymous PL/SQL block, or an OCI call such as OCISqlExecute.

In dictionary-managed tablespaces, you cannot resume an index- or table-creating operation that encounters the limit for rollback segments or the maximum number of extents. You must use locally managed tablespaces and automatic undo management in combination with resumable storage allocation.

6.9.2 Handling Suspended Storage Allocation

When a statement in an application is suspended because of a storage allocation error, the application does not receive an error code. Therefore, either the application must use an AFTER SUSPEND trigger or the DBA must periodically check for suspended statements.

After the problem is corrected (usually by the DBA), the suspended statement automatically resumes execution. If the timeout period expires before the problem is corrected, then the statement raises a SERVERERROR exception.

Topics:

- [Using an AFTER SUSPEND Trigger in the Application](#)
- [Checking for Suspended Statements](#)

6.9.2.1 Using an AFTER SUSPEND Trigger in the Application

In the application, an AFTER SUSPEND trigger can get information about the problem by invoking subprograms in the DBMS_RESUMABLE package (described in *Oracle Database PL/SQL Packages and Types Reference*). Then the trigger can send the information to an operator, using email (for example).

To reduce the chance of out-of-space errors within the trigger itself, declare the trigger as an autonomous transaction. As an autonomous transaction, the trigger uses a rollback segment in the SYSTEM tablespace. If the trigger encounters a deadlock condition because of locks held by the suspended statement, then the trigger terminates and the application receives the original error code, as if the statement were never suspended. If the trigger encounters an out-of-space condition, then both the trigger and the suspended statement are rolled back. To prevent rollback, use an exception handler in the trigger to wait for the statement to resume.

For general information about triggers, see *Oracle Database PL/SQL Language Reference*.

The trigger in [Example 6-4](#) handles storage errors within the database. For some kinds of errors, the trigger terminates the statement and alerts the DBA, using e-mail. For other errors, which might be temporary, the trigger specifies that the statement waits for eight hours before resuming, expecting the storage problem to be fixed by then. To run this example, you must connect to the database as SYSDBA.

Example 6-4 AFTER SUSPEND Trigger Handles Suspended Storage Allocation

```
-- Create table used by trigger body

DROP TABLE rbs_error;
CREATE TABLE rbs_error (
    SQL_TEXT VARCHAR2(64),
    ERROR_MSG VARCHAR2(64),
    SUSPEND_TIME VARCHAR2(64)
);

-- 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_RESUMABLESPACE_ERROR_INFO
        (err_type,
        object_owner,
        object_type,
        table_space_name,
```

```

        object_name,
        sub_object_name);
IF object_type = 'ROLLBACK SEGMENT' THEN
  INSERT INTO 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 :=
    'Space error occurred: Space limit reached for rollback segment '
    || object_name || ' on ' || to_char(SYSDATE, 'Month dd, YYYY, HH:MIam')
    || '. Error message was: ' || error_txt;

  mail_conn := UTL_SMTP.OPEN_CONNECTION('localhost', 25);
  UTL_SMTP.HELO(mail_conn, 'localhost');
  UTL_SMTP.MAIL(mail_conn, 'sender@localhost');
  UTL_SMTP.RCPT(mail_conn, 'recipient@localhost');
  UTL_SMTP.DATA(mail_conn, msg_body);
  UTL_SMTP.QUIT(mail_conn);
  DBMS_RESUMABLE.ABORT(cur_sid);
ELSE
  DBMS_RESUMABLE.SET_TIMEOUT(3600*8);
END IF;
COMMIT;
END;
/

```

6.9.2.2 Checking for Suspended Statements

If the application does not use an AFTER SUSPEND trigger, then the DBA must periodically check for suspended statements, using the static data dictionary view DBA_RESUMABLE (described in *Oracle Database Reference*).

The DBA can get additional information from the dynamic performance view V\$SESSION_WAIT (described in *Oracle Database Reference*).

Using SQL Data Types in Database Applications

This chapter explains how to choose the correct SQL data types for database columns that you create for your database applications. For syntactic and semantic information about SQL data types, see *Oracle Database SQL Language Reference*.

Topics:

- [Using the Correct and Most Specific Data Type](#)
- [Representing Character Data](#)
- [Representing Numeric Data](#)
- [Representing Date and Time Data](#)
- [Representing Specialized Data](#)
- [Identifying Rows by Address](#)
- [Displaying Metadata for SQL Operators and Functions](#)

See Also:

- *Oracle Database SQL Language Reference* for information about data type conversion
- [Section 11.4.4, "PL/SQL Data Types"](#)

Note: Oracle precompilers recognize, in embedded SQL programs, data types other than SQL and PL/SQL data types. These **external data types** are associated with host variables. For information about Oracle precompilers, see [Section 17.6](#).

7.1 Using the Correct and Most Specific Data Type

Using the correct and most specific data type for each database column that you create for your database application increases data integrity, decreases storage requirements, and improves performance.

Topics:

- [How the Correct Data Type Increases Data Integrity](#)
- [How the Most Specific Data Type Decreases Storage Requirements](#)
- [How the Correct Data Type Improves Performance](#)

7.1.1 How the Correct Data Type Increases Data Integrity

The correct data type increases data integrity by acting as a constraint. For example, if you use a datetime data type for a column of dates, then only dates can be stored in that column. However, if you use a character or numeric data type for the column, then eventually someone will store a character or numeric value that does not represent a date. You could write code to prevent this problem, but it is more efficient to use the correct data type. Therefore, store characters in character data types, numbers in numeric data types, and dates and times in datetime data types.

See Also: [Appendix 10, "Maintaining Data Integrity in Database Applications,"](#) for information about data integrity and constraints

7.1.2 How the Most Specific Data Type Decreases Storage Requirements

In addition to using the correct data type, use the most specific length or precision; for example:

- When creating a `VARCHAR2` column intended for strings of at most *n* characters, specify `VARCHAR2 (n)`.
- When creating a column intended for integers, use the data type `NUMBER (38)` rather than `NUMBER`.

Besides acting as constraints and thereby increasing data integrity, length and precision affect storage requirements.

If you give every column the maximum length or precision for its data type, then your application needlessly allocates many megabytes of RAM. For example, suppose that a query selects 10 `VARCHAR2 (4000)` columns and a bulk fetch operation returns 100 rows. The RAM that your application must allocate is $10 \times 4,000 \times 100$ —almost 4 MB. In contrast, if the column length is 80, the RAM that your application must allocate is $10 \times 80 \times 100$ —about 78 KB. This difference is significant for a single query, and your application will process many queries concurrently. Therefore, your application must allocate the 4 MB or 78 KB of RAM *for each connection*.

Therefore, do not give a column the maximum length or precision for its data type only because you might need to increase that property later. If you must change a column after creating it, then use the `ALTER TABLE` statement (described in *Oracle Database SQL Language Reference*). For example, to increase the length of a column, use:

```
ALTER TABLE table_name MODIFY column_name VARCHAR2 (larger_number)
```

Note: The maximum length of the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types is 32,767 bytes if the `MAX_STRING_SIZE` initialization parameter is `EXTENDED`. For more information about extended data types, see *Oracle Database SQL Language Reference*.

7.1.3 How the Correct Data Type Improves Performance

The correct data type improves performance because the incorrect data type can result in the incorrect execution plan. (For information about the execution plan, see [Section 2.2.3](#).)

[Example 7-1](#) performs the same conceptual operation—selecting rows whose dates are between December 31, 2000 and January 1, 2001—for three columns with different data types and shows the execution plan for each query. In the three execution plans, compare Rows (cardinality), Cost, and Operation.

Example 7-1 Performance Comparison of Three Data Types

Create a table that stores the same dates in three columns: str_date, with data type VARCHAR2; date_date, with data type DATE, and number_date, with data type NUMBER:

```
CREATE TABLE t (str_date, date_date, number_date, data)
AS
SELECT TO_CHAR(dt+rownum, 'yyyymmdd')          str_date,    -- VARCHAR2
       dt+rownum                                date_date,   -- DATE
       TO_NUMBER(TO_CHAR(dt+rownum, 'yyyymmdd'))  number_date, -- NUMBER
       RPAD('*', 45, '*')                         data
FROM (SELECT TO_DATE('01-jan-1995', 'dd-mm-yyyy') dt
      FROM all_objects)
ORDER BY DBMS_RANDOM.VALUE
/
```

Create an index on each column:

```
CREATE INDEX t_str_date_idx ON t(str_date);
CREATE INDEX t_date_date_idx ON t(date_date);
CREATE INDEX t_number_date_idx ON t(number_date);
```

Gather statistics for the table:

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (
    'HR',
    'T',
    method_opt => 'for all indexed columns size 254',
    cascade => TRUE
  );
END;
/
```

Show the execution plans of subsequent SQL statements (SQL*Plus command):

```
SET AUTOTRACE ON EXPLAIN
```

Select the rows for which the dates in str_date are between December 31, 2000 and January 1, 2001:

```
SELECT * FROM t WHERE str_date BETWEEN '20001231' AND '20010101'
ORDER BY str_date;
```

Result and execution plan:

```
STR_DATE DATE_DATE NUMBER_DATE DATA
-----
20001231 31-DEC-00    20001231 ****
20010101 01-JAN-01    20010101 ****
2 rows selected.
```

Execution Plan

```
-----
```

```
Plan hash value: 948745535
```

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		236	11092	216 (8)	00:00:01
1	SORT ORDER BY		236	11092	216 (8)	00:00:01

```
|* 2 | TABLE ACCESS FULL| T | 236 | 11092 | 215 (8) | 00:00:01 |
```

Predicate Information (identified by operation id):

```
2 - filter("STR_DATE" <='20010101' AND "STR_DATE" >='20001231')
```

Select the rows for which the dates in number_date are between December 31, 2000 and January 1, 2001:

```
SELECT * FROM t WHERE number_date BETWEEN 20001231 AND 20010101;
ORDER BY str_date;
```

Result and execution plan:

```
STR_DATE DATE_DATE NUMBER_DATE DATA
```

```
20001231 31-DEC-00 20001231 ****
20010101 01-JAN-01 20010101 ****
```

2 rows selected.

Execution Plan

Plan hash value: 948745535

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		234	10998	219 (10)	00:00:01
1	SORT ORDER BY		234	10998	219 (10)	00:00:01
* 2	TABLE ACCESS FULL T		234	10998	218 (9)	00:00:01

Predicate Information (identified by operation id):

```
2 - filter("NUMBER_DATE" <=20010101 AND "NUMBER_DATE" >=20001231)
```

Select the rows for which the dates in date_date are between December 31, 2000 and January 1, 2001:

```
SELECT * FROM t WHERE date_date
BETWEEN TO_DATE('20001231','yyyyymmdd')
AND TO_DATE('20010101','yyyyymmdd');
ORDER BY str_date;
```

Result and execution plan (reformatted to fit the page):

```
STR_DATE DATE_DATE NUMBER_DATE DATA
```

```
20001231 31-DEC-00 20001231 ****
20010101 01-JAN-01 20010101 ****
```

2 rows selected.

Execution Plan

Plan hash value: 2411593187

```

-----+
| Id  | Operation                                | Name      | Rows | Bytes |
| 1   | SORT ORDER BY                            |           | 1    | 47    |
| 2   | TABLE ACCESS BY INDEX ROWID BATCHED    | T         | 1    | 47    |
| * 3  |      INDEX RANGE SCAN                  | T_DATE_DATE_IDX | 1    | 47    |
| 0   | SELECT STATEMENT                         |           | 1    | 47    |
-----+
Cost (%CPU) | Time      |
4  (25) | 00:00:01 |
4  (25) | 00:00:01 |
3   (0) | 00:00:01 |
2   (0) | 00:00:01 |
-----+
Predicate Information (identified by operation id):
-----+
3 - access("DATE_DATE">>=TO_DATE(' 2000-12-31 00:00:00',
'syyyy-mm-dd hh24:mi:ss') AND
"DATE_DATE" <=TO_DATE(' 2001-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Performance improved for the final query because, for the DATE data type, the optimizer could determine that there was only one day between December 31, 2000 and January 1, 2001. Therefore, it performed an index range scan, which is faster than a full table scan. (For information about full table scans, see *Oracle Database SQL Tuning Guide*. For information about index range scans, see *Oracle Database SQL Tuning Guide*.)

7.2 Representing Character Data

Table 7–1 summarizes the SQL data types that store character data.

Table 7–1 SQL Character Data Types

Data Types	Values Stored
CHAR	Fixed-length character literals
VARCHAR2	Variable-length character literals
NCHAR	Fixed-length Unicode character literals
NVARCHAR2	Variable-length Unicode character literals
CLOB	Single-byte and multibyte character strings of up to (4 gigabytes - 1) * (the value obtained from DBMS_LOB.GETCHUNKSIZE)
NCLOB	Single-byte and multibyte Unicode character strings of up to (4 gigabytes - 1) * (the value obtained from DBMS_LOB.GETCHUNKSIZE)
LONG	Variable-length character data of up to 2 gigabytes - 1. Provided only for backward compatibility.

Note: Do not use the `VARCHAR` data type. Use the `VARCHAR2` data type instead. Although the `VARCHAR` data type is currently synonymous with `VARCHAR2`, the `VARCHAR` data type is scheduled to be redefined as a separate data type used for variable-length character strings compared with different comparison semantics.

When choosing between `CHAR` and `VARCHAR2`, consider:

- Space usage

Oracle Database blank-pads values stored in `CHAR` columns but not values stored in `VARCHAR2` columns. Therefore, `VARCHAR2` columns use space more efficiently than `CHAR` columns.

- Performance

Because of the blank-padding difference, a full table scan on a large table containing `VARCHAR2` columns might read fewer data blocks than a full table scan on a table containing the same data stored in `CHAR` columns. If your application often performs full table scans on large tables containing character data, then you might be able to improve performance by storing data in `VARCHAR2` columns rather than in `CHAR` columns.

- Comparison semantics

When you need ANSI compatibility in comparison semantics, use the `CHAR` data type. When trailing blanks are important in string comparisons, use the `VARCHAR2` data type.

See Also: *Oracle Database SQL Language Reference* for more information about comparison semantics for these data types

For a client/server application, if the character set on the client side differs from the character set on the server side, then Oracle Database converts `CHAR`, `VARCHAR2`, and `LONG` data from the database character set (determined by the `NLS_LANGUAGE` parameter) to the character set defined for the user session.

See Also:

- *Oracle Database SQL Language Reference* for more information about `CHAR`, `VARCHAR2`, `NCHAR`, and `NVARCHAR2` data types
- [Section 7.5.3.1, "Large Objects \(LOBs\),"](#) for more information about `CLOB` and `NCLOB` data types
- [Section 7.5.3.2, "LONG and LONG RAW Data Types,"](#) for more information about `LONG` data type

7.3 Representing Numeric Data

The SQL data types that store numeric data are `NUMBER`, `BINARY_FLOAT`, and `BINARY_DOUBLE`.

The `NUMBER` data type stores real numbers in either a fixed-point or floating-point format. `NUMBER` offers up to 38 decimal digits of precision. In a `NUMBER` column, you can store positive and negative numbers of magnitude 1×10^{-130} through 9.99×10^{125} , and 0. All Oracle Database platforms support `NUMBER` values.

The `BINARY_FLOAT` and `BINARY_DOUBLE` data types store floating-point numbers in the single-precision (32-bit) IEEE 754 format and the double-precision (64-bit) IEEE 754 format, respectively. High-precision values use less space when stored as `BINARY_FLOAT` and `BINARY_DOUBLE` than when stored as `NUMBER`. Arithmetic operations on floating-point numbers are usually faster for `BINARY_FLOAT` and `BINARY_DOUBLE` values than for `NUMBER` values.

In client interfaces that Oracle Database supports, arithmetic operations on `BINARY_FLOAT` and `BINARY_DOUBLE` values are performed by the native instruction set that the hardware vendor supplies. The term **native floating-point data type** includes `BINARY_FLOAT` and `BINARY_DOUBLE` data types and all implementations of these types in supported client interfaces.

Native floating-point data types conform substantially with the Institute of Electrical and Electronics Engineers (IEEE) Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-1985 (IEEE754). For details, see *Oracle Database SQL Language Reference*.

Note: Oracle recommends using `BINARY_FLOAT` and `BINARY_DOUBLE` instead of `FLOAT`, a subtype of `NUMBER` described in *Oracle Database SQL Language Reference*.

Topics:

- [Floating-Point Number Components](#)
- [Floating-Point Number Formats](#)
- [Representing Special Values with Native Floating-Point Data Types](#)
- [Comparing Native Floating-Point Values](#)
- [Arithmetic Operations with Native Floating-Point Data Types](#)
- [Conversion Functions for Native Floating-Point Data Types](#)
- [Client Interfaces for Native Floating-Point Data Types](#)

See Also:

- *Oracle Database SQL Language Reference* for more information about the `NUMBER` data type
- *Oracle Database SQL Language Reference* for more information about the `BINARY_FLOAT` and `BINARY_DOUBLE` data types

7.3.1 Floating-Point Number Components

The formula for a floating-point value is:

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

For example, the floating-point value 4.31 is represented:

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

The components of the preceding representation are:

Component Name	Component Value
Sign	0
Significand	431

Component Name	Component Value
Base	10
Exponent	-2

7.3.2 Floating-Point Number Formats

A floating-point number format specifies how the components of a floating-point number are represented, thereby determining the range and precision of the values that the format can represent. The **range** is the interval bounded by the smallest and largest values and the **precision** is the number of significant digits. Both range and precision are finite. If a floating-point number is too precise for a given format, then the number is rounded.

How the number is rounded depends on the base of its format, which can be either decimal or binary. A number stored in decimal format is rounded to the nearest decimal place (for example, 1000, 10, or 0.01). A number stored in binary format is rounded to the nearest binary place (for example, 1024, 512, or 1/64).

NUMBER values are stored in decimal format. For calculations that need decimal rounding, use the NUMBER data type.

Native floating-point values are stored in binary format.

Table 7–2 shows the range and precision of the IEEE 754 single- and double-precision formats and Oracle Database NUMBER. Range limits are expressed as positive numbers, but they also apply to absolute values of negative numbers. (The notation "number e exponent" means $number * 10^{\text{exponent}}$.)

Table 7–2 Range and Precision of Floating-Point Data Types

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

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

7.3.2.1 Binary Floating-Point Formats

This formula determines the value of a floating-point number that uses a binary format:

$$(-1)^{\text{sign}} 2^{\text{E}} (\text{bit}_0 \text{ bit}_1 \text{ bit}_2 \dots \text{ bit}_{p-1})$$

Table 7–3 describes the components of the preceding formula.

Table 7-3 Binary Floating-Point Format Components

Component	Component Value
sign	0 or 1
E (exponent)	For single-precision (32-bit) data type, an integer from -126 through 127. For double-precision (64-bit) data type, an integer from -1022 through 1023.
bit _i	0 or 1. (The bit sequence represents a number in base 2.)
p (precision)	For single-precision data type, 24. For double-precision data type, 53.

The leading bit of the significand, b_0 , must be set (1), except for subnormal numbers (explained later). Therefore, the leading bit is not stored, and a binary format provides n bits of precision while storing only $n-1$ bits. The IEEE 754 standard defines the in-memory formats for single-precision and double-precision data types, as [Table 7-4](#) shows.

Table 7-4 Summary of Binary Format Storage Parameters

Data Type	Sign Bit	Exponent Bits	Significand Bits	Total Bits
Single-precision	1	8	24 (23 stored)	32
Double-precision	1	11	53 (52 stored)	64

Note: Oracle Database does not support the extended single- and double-precision formats that the IEEE 754 standard defines.

A significand whose leading bit is set is called **normalized**. The IEEE 754 standard defines **subnormal numbers** (also called **denormal numbers**) that are too small to represent with normalized significands. If the significand of a subnormal number were normalized, then its exponent would be too large. Subnormal numbers preserve this property: If $x-y=0.0$ (using floating-point subtraction), then $x=y$.

7.3.3 Representing Special Values with Native Floating-Point Data Types

The IEEE 754 standard supports the special values shown in [Table 7-5](#).

Table 7-5 Special Values for Native Floating-Point Formats

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

Each value in [Table 7-5](#) is represented by a specific bit pattern, except NaN. NaN, the result of any undefined operation, is represented by many bit patterns. Some of these bits patterns have the sign bit set and some do not, but the sign bit has no meaning.

The IEEE 754 standard distinguishes between quiet NaNs (which do not raise additional exceptions as they propagate through most operations) and signaling NaNs (which do). The IEEE 754 standard specifies action for when exceptions are enabled and action for when they are disabled.

In Oracle Database, exceptions cannot be enabled. Oracle Database acts as the IEEE 754 standard specifies for when exceptions are disabled. In particular, Oracle Database does not distinguish between quiet and signaling NaNs. You can use Oracle Call Interface (OCI) to retrieve NaN values from Oracle Database, but whether a retrieved NaN value is signaling or quiet depends on the client platform and is beyond the control of Oracle Database.

The IEEE 754 standard defines these classes of special values:

- Zero
- Subnormal
- Normal
- Infinity
- NaN

The values in each class in the preceding list are larger than the values in the classes that precede it in the list (ignoring signs), except NaN. NaN is unordered with other classes of special values and with itself.

In Oracle Database:

- All NaNs are quiet.
- Any non-NaN value < NaN
- Any NaN == any other NaN
- All NaNs are converted to the same bit pattern.
- -0 is converted to +0.
- IEEE 754 exceptions are not raised.

See Also: *Oracle Database SQL Language Reference* for information about floating-point conditions, which let you determine whether an expression is infinite or is the undefined result of an operation (is not a number or NaN).

7.3.4 Comparing Native Floating-Point Values

When comparing numeric expressions, Oracle Database uses numeric precedence to determine whether the condition compares NUMBER, BINARY_FLOAT, or BINARY_DOUBLE values. For information about numeric precedence, see *Oracle Database SQL Language Reference*. For general information about comparing numeric expressions, see *Oracle Database SQL Language Reference*.

Comparisons ignore the sign of zero (-0 equals +0).

7.3.5 Arithmetic Operations with Native Floating-Point Data Types

IEEE 754 does not require floating-point arithmetic to be exactly reproducible. Therefore, results of operations can be delivered to a destination that uses a range greater than the range that the operands of the operation use.

You can compute the result of a double-precision multiplication at an extended double-precision destination, but 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; however, this might cause 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. Therefore, except for this case, arithmetic for these data types 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.

See Also: *Oracle Database SQL Language Reference* for general information about arithmetic operations

7.3.6 Conversion Functions for Native Floating-Point Data Types

Oracle Database defines functions that convert between floating-point and other data types, including string formats that use decimal precision (but precision might be lost during the conversion). For example:

- `TO_BINARY_DOUBLE`, described in *Oracle Database SQL Language Reference*
- `TO_BINARY_FLOAT`, described in *Oracle Database SQL Language Reference*
- `TO_CHAR`, described in *Oracle Database SQL Language Reference*
- `TO_NUMBER`, described in *Oracle Database SQL Language Reference*

Oracle Database can raise exceptions during conversion. The IEEE 754 standard defines these exceptions:

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

However, Oracle Database does not raise these exceptions for native floating-point data types. Generally, operations that raise exceptions produce the values described in [Table 7–6](#).

Table 7–6 Values Resulting from Exceptions

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

7.3.7 Client Interfaces for Native Floating-Point Data Types

Oracle Database supports native floating-point data types in these client interfaces:

- SQL and PL/SQL

Support for BINARY_FLOAT and BINARY_DOUBLE includes their use as attributes of Abstract Data Types (ADTs), which you create with the SQL statement CREATE TYPE (fully described in *Oracle Database PL/SQL Language Reference*).

- Oracle Call Interface (OCI)
For information about using BINARY_FLOAT and BINARY_DOUBLE with OCI, see *Oracle Call Interface Programmer's Guide*.
- Oracle C++ Call Interface (OCCI)
For information about using BINARY_FLOAT with OCCI, see *Oracle C++ Call Interface Programmer's Guide*.
For information about using BINARY_DOUBLE with OCCI, see *Oracle C++ Call Interface Programmer's Guide*.
- Pro*C/C++ precompiler
To use BINARY_FLOAT and BINARY_DOUBLE, set the Pro*C/C++ precompiler command line option NATIVE_TYPES to YES when you compile your application. For information about the NATIVE_TYPES option, see *Pro*C/C++ Programmer's Guide*.
- Oracle JDBC
For information about using BINARY_FLOAT and BINARY_DOUBLE with Oracle JDBC, see *Oracle Database JDBC Developer's Guide*.

7.4 Representing Date and Time Data

Oracle Database stores DATE and TIMESTAMP (**datetime**) data in a binary format that represents the century, year, month, day, hour, minute, second, and optionally, fractional seconds and timezones.

[Table 7-7](#) summarizes the SQL datetime data types. For more information about these data types, see *Oracle Database SQL Language Reference*.

Table 7-7 SQL Datetime Data Types

Date Type	Usage
DATE	For storing datetime values in a table—for example, dates of jobs.
TIMESTAMP	For storing datetime values that are precise to fractional seconds—for example, times of events that must be compared to determine the order in which they occurred.
TIMESTAMP WITH TIME ZONE	For storing datetime values that must be gathered or coordinated across geographic regions.
TIMESTAMP WITH LOCAL TIME ZONE	For storing datetime values when the time zone is insignificant—for example, in an application that schedules teleconferences, where participants see the start and end times for their own time zone. Appropriate for two-tier applications in which you want to display dates and times that use the time zone of the client system. Usually inappropriate for 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.

Table 7-7 (Cont.) SQL Datetime Data Types

Date Type	Usage
INTERVAL YEAR TO MONTH	For storing the difference between two datetime values, where only the year and month are significant—for example, to set a reminder for a date 18 months in the future, or check whether 6 months have elapsed since a particular date.
INTERVAL DAY TO SECOND	For storing the precise difference between two datetime values—for example, 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, use a large number of days.

Topics:

- [Displaying Current Date and Time](#)
- [Inserting and Displaying Dates](#)
- [Inserting and Displaying Times](#)
- [Arithmetic Operations with Datetime Data Types](#)
- [Conversion Functions for Datetime Data Types](#)
- [Importing, Exporting, and Comparing Datetime Types](#)

See Also: *Oracle Call Interface Programmer's Guide* for more information about Oracle Database internal date types

7.4.1 Displaying Current Date and Time

The simplest way to display the current date and time is:

```
SELECT TO_CHAR(SYSDATE, format_model) FROM DUAL
```

The default format model depends on the initialization parameter `NLS_DATE_FORMAT`.

The standard Oracle Database default date format is `DD-MON-RR`. The `RR` datetime format element lets you store 20th century dates in the 21st century by specifying only the last two digits of the year. For example, in the datetime format `DD-MON-YY`, `13-NOV-54` 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 2149.

Note: For program correctness and to avoid problems with SQL injection and dynamic SQL, Oracle recommends specifying a format model for every datetime value.

The simplest way to display the current date and time using a format model is:

```
SELECT TO_CHAR(SYSDATE, format_model) FROM DUAL
```

Example 7-2 uses `TO_CHAR` with a format model to display `SYSDATE` in a format with the qualifier `BC` or `AD`. (By default, `SYSDATE` is displayed without this qualifier.)

Example 7-2 Displaying Current Date and Time

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

Result:

```
NOW
-----
18-MAR-2009 AD

1 row selected.
```

Tip: When testing code that uses SYSDATE, it can be helpful to set SYSDATE to a constant. Do this with the initialization parameter `FIXED_DATE`, described in *Oracle Database Reference*.

See Also:

- *Oracle Database SQL Language Reference* for more information about SYSDATE
- *Oracle Database Globalization Support Guide* for information about `NLS_DATE_FORMAT`
- *Oracle Database SQL Language Reference* for more information about `TO_CHAR`
- *Oracle Database SQL Language Reference* for information about datetime format models
- *Oracle Database SQL Language Reference* for more information about the `RR` datetime format element

7.4.2 Inserting and Displaying Dates

When you display and insert dates, Oracle recommends using the `TO_CHAR` and `TO_DATE` functions, respectively, with datetime format models.

Example 7–3 creates a table with a `DATE` column and inserts a date into it, specifying a format model. Then the example displays the date with and without specifying a format model.

Example 7–3 Inserting and Displaying Dates

Create table:

```
DROP TABLE dates;
CREATE TABLE dates (d DATE);
```

Insert date specified into table, specifying a format model:

```
INSERT INTO dates VALUES (TO_DATE('OCT 27, 1998', 'MON DD, YYYY'));
```

Display date without specifying a format model:

```
SELECT d FROM dates;
```

Result:

```
D
-----
27-OCT-98
```

1 row selected.

Display date, specifying a format model:

```
SELECT TO_CHAR(d, 'YYYY-MON-DD') D FROM dates;
```

Result:

```
D
-----
1998-OCT-27
1 row selected.
```

Caution: Be careful when using the YY datetime format element, which indicates the year in the current century. For example, in the 21st century, the format DD-MON-YY, 31-DEC-92 is December 31, 2092 (not December 31, 1992, as you might expect). To store 20th century dates in the 21st century by specifying only the last two digits of the year, use the RR datetime format element (the default).

See Also:

- *Oracle Database Globalization Support Guide* for information about NLS_DATE_FORMAT
- *Oracle Database SQL Language Reference* for more information about TO_CHAR
- *Oracle Database SQL Language Reference* for more information about TO_DATE
- *Oracle Database SQL Language Reference* for information about datetime format models
- *Oracle Database SQL Language Reference* for more information about the RR datetime format element

7.4.3 Inserting and Displaying Times

When you display and insert times, Oracle recommends using the TO_CHAR and TO_DATE functions, respectively, with datetime format models.

In a DATE column:

- The default time is 12:00:00 A.M. (midnight).
The default time applies to any value in the column that has no time portion, either because none was specified or because the value was truncated.
- The default day is the first day of the current month.
The default date applies to any value in the column that has no date portion, because none was specified.

[Example 7-4](#) creates a table with a DATE column and inserts three dates into it, specifying a different format model for each date. The first format model has both date and time portions, the second has no time portion, and the third has no date portion. Then the example displays the three dates, specifying a format model that includes both date and time portions.

Example 7-4 Inserting and Displaying Dates and Times

Create table:

```
DROP TABLE birthdays;
CREATE TABLE birthdays (name VARCHAR2(20), day DATE);
```

Insert three dates, specifying a different format model for each date:

```
INSERT INTO birthdays (name, day)
VALUES ('Annie',
        TO_DATE('13-NOV-92 10:56 A.M.', 'DD-MON-RR HH:MI A.M.')
        );

INSERT INTO birthdays (name, day)
VALUES ('Bobby',
        TO_DATE('5-APR-02', 'DD-MON-RR')
        );

INSERT INTO birthdays (name, day)
VALUES ('Cindy',
        TO_DATE('8:25 P.M.', 'HH:MI A.M.')
        );
```

Display both date and time portions of stored datetime values:

```
SELECT name,
       TO_CHAR(day, 'Mon DD, RRRR') DAY,
       TO_CHAR(day, 'HH:MI A.M.') TIME
  FROM birthdays;
```

Result:

NAME	DAY	TIME
Annie	Nov 13, 1992	10:56 A.M.
Bobby	Apr 05, 2002	12:00 A.M.
Cindy	Nov 01, 2010	08:25 P.M.

3 rows selected.

7.4.4 Arithmetic Operations with Datetime Data Types

The results of arithmetic operations on datetime values are determined by the rules in *Oracle Database SQL Language Reference*.

SQL has many datetime functions that you can use in datetime expressions. For example, the function `ADD_MONTHS` returns the date that is a specified number of months from a specified date. For the complete list of datetime functions, see *Oracle Database SQL Language Reference*.

7.4.5 Conversion Functions for Datetime Data Types

[Table 7-8](#) summarizes the SQL functions that convert to or from datetime data types. For more information about these functions, see *Oracle Database SQL Language Reference*.

Table 7-8 SQL Conversion Functions for Datetime Data Types

Function	Converts ...	To ...
NUMTODSINTERVAL	NUMBER	INTERVAL DAY TO SECOND
NUMTOYMINTERVAL	NUMBER	INTERVAL DAY TO MONTH

Table 7–8 (Cont.) SQL Conversion Functions for Datetime Data Types

Function	Converts ...	To ...
TO_CHAR	DATE TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE INTERVAL DAY TO SECOND INTERVAL YEAR TO MONTH	VARCHAR2
TO_DATE	CHAR VARCHAR2 NCHAR NVARCHAR2	DATE
TO_DSINTERVAL	CHAR VARCHAR2 NCHAR NVARCHAR2	INTERVAL DAY TO SECOND
TO_TIMESTAMP	CHAR VARCHAR2 NCHAR NVARCHAR2	TIMESTAMP
TO_TIMESTAMP_TZ	CHAR VARCHAR2 NCHAR NVARCHAR2	TIMESTAMP WITH TIME ZONE
TO_YMINTERVAL	CHAR VARCHAR2 NCHAR NVARCHAR2	INTERVAL DAY TO MONTH

7.4.6 Importing, Exporting, and Comparing Datetime Types

You can import, export, and compare TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE values without worrying about time zone offsets, because the database stores these values in normalized format.

When importing, exporting, and comparing DATE and TIMESTAMP values, you must adjust them to account for any time zone differences between source and target databases, because the database does not store their time zones.

7.5 Representing Specialized Data

Topics:

- [Representing Spatial Data](#)
- [Representing Multimedia Data](#)
- [Representing Large Amounts of Data](#)

- [Representing Searchable Text](#)
- [Representing XML Data](#)
- [Representing Dynamically Typed Data](#)
- [Representing ANSI, DB2, and SQL/DS Data](#)

7.5.1 Representing Spatial Data

Spatial data is used by location-enabled applications, geographic information system (GIS) applications, and geoimaging applications. For information about representing spatial data in Oracle Database, see *Oracle Database SQL Language Reference*.

7.5.2 Representing Multimedia Data

Oracle Multimedia lets Oracle Database store, manage, and retrieve images, audio, video, or other heterogeneous media data. For information about representing multimedia data in Oracle Database, see *Oracle Database SQL Language Reference*.

7.5.3 Representing Large Amounts of Data

For representing large amounts of data, Oracle Database provides:

- [Large Objects \(LOBs\)](#)
- [LONG and LONG RAW Data Types](#) (for backward compatibility)

7.5.3.1 Large Objects (LOBs)

Large Objects (LOBs) are data types that are designed to store large amounts of data in a way that lets your application access and manipulate it efficiently.

Table 7-9 summarizes the LOBs. For more information about these functions, see *Oracle Database SQL Language Reference*.

Table 7-9 Large Objects (LOBs)

Data Type	Description
BLOB	<p>Binary large object</p> <p>Stores any kind of data in binary format.</p> <p>Typically used for multimedia data such as images, audio, and video.</p>
CLOB	<p>Character large object</p> <p>Stores string data in the database character set format.</p> <p>Used for large strings or documents that use the database character set exclusively.</p>
NCLOB	<p>National character large object</p> <p>Stores string data in National Character Set format.</p> <p>Used for large strings or documents in the National Character Set.</p>
BFILE	<p>External large object</p> <p>Stores a binary file outside the database in the host operating system file system.</p> <p>Applications have read-only access to BFILES.</p> <p>Used for static data that applications do not manipulate, such as image data.</p> <p>Any kind of data (that is, any operating system file) can be stored in a BFILE. For example, you can store character data in a BFILE and then load the BFILE data into a CLOB, specifying the character set when loading.</p>

An instance of type BLOB, CLOB, or NCLOB can be either **temporary** (declared in the scope of your application) or **persistent** (created and stored in the database).

See Also: *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about using LOBs in application development

7.5.3.2 LONG and LONG RAW Data Types

Note: Oracle supports the LONG and LONG RAW data types for backward compatibility, but strongly recommends that you convert LONG columns to LOB columns and LONG RAW columns to BLOB columns.

LONG columns store variable-length character strings containing up to 2 gigabytes - 1 bytes. For more information about the LONG data type, including its many restrictions, see *Oracle Database SQL Language Reference*.

The LONG RAW (and RAW) data types store data that is not to be explicitly converted by Oracle Database when moving data between different systems. These data types are intended for binary data or byte strings. For more information about the RAW and LONG RAW data types, see *Oracle Database SQL Language Reference*.

7.5.4 Representing Searchable Text

Rather than writing low-level code to do full-text searches, you can use Oracle Text. Oracle Text provides indexing, word and theme searching, and viewing capabilities for text in query applications and document classification applications. You can also use Oracle Text to search XML data.

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

7.5.5 Representing XML Data

If you have information stored as files in XML format, or want to store an ADT in XML format, then you can use the Oracle-supplied type XMLType, described in *Oracle Database SQL Language Reference*.

With XMLType values, you can use:

- XMLType member functions (see *Oracle XML DB Developer's Guide*).
- SQL XML functions (see *Oracle Database SQL Language Reference*)
- PL/SQL DBMS_XML packages (see *Oracle Database PL/SQL Packages and Types Reference*)

See Also:

- *Oracle XML DB Developer's Guide* for information about Oracle XML DB and how you can use it to store, generate, manipulate, manage, and query XML data in the database
- *Oracle XML Developer's Kit Programmer's Guide* for information about client-side programming with XML

7.5.6 Representing Dynamically Typed Data

Some languages allow data types to change at runtime, and some let a program check the type of a variable. For example, C has the `union` keyword and the `void *` pointer, and Java has the `typeof` operator and wrapper types such as `Number`.

In Oracle Database, you can create variables and columns that can hold data of any type and test their values to determine their underlying representation. For example, a single table column can have a numeric value in one row, a string value in another row, and an object in another row.

You can use the Oracle-supplied ADT `SYS.ANYDATA` to represent values of any scalar type or ADT. `SYS.ANYDATA` has methods that accept scalar values of any type, and turn them back into scalars or objects. Similarly, you can use the Oracle-supplied ADT `SYS.ANYDATASET` to represent values of any collection type. For more information about these ADTs, see *Oracle Database Object-Relational Developer's Guide*.

To check and manipulate type information, use the `DBMS_TYPES` package (described in *Oracle Database PL/SQL Packages and Types Reference*), as in [Example 7-5](#).

With OCI, use the `OCIAnyData` and `OCIAnyDataSet` interfaces, described in *Oracle Call Interface Programmer's Guide*.

Example 7-5 Accessing Information in a `SYS.ANYDATA` Column

```

CREATE OR REPLACE TYPE employee_type AS
  OBJECT (empno NUMBER, ename VARCHAR2(10));
/

DROP TABLE mytab;
CREATE TABLE mytab (id NUMBER, data SYS.ANYDATA);

INSERT INTO mytab (id, data)
VALUES (1, SYS.ANYDATA.ConvertNumber(5));

INSERT INTO mytab (id, data)
VALUES (2, SYS.ANYDATA.ConvertObject(Employee_type(5555, 'john')));

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
  FOR x IN cur LOOP
    FETCH cur INTO v_id, v_data;
    EXIT WHEN cur%NOTFOUND;

    /* typecode signifies type 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 */);
  
```

```

/* Compare typecode to DBMS_TYPES constants to determine type of data
   and decide how to display it. */

CASE v_typecode
  WHEN DBMS_TYPES.TYPENAME THEN
    IF v_type IS NOT NULL THEN -- This condition should never happen.
      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.TYPENAME_OBJECT THEN
    v_typename := v_data.GetTypeName();
    IF v_typename NOT IN ('HR.EMPLOYEE_TYPE') 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;
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;
/

```

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

Result:

```

Type Name
-----
SYS.NUMBER
HR.EMPLOYEE_TYPE

2 rows selected.

```

7.5.7 Representing ANSI, DB2, and SQL/DS Data

SQL statements that create tables and clusters can use ANSI data types and data types from the IBM products SQL/DS and DB2 (except those noted after this paragraph). Oracle Database converts the ANSI or IBM data type to the equivalent Oracle data type, records the Oracle data type as the name of the column data type, and stores the column data in the Oracle data type. For conversion details, see *Oracle Database SQL Language Reference*.

Note: SQL statements cannot use the SQL/DS and DB2 data types TIME, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC, because they have no equivalent Oracle data types.

7.6 Identifying Rows by Address

The fastest way to access the row of a database table is by its address, or **rowid**, which uniquely identifies it. Different rows in the same data block can have the same rowid only if they are in different clustered tables. If a row is larger than one data block, then its rowid identifies its initial row piece.

To see rowids, query the **ROWID** pseudocolumn. Each value in the **ROWID** pseudocolumn is a string that represents the address of a row. The data type of the string is either **ROWID** or **UROWID**.

See Also:

- *Oracle Database Concepts* for an overview of the **ROWID** pseudocolumn
- *Oracle Database Concepts* for an overview of rowid data types
- *Oracle Database SQL Language Reference* for more information about the **ROWID** pseudocolumn
- *Oracle Database SQL Language Reference* for more information about the **ROWID** data type
- *Oracle Database SQL Language Reference* for more information about the **UROWID** data type
- *Oracle Call Interface Programmer's Guide* for information about using the **ROWID** data type in C
- *Pro*C/C++ Programmer's Guide* for information about using the **ROWID** data type with the Pro*C/C++ precompiler
- *Oracle Database JDBC Developer's Guide* for information about using the **ROWID** data type in Java

Note: When you update a row in a table compressed with Hybrid Columnar Compression (HCC), the **ROWID** of the row changes. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

Example 7–6 creates a table with a column of the data type **ROWID**, populates it with rowids by querying the **ROWID** pseudocolumn inside an **INSERT** statement, and then displays it. The rowids of the table rows show how they are stored.

Note: Merely creating a column of the type **ROWID** (like **col1** in **Example 7–6**) does not guarantee that its values will be valid rowids.

Example 7–6 Querying the **ROWID** Pseudocolumn

```
DROP TABLE t_tab;  -- in case it exists
CREATE TABLE t_tab (col1 ROWID);
```

```

INSERT INTO t_tab (col1)
SELECT ROWID
FROM employees
WHERE employee_id > 199;

```

Query:

```

SELECT employee_id, rowid
FROM employees
WHERE employee_id > 199;

```

ROWID varies, but result is similar to:

```

EMPLOYEE_ID ROWID
-----
200 AAAAPeSAAFAAAABTAAC
201 AAAAPeSAAFAAAABTAAD
202 AAAAPeSAAFAAAABTAAE
203 AAAAPeSAAFAAAABTAAF
204 AAAAPeSAAFAAAABTAAG
205 AAAAPeSAAFAAAABTAAH
206 AAAAPeSAAFAAAABTAAI

```

7 rows selected.

Query:

```

SELECT * FROM t_tab;

```

COL1 varies, but result is similar to:

```

COL1
-----
AAAPeSAAFAAAABTAAC
AAAPeSAAFAAAABTAAD
AAAPeSAAFAAAABTAAE
AAAPeSAAFAAAABTAAF
AAAPeSAAFAAAABTAAG
AAAPeSAAFAAAABTAAH
AAAPeSAAFAAAABTAAI

```

7 rows selected.

7.7 Displaying Metadata for SQL Operators and Functions

The dynamic performance view V\$SQLFN_METADATA displays metadata about SQL operators and functions. For every function that V\$SQLFN_METADATA displays, the dynamic performance view V\$SQLFN_ARG_METADATA has one row of metadata about each function argument. If a function argument can be repeated (as in the functions LEAST and GREATEST), then V\$SQLFN_ARG_METADATA has only one row for each repeating argument. You can join the views V\$SQLFN_METADATA and V\$SQLFN_ARG_METADATA on the column FUNC_ID.

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

Topics:

- [ARGn Data Type](#)

- [DISP_TYPE Data Type](#)
- [SQL Data Type Families](#)

See Also:

- *Oracle Database Reference* for more information about V\$SQLFN_METADATA
- *Oracle Database Reference* for more information about V\$SQLFN_ARG_METADATA

7.7.1 ARGn Data Type

In the view V\$SQLFN_METADATA, the column DATATYPE is the data type of the function (that is, the data type that the function returns). This data type can be an Oracle data type, data type family (see [Section 7.7.3](#)), or ARGn. ARGn is the data type of the *n*th argument of the function. For example:

- The MAX function (described in *Oracle Database SQL Language Reference*) returns a value that has the data type of its first argument, so the MAX function has return data type ARG1.
- The DECODE function (described in *Oracle Database SQL Language Reference*) returns a value that has the data type of its third argument, so the DECODE function has data type ARG3.

7.7.2 DISP_TYPE Data Type

In the view V\$SQLFN_METADATA, the column DISP_TYPE is the data type 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 7-10 Display Types of SQL Functions

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

7.7.3 SQL Data Type Families

Often, a SQL function argument can have any data type in a data type family. [Table 7-11](#) shows the SQL data type families and their member data types.

Table 7-11 SQL Data Type Families

Family	Data Types
STRING	CHARACTER
	VARCHAR2
	CLOB
	NCHAR

Table 7-11 (Cont.) SQL Data Type Families

Family	Data Types
	NVARCHAR2
	NCLOB
	LONG
NUMERIC	NUMBER
	BINARY_FLOAT
	BINARY_DOUBLE
DATETIME	DATE
	TIMESTAMP
	TIMESTAMP WITH TIME ZONE
	TIMESTAMP WITH LOCAL TIME ZONE
	INTERVAL YEAR TO MONTH
	INTERVAL DAY TO SECOND
BINARY	BLOB
	RAW
	LONGRAW

Using Regular Expressions in Database Applications

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

Topics:

- [Overview of Regular Expressions](#)
- [Oracle SQL Support for Regular Expressions](#)
- [Oracle SQL and POSIX Regular Expression Standard](#)
- [Operators in Oracle SQL Regular Expressions](#)
- [Using Regular Expressions in SQL Statements: Scenarios](#)

See Also:

- *Oracle Database Globalization Support Guide* for information about 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

8.1 Overview of Regular Expressions

A regular expression specifies a search pattern, using **metacharacters** (which are, or belong to, **operators**) and **character literals** (described in *Oracle Database SQL Language Reference*).

The search pattern can be complex. For example, this regular expression matches any string that begins with either f or ht, followed by tp, optionally followed by s, followed by the colon (:):

(f|ht)tps?:

The metacharacters (which are also operators) in the preceding example are the parentheses, the pipe symbol (|), and the question mark (?). The character literals are f, ht, tp, s, and the colon (:).

Parentheses group multiple pattern elements into a single element. The pipe symbol (|) indicates a choice between the elements on either side of it, f and ht. The question

mark (?) indicates that the preceding element, *s*, is optional. Thus, the preceding regular expression matches these strings:

- http:
- https:
- ftp:
- ftps:

Regular expressions are a powerful text-processing component of the programming languages Java and PERL. For example, a PERL script can read the contents of each HTML file in a directory into a single string variable and then use a regular expression to search that string for URLs. This robust pattern-matching functionality is one reason that many application developers use PERL.

8.2 Oracle SQL Support for Regular Expressions

Oracle SQL support for regular expressions lets application developers implement complex pattern-matching logic in the database, which is useful for these reasons:

- By centralizing pattern-matching logic in the 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 DNA 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.
- By using server-side regular expressions to enforce constraints, you avoid duplicating validation logic on multiple clients.

Oracle SQL supports regular expressions with the pattern-matching condition and functions summarized in [Table 8-1](#). Each pattern matcher searches a given string for a given pattern (described with a regular expression), and each has the pattern-matching options described in [Table 8-2](#). The functions have additional options (for example, the character position at which to start searching the string for the pattern). For details, see *Oracle Database SQL Language Reference*.

Table 8-1 Oracle SQL Pattern-Matching Condition and Functions

Name	Description
REGEXP_LIKE	<p>Condition that can appear in the <code>WHERE</code> clause of a query, causing the query to return rows that match the given pattern.</p> <p>Example: This <code>WHERE</code> clause identifies employees with the first name of Steven or Stephen:</p> <pre>WHERE REGEXP_LIKE((hr.employees.first_name, '^Ste(v ph)en\$')</pre>
REGEXP_COUNT	<p>Function that returns the number of times the given pattern appears in the given string.</p> <p>Example: This function invocation returns the number of times that e (but not E) appears in the string 'Albert Einstein', starting at character position 7:</p> <pre>REGEXP_COUNT('Albert Einstein', 'e', 7, 'c')</pre> <p>(The returned value is 1, because the c option specifies case-sensitive matching.)</p>

Table 8–1 (Cont.) Oracle SQL Pattern-Matching Condition and Functions

Name	Description
REGEXP_INSTR	<p>Function that returns an integer that indicates the starting position of the given pattern in the given string. Alternatively, the integer can indicate the position immediately following the end of the pattern.</p> <p>Example: This function invocation returns the starting position of the first valid email address in the column <code>hr.employees.email</code>:</p> <pre>REGEXP_INSTR(hr.employees.email, '\w+@\w+(\.\w+)+')</pre> <p>If the returned value is greater than zero, then the column contains a valid email address.</p>
REGEXP_REPLACE	<p>Function that returns the string that results from replacing occurrences of the given pattern in the given string with a replacement string.</p> <p>Example: This function invocation puts a space after each character in the column <code>hr.countries.country_name</code>:</p> <pre>REGEXP_REPLACE(hr.countries.country_name, '(.)', '\1 ')</pre>
REGEXP_SUBSTR	<p>Function that is like <code>REGEXP_INSTR</code> except that instead of returning the starting position of the given pattern in the given string, it returns the matching substring itself.</p> <p>Example: This function invocation returns 'Oracle' because the <code>x</code> option ignores the spaces in the pattern:</p> <pre>REGEXP_SUBSTR('Oracle 2010', 'O r a c l e', 1, 1, 'x')</pre>

Table 8–2 describes the pattern-matching options that are available to each pattern matcher in [Table 8–1](#).

Table 8–2 Pattern-Matching Options for Oracle SQL Pattern-Matching Condition and Functions

Option	Description	Example
i	Specifies case-insensitive matching.	This function invocation returns 3: <pre>REGEXP_COUNT('Albert Einstein', 'e', 'i')</pre>
c	Specifies case-sensitive matching.	This function invocation returns 2: <pre>REGEXP_COUNT('Albert Einstein', 'e', 'c')</pre>
n	Allows the Dot operator (.) to match the newline character, which is not the default (see Table 8–3).	In this function invocation, the string and search pattern match only because the <code>n</code> option is specified: <pre>REGEXP_SUBSTR('a' CHR(10) 'd', 'a.d', 1, 1, 'n')</pre>
m	<p>Specifies multiline mode, where a newline character inside a string terminates a line. The string can contain multiple lines.</p> <p>Multiline mode affects POSIX operators Beginning-of-Line Anchor (^) and End-of-Line Anchor (\$) (described in Table 8–3) but not PERL-influenced operators \A, \Z, and \z (described in Table 8–5).</p>	<p>This function invocation returns ac:</p> <pre>REGEXP_SUBSTR('ab' CHR(10) 'ac', '^a.', 1, 2, 'm')</pre>
x	Ignores whitespace characters in the search pattern. By default, whitespace characters match themselves.	<p>This function invocation returns abcd:</p> <pre>REGEXP_SUBSTR('abcd', 'a b c d', 1, 1, 'x')</pre>

8.3 Oracle SQL and POSIX Regular Expression Standard

Oracle SQL implementation of regular expressions conforms to these standards:

- IEEE Portable Operating System Interface (POSIX) standard draft 1003.2/D11.2

Oracle SQL follows exactly the syntax and matching semantics for regular expression operators as defined in the POSIX standard for matching ASCII (English language) data. You can find the POSIX standard draft at this URL:

<http://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html>

For more information, see "POSIX Operators in Oracle SQL Regular Expressions" on page 8-4.

- Unicode Regular Expression Guidelines of the Unicode Consortium

Oracle SQL extends regular expression support beyond the POSIX standard in these ways:

- Extends the matching capabilities for multilingual data

For details, see [Section 8.4.2, "Oracle SQL Multilingual Extensions to POSIX Standard."](#)

- Supports some commonly used PERL regular expression operators that are not included in the POSIX standard but do not conflict with it (for example, character class shortcuts and the nongreedy modifier (?))

For details, see [Section 8.4.3, "Oracle SQL PERL-Influenced Extensions to POSIX Standard."](#)

8.4 Operators in Oracle SQL Regular Expressions

Oracle SQL supports a set of common operators (composed of metacharacters) used in regular expressions.

Caution: 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 Oracle SQL supports their syntax and interprets them as you expect.

Topics:

- [POSIX Operators in Oracle SQL Regular Expressions](#)
- [Oracle SQL Multilingual Extensions to POSIX Standard](#)
- [Oracle SQL PERL-Influenced Extensions to POSIX Standard](#)

8.4.1 POSIX Operators in Oracle SQL Regular Expressions

Table 8-3 summarizes the POSIX operators defined in the POSIX standard Extended Regular Expression (ERE) syntax. Oracle SQL follows the exact syntax and matching semantics for these operators as defined in the POSIX standard for matching ASCII (English language) data. Any differences in action between Oracle SQL and the POSIX standard are noted in the Description column.

Table 8-3 *POSIX Operators in Oracle SQL Regular Expressions*

Operator Syntax	Names	Description	Examples
.	Any Character Dot	Matches any character in the database character set, including the newline character if you specify matching option <code>n</code> (see Table 8-2). The Linux, UNIX, and Windows platforms recognize the newline character as the linefeed character (<code>\x0a</code>). The Macintosh platforms recognize the newline character as the carriage return character (<code>\x0d</code>). Note: In the POSIX standard, this operator matches any English character except NULL and the newline character.	The expression <code>a.b</code> matches the strings <code>abb</code> , <code>acb</code> , and <code>adb</code> , but does not match <code>acc</code> .
+	One or More Plus Quantifier	Matches one or more occurrences of the preceding subexpression (greedy ¹).	The expression <code>a+</code> matches the strings <code>a</code> , <code>aa</code> , and <code>aaa</code> , but does not match <code>ba</code> or <code>ab</code> .
*	Zero or More Star Quantifier	Matches zero or more occurrences of the preceding subexpression (greedy ¹).	The expression <code>ab*c</code> matches the strings <code>ac</code> , <code>abc</code> , and <code>abbc</code> , but does not match <code>abb</code> or <code>bbc</code> .
?	Zero or One Question Mark Quantifier	Matches zero or one occurrences of the preceding subexpression (greedy ¹).	The expression <code>ab?c</code> matches the strings <code>abc</code> and <code>ac</code> , but does not match <code>abbc</code> or <code>adc</code> .
{ <i>m</i> }	Interval Exact Count	Matches exactly <i>m</i> occurrences of the preceding subexpression.	The expression <code>a{3}</code> matches the string <code>aaa</code> , but does not match <code>aa</code> .
{ <i>m</i> , <i>n</i> }	Interval At-Least Count	Matches at least <i>m</i> occurrences of the preceding subexpression (greedy ¹).	The expression <code>a{3,}</code> matches the strings <code>aaa</code> and <code>aaaa</code> , but does not match <code>aa</code> .
{ <i>m</i> , <i>n</i> }	Interval Between Count	Matches at least <i>m</i> but not more than <i>n</i> occurrences of the preceding subexpression (greedy ¹).	The expression <code>a{3,5}</code> matches the strings <code>aaa</code> , <code>aaaa</code> , and <code>aaaaaa</code> , but does not match <code>aa</code> or <code>aaaaaa</code> .
[<i>char...</i>]	Matching Character List	Matches any single character in the list within the brackets. In the list, all operators except these are treated as literals: <ul style="list-style-type: none">■ Range operator: <code>-</code>■ POSIX character class: <code>[: :]</code>■ POSIX collation element: <code>[. .]</code>■ POSIX character equivalence class: <code>[= =]</code> A dash (<code>-</code>) is a literal when it occurs first or last in the list, or as an ending range point in a range expression, as in <code>[#--]</code> . A right bracket (<code>]</code>) 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 value ranges; the semantics of the range expression are independent of the character set. In Oracle Database, the linguistic range is determined by the <code>NLS_SORT</code> initialization parameter.	The expression <code>[abc]</code> matches the first character in the strings <code>all</code> , <code>bill</code> , and <code>cold</code> , but does not match any characters in <code>doll</code> .
[^ <i>char...</i>]	Nonmatching Character List	Matches any single character <i>not</i> in the list within the brackets. For information about operators and ranges in the character list, see the description of the Matching Character List operator.	The expression <code>[^abc]def</code> matches the string <code>xdef</code> , but not <code>adef</code> , <code>bdef</code> , or <code>cdef</code> . The expression <code>[^a-i]x</code> matches the string <code>jx</code> , but does not match <code>ax</code> , <code>fx</code> , or <code>ix</code> .
[<i>alt1</i> <i>alt2</i>]	Or	Matches either alternative.	The expression <code>a b</code> matches the character <code>a</code> or <code>b</code> .

Table 8-3 (Cont.) POSIX Operators in Oracle SQL Regular Expressions

Operator Syntax	Names	Description	Examples
(expr)	Subexpression Grouping	Treats the expression within the parentheses as a unit. The expression can be a string or a complex expression containing operators. You can refer to a subexpression in a back reference.	The expression (abc)?def matches the strings abcdef and def, but does not match abcdefg or xdef.
\n	Back Reference	Matches the n^{th} preceding subexpression, where n is an integer from 1 through 9. A back reference counts subexpressions from left to right, starting with the opening parenthesis of each preceding subexpression. The expression is invalid if fewer than n subexpressions precede \n. A back reference lets you search for a repeated string without knowing what it is. For the REGEXP_REPLACE function, Oracle SQL supports back references in both the regular expression pattern and the replacement string.	The expression (abc def)xy\1 matches the strings abcxyabc and defxydef, but does not match abcxydef or abcxy. The expression ^(.*)\1\$ matches a line consisting of two adjacent instances of the same string.
\	Escape Character	Treats the subsequent character as a literal. A backslash (\) lets you search for a character that would otherwise be treated as a metacharacter. Use consecutive backslashes (\) to match the backslash literal itself.	The expression abc\+def matches the string abc+def, but does not match abcdef or abccdef.
^	Beginning-of-Line Anchor	Default mode: Matches the beginning of a string. Multiline mode: ² Matches the beginning of any line the source string.	The expression ^def matches the substring def in the string defghi but not in the string abcdef.
\$	End-of-Line Anchor	Default mode: Matches the end of a string. Multiline mode: ² Matches the end of any line the source string.	The expression def\$ matches the substring def in the string abcdef but not in the string defghi.
[:class:]	POSIX Character Class	Matches any character in the specified POSIX character class (such as uppercase characters, digits, or punctuation characters). 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.	The expression [:upper:]+ , which specifies one or more consecutive uppercase characters, matches the substring DEF in the string abcDEFghi, but does not match any substring in abcdefghi.
[.element.]	POSIX Collating Element Operator	Specifies a collating element defined in the current locale. The NLS_SORT initialization parameter determines the supported collation elements. This syntax lets you use a multicharacter collating element where otherwise only single-character collating elements are 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.] , which specifies the collating element ch, matches ch in the string chabc, but does not match any substring in cdefg. The expression [a-[.ch.]] specifies the range from a through ch.
[=char=]	POSIX Character Equivalence Class	Matches all characters that belong to the same POSIX character equivalence class as the specified character, in the current locale. This syntax must appear within a character list; that is, it must be nested within the brackets for a character list. Character equivalents depend on how canonical rules are defined for your database locale. For details, see <i>Oracle Database Globalization Support Guide</i> .	The expression [=n=] , which specifies characters equivalent to n in a Spanish locale, matches both N and ñ in the string El Niño.

¹ A **greedy** operator matches as many occurrences as possible while allowing the rest of the match to succeed. To make the operator nongreedy, follow it with the nongreedy modifier (?) (see Table 8-5).

² Specify multiline mode with the pattern-matching option `m`, described in [Table 8–2](#).

8.4.2 Oracle SQL Multilingual Extensions to POSIX Standard

When applied to multilingual data, Oracle SQL POSIX operators extend beyond the matching capabilities specified in the POSIX standard.

[Table 8–4](#) shows, for each POSIX operator, which POSIX standards define its syntax and whether Oracle SQL extends its semantics for handling multilingual data. The POSIX standards are Basic Regular Expression (BRE) and Extended Regular Expression (ERE).

Table 8–4 *POSIX Operators 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

Multilingual data might have multibyte characters. Oracle Database lets you enter multibyte characters directly (if you have a direct input method) or use functions to compose them. You cannot use the Unicode hexadecimal encoding value of the form `\xxxx`. Oracle Database evaluates the characters based on the byte values used to encode the character, not the graphical representation of the character.

8.4.3 Oracle SQL PERL-Influenced Extensions to POSIX Standard

Oracle SQL supports some commonly used PERL regular expression operators that are not included in the POSIX standard but do not conflict with it.

[Table 8–5](#) summarizes the PERL-influenced operators that Oracle SQL supports.

Caution: PERL character class matching is based on the locale model of the operating system, whereas Oracle SQL regular expressions are based on the language-specific data of the database. In general, you cannot expect a regular expression involving locale data to produce the same results in PERL and Oracle SQL.

Table 8-5 PERL-Influenced Operators in Oracle SQL Regular Expressions

Operator Syntax	Description	Examples
\d	Matches a digit character. Equivalent to POSIX expression <code>[:digit:]</code> .	The expression <code>^\(\d{3}\) \d{3}-\d{4}\$</code> matches <code>(650) 555-0100</code> but does not match <code>650-555-0100</code> .
\D	Matches a nondigit character. Equivalent to POSIX expression <code>[^[:digit:]]</code> .	The expression <code>\w\d\D</code> matches <code>b2b</code> and <code>b2_</code> but does not match <code>b22</code> .
\w	Matches a word character (that is, an alphanumeric or underscore (<code>_</code>) character). Equivalent to POSIX expression <code>[:alnum:]_</code> .	The expression <code>\w@\w+(\.\w+)+</code> matches the string <code>jdoe@company.co.uk</code> but does not match <code>jdoe@company</code> .
\W	Matches a nonword character. Equivalent to POSIX expression <code>[^[:alnum:]_]</code> .	The expression <code>\w+\W\s\w+</code> matches the string <code>to: bill</code> but does not match <code>to bill</code> .
\s	Matches a whitespace character. Equivalent to POSIX expression <code>[:space:]</code> .	The expression <code>\(\w\s\w\s\)</code> matches the string <code>(a b)</code> but does not match <code>(ab)</code> or <code>(a,b.)</code> .
\S	Matches a nonwhitespace character. Equivalent to POSIX expression <code>[^[:space:]]</code> .	The expression <code>\(\w\S\w\S\)</code> matches the strings <code>(abde)</code> and <code>(a,b.)</code> but does not match <code>(a b d e)</code> .
\A	Matches the beginning of a string, in either single-line or multiline mode. Not equivalent to POSIX operator <code>^</code> .	The expression <code>\AL</code> matches only the first <code>L</code> in the string <code>Line1\nLine2\n</code> (where <code>\n</code> is the newline character), in either single-line or multiline mode.
\Z	Matches the end of a string, in either single-line or multiline mode. Not equivalent to POSIX operator <code>\$</code> .	The expression <code>\s\Z</code> matches the last space in the string <code>L i n e \n</code> (where <code>\n</code> is the newline character), in either single-line or multiline mode.
\z	Matches the end of a string, in either single-line or multiline mode. Not equivalent to POSIX operator <code>\$</code> .	The expression <code>\s\z</code> matches the newline character (<code>\n</code>) in the string <code>L i n e \n</code> , in either single-line or multiline mode.
+\?	Matches one or more occurrences of the preceding subexpression (nongreedy ¹).	The expression <code>\w+\?x\w</code> matches <code>abxc</code> in the string <code>abxcxd</code> (and the greedy expression <code>\w+x\w</code> matches <code>abxcxd</code>).
\?	Matches zero or more occurrences of the preceding subexpression (nongreedy ¹). Matches the empty string whenever possible.	The expression <code>\w\?x\w</code> matches <code>xa</code> in the string <code>xaxbxc</code> (and the greedy expression <code>\w*x\w</code> matches <code>xaxbxc</code>).
?\?	Matches zero or one occurrences of the preceding subexpression (nongreedy ¹). Matches the empty string whenever possible.	The expression <code>a?\?aa</code> matches <code>aa</code> in the string <code>aaaa</code> (and the greedy expression <code>a?aa</code> matches <code>aaa</code>).

Table 8–5 (Cont.) PERL-Influenced Operators in Oracle SQL Regular Expressions

Operator Syntax	Description	Examples
{m}?	Matches exactly m occurrences of the preceding subexpression (nongreedy) ¹ .	The expression (a aa){2}? matches aa in the string aaaa (and the greedy expression (a aa){2} matches aaaa). Both the expression b{2}? and the greedy expression b{2} match bb in the string bbbb.
{m, }?	Matches at least m occurrences of the preceding subexpression (nongreedy) ¹ .	The expression a{2,}? matches aa in the string aaaaa (and the greedy expression a{2,} matches aaaa).
{m, n}?	Matches at least m but not more than n occurrences of the preceding subexpression (nongreedy ¹). {0,n}? matches the empty string whenever possible.	The expression a{2,4}? matches aa in the string aaaaa (and the greedy expression a{2,4} matches aaaa).

¹ A **nongreedy** operator matches as few occurrences as possible while allowing the rest of the match to succeed. To make the operator greedy, omit the nongreedy modifier (?).

8.5 Using Regular Expressions in SQL Statements: Scenarios

Scenarios:

- [Using a Constraint to Enforce a Phone Number Format](#)
- [Using Back References to Reposition Characters](#)

8.5.1 Using a Constraint to Enforce a Phone Number Format

Regular expressions are useful for enforcing constraints—for example, to ensure that phone numbers are entered into the database in a standard format. [Example 8–1](#) creates a contacts table and adds a CHECK constraint to the p_number column to enforce this format model:

(XXX) XXX-XXXX

Example 8–1 Enforcing a Phone Number Format with Regular Expressions

```
DROP TABLE contacts;
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 8–6](#) explains the elements of the regular expression.

Table 8–6 Explanation of the Regular Expression Elements in Example 8–1

Regular Expression Element	Matches . . .
^	The beginning of the string.
\(A left parenthesis. The backslash (\) is an escape character that indicates that the left parenthesis after it is a literal rather than a subexpression delimiter.
\d{3}	Exactly three digits.

Table 8–6 (Cont.) Explanation of the Regular Expression Elements in Example 8–1

Regular Expression Element	Matches . . .
\)	A right parenthesis. The backslash (\) is an escape character that indicates that the right parenthesis after it is a literal rather than a subexpression delimiter.
space character	A space character.
\d{3}	Exactly three digits.
-	A hyphen.
\d{4}	Exactly four digits.
\$	The end of the string.

[Example 8–2](#) shows some statements that correctly and incorrectly insert phone numbers into the contacts table.

Example 8–2 Inserting Phone Numbers in Correct and Incorrect Formats

These are correct:

```
INSERT INTO contacts (p_number) VALUES('(650) 555-0100');
INSERT INTO contacts (p_number) VALUES('(215) 555-0100');
```

These generate CHECK constraint errors:

```
INSERT INTO contacts (p_number) VALUES('650 555-0100');
INSERT INTO contacts (p_number) VALUES('650 555 0100');
INSERT INTO contacts (p_number) VALUES('650-555-0100');
INSERT INTO contacts (p_number) VALUES('(650)555-0100');
INSERT INTO contacts (p_number) VALUES(' (650) 555-0100');
```

8.5.2 Using Back References to Reposition Characters

A back reference (described in [Table 8–3](#)) stores the referenced subexpression in a temporary buffer. Therefore, you can use back references to reposition characters, as in [Example 8–3](#). For an explanation of the elements of the regular expression in [Example 8–3](#), see [Table 8–7](#).

Example 8–3 Using Back References to Reposition Characters

Create table and populate it with names in different formats:

```
DROP TABLE famous_people;
CREATE TABLE famous_people (names VARCHAR2(20));
INSERT INTO famous_people (names) VALUES ('John Quincy Adams');
INSERT INTO famous_people (names) VALUES ('Harry S. Truman');
INSERT INTO famous_people (names) VALUES ('John Adams');
INSERT INTO famous_people (names) VALUES (' John Quincy Adams');
INSERT INTO famous_people (names) VALUES ('John_Quincy_Adams');
```

SQL*Plus formatting command:

```
COLUMN "names after regexp" FORMAT A20
```

For each name in the table whose format is "first middle last", use back references to reposition characters so that the format becomes "last, first middle":

```

SELECT names "names",
  REGEXP_REPLACE(names, '^(\S+)\s(\S+)\s(\S+)\$', '\3, \1 \2')
  AS "names after regexp"
FROM famous_people
ORDER BY "names";

```

Result:

names	names after regexp
John Quincy Adams	John Quincy Adams
Harry S. Truman	Truman, Harry S.
John Adams	John Adams
John Quincy Adams	Adams, John Quincy
John_Quincy_Adams	John_Quincy_Adams

5 rows selected.

[Table 8–7](#) explains the elements of the regular expression.

Table 8–7 Explanation of the Regular Expression Elements in Example 8–3

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.

Using Indexes in Database Applications

Indexes are optional structures, associated with tables and clusters, which allow SQL queries to execute more quickly. Just as the index in this guide helps you locate information faster than if there were no index, an Oracle Database index provides a faster access path to table data. You can use indexes without rewriting any queries. Your results are the same, but you see them more quickly.

This chapter supplements this information:

- The explanation of indexes and index-organized tables in *Oracle Database Concepts*
- The information about managing indexes in *Oracle Database Administrator's Guide*
- The explanation of how indexes and clusters can enhance or degrade performance in *Oracle Database SQL Tuning Guide*

Topics:

- [Guidelines for Managing Indexes](#)
- [Managing Indexes](#)
- [When to Use Domain Indexes](#)
- [When to Use Function-Based Indexes](#)

9.1 Guidelines for Managing Indexes

Here is a summary of the guidelines for managing indexes. For details, see *Oracle Database Administrator's Guide*:

- Create indexes after inserting table data
- Index the correct tables and columns
- Order index columns for performance
- Limit the number of indexes for each table
- Drop indexes that are no longer needed
- Understand deferred segment creation
- Estimate index size and set storage parameters
- Specify the tablespace for each index
- Consider parallelizing index creation
- Consider creating indexes with NOLOGGING
- Understand when to use unusable or invisible indexes

- Consider costs and benefits of coalescing or rebuilding indexes
- Consider cost before disabling or dropping constraints

9.2 Managing Indexes

Oracle Database Administrator's Guide has this information about managing indexes:

- Creating indexes
- Altering indexes
- Monitoring space use of indexes
- Dropping indexes
- Data dictionary views that display information about indexes

See Also: [Section 10.3, "Creating Indexes for Use with Constraints"](#)

9.3 When to Use Domain Indexes

A **domain index** (also called an **application domain index**) is a customized index specific to an application that was implemented using a data cartridge (for example, a search engine or geographic information system). For information about domain indexes, see *Oracle Database Concepts*.

See Also: [Oracle Database Data Cartridge Developer's Guide](#) for conceptual background to help you decide when to build domain indexes

9.4 When to Use Function-Based Indexes

A **function-based index** computes the value of an expression that involves one or more columns and stores it in the index. The index expression can be an arithmetic expression or an expression that contains a SQL function, PL/SQL function, package function, or C callout. Function-based indexes also support linguistic sorts based on collation keys, efficient linguistic collation of SQL statements, and case-insensitive sorts.

A function-based index improves the performance of queries that use the index expression (especially if the expression is computationally intensive). However:

- The database must also evaluate the index expression to process statements that do not use it.
- Function-based indexes on columns that are frequently modified are expensive for the database to maintain.

The optimizer can use function-based indexes only for cost-based optimization, while it can use indexes on columns for both cost-based and rule-based optimization.

Note:

- A function-based index cannot contain the value `NULL`. Therefore, either ensure that no column involved in the index expression can contain `NULL` or use the `NVL` function in the index expression to substitute another value for `NULL`. (For information about `NVL`, see *Oracle Database SQL Language Reference*.)
 - Oracle Database treats descending indexes as if they were function-based indexes (for more information, see *Oracle Database SQL Language Reference*).
-

Topics:

- [Advantages of Function-Based Indexes](#)
- [Disadvantages of Function-Based Indexes](#)
- [Examples of Function-Based Indexes](#)

See Also:

- *Oracle Database Concepts* for additional conceptual information about function-based indexes
- *Oracle Database Administrator's Guide* for information about creating function-based indexes
- *Oracle Database Globalization Support Guide* for information about function-based linguistic indexes
- *Oracle Database Concepts* for more information about how the optimizer uses function-based indexes
- *Oracle Database SQL Tuning Guide* for information about using function-based indexes for performance

9.4.1 Advantages of Function-Based Indexes

A function-based index has these advantages:

- A function-based index increases the number of situations where the database can perform an index range scan (described in *Oracle Database Concepts*) instead of a full index scan (described in *Oracle Database Concepts*).

An index range scan typically has a fast response time when the `WHERE` clause selects fewer than 15% of the rows of a large table. The optimizer can more accurately estimate how many rows an expression selects if the expression is materialized in a function-based index.

Oracle Database represents the index expression as a virtual column, on which the `ANALYZE` statement (described in *Oracle Database SQL Language Reference*) can build a histogram.

- A function-based index precomputes and stores the value of an expression.

Queries can get the value of the expression from the index instead of computing it. The more queries that need the value, and the more computationally intensive the index expression, the more the index improves application performance. (See [Example 9-1](#).)

- You can create a function-based index on an object column or REF column. The index expression can be the invocation of a method that returns an object type. For more information, see *Oracle Database Object-Relational Developer's Guide*. (See [Example 9-2](#) and the example in *Oracle Database SQL Language Reference*.)
- A function-based index lets you perform more powerful sorts. The index expression can invoke the SQL functions UPPER and LOWER for case-insensitive sorts (as in [Example 9-3](#)) and the SQL function NLSSORT for linguistic-based sorts (as in [Example 9-4](#)).

9.4.2 Disadvantages of Function-Based Indexes

A function-based index has these disadvantages:

- The optimizer can use a function-based index only for cost-based optimization, not for rule-based optimization. The cost-based optimizer uses statistics stored in the dictionary. To gather statistics for a function-based index, invoke either DBMS_STATS.GATHER_TABLE_STATS (described in *Oracle Database PL/SQL Packages and Types Reference*) or DBMS_STATS.GATHER_SCHEMA_STATS (described in *Oracle Database PL/SQL Packages and Types Reference*).
- The database does not use a function-based index until you analyze the index itself and the table on which it is defined. To analyze the index and the table on which the index is defined, invoke either DBMS_STATS.GATHER_TABLE_STATS or DBMS_STATS.GATHER_SCHEMA_STATS.
- The database does not use function-based indexes when doing OR expansion.
- You must ensure that any schema-level or package-level PL/SQL function that the index expression invokes is deterministic (that is, that the function always return the same result for the same input). You must declare the function as DETERMINISTIC, but because Oracle Database does not check this assertion, you must ensure that the function really is deterministic. If you change the semantics of a DETERMINISTIC function and recompile it, then you must manually rebuild any dependent function-based indexes and materialized views. Otherwise, they report results for the prior version of the function.
- If the index expression is a function invocation, then the function return type cannot be constrained. Because you cannot constrain the function return type with NOT NULL, you must ensure that the query that uses the index cannot fetch NULL values. Otherwise, the database performs a full table scan.
- The index expression cannot invoke an aggregate function (described in *Oracle Database SQL Language Reference*).
- A bitmapped function-based index cannot be a descending index (see *Oracle Database SQL Language Reference*).
- The data type of the index expression cannot be VARCHAR2, RAW, LONGRAW, or a PL/SQL data type of unknown length. That is, you cannot index an expression of unknown length. However, you can index a known-length substring of that expression. For example:

```

CREATE OR REPLACE FUNCTION initials (
  name  IN VARCHAR2
) RETURN VARCHAR2
DETERMINISTIC
IS
BEGIN
  RETURN('A. J.');
END;
/

/* Invoke SUBSTR both when creating index and when referencing
   function in queries. */

CREATE INDEX func_substr_index ON
EMPLOYEES(SUBSTR(initials(FIRST_NAME),1,10));

SELECT SUBSTR(initials(FIRST_NAME),1,10) FROM EMPLOYEES;

```

See Also:

- *Oracle Database SQL Language Reference* for notes on function-based indexes
- *Oracle Database SQL Language Reference* for restrictions on function-based indexes
- *Oracle Database PL/SQL Language Reference* for information about the CREATE FUNCTION statement, including restrictions

9.4.3 Examples of Function-Based Indexes

Examples:

- [Example 9-1, "Function-Based Index for Precomputing Arithmetic Expression"](#)
- [Example 9-2, "Function-Based Indexes on Object Column"](#)
- [Example 9-3, "Function-Based Index for Faster Case-Insensitive Searches"](#)
- [Example 9-4, "Function-Based Index for Language-Dependent Sorting"](#)

Note: [Example 9-1](#) and [Example 9-2](#) use composite indexes (indexes on multiple table columns), described in *Oracle Database Concepts*.

[Example 9-1](#) creates a table with columns a, b, and c; creates an index on the table, and then queries the table. The index is a composite index on three columns: a virtual column that represents the expression $a+b*(c-1)$, column a, and column b. The query uses the indexed expression in its WHERE clause; therefore, it can use an index range scan instead of a full index scan.

Example 9-1 Function-Based Index for Precomputing Arithmetic Expression

Create table on which to create index:

```

DROP TABLE Fbi_tab;
CREATE TABLE Fbi_tab (
  a INTEGER,
  b INTEGER,
  c INTEGER
);

```

Create index:

```
CREATE INDEX Idx ON Fbi_tab (a+b*(c-1), a, b);
```

This query can use an index range scan instead of a full index scan:

```
SELECT a FROM Fbi_tab WHERE a+b*(c-1) < 100;
```

See Also: *Oracle Database Concepts* for information about fast full index scans

In [Example 9–2](#), assume that the object type `Reg_obj` has been defined, and that it stores information about a city. The example creates a table whose first column has type `Reg_obj`, a deterministic function with a parameter of type `Reg_obj`, and two function-based indexes that invoke the function. The first query uses the first index to quickly find cities further than 1000 miles from the equator. The second query uses the second index (which is composite) to quickly find cities where the temperature delta is less than 20 and the maximum temperature is greater than 75. (The table is not populated for the example, so the queries return no rows.)

Example 9–2 Function-Based Indexes on Object Column

Create table with object column:

```
DROP TABLE Weatherdata_tab;
CREATE TABLE Weatherdata_tab (
  Reg_obj INTEGER,
  Maxtemp INTEGER,
  Mintemp INTEGER
);
```

Create deterministic function with parameter of type `Reg_obj`:

```
CREATE OR REPLACE FUNCTION Distance_from_equator (
  Reg_obj IN INTEGER
) RETURN INTEGER
  DETERMINISTIC
IS
BEGIN
  RETURN(3000);
END;
/
```

Create first function-based index:

```
CREATE INDEX Distance_index
ON Weatherdata_tab (Distance_from_equator (Reg_obj));
```

Use index expression in query:

```
SELECT * FROM Weatherdata_tab
WHERE (Distance_from_equator (Reg_Obj)) > '1000';
```

Result:

```
no rows selected
```

Create second function-based (and composite) index:

```
CREATE INDEX Compare_index
ON Weatherdata_tab ((Maxtemp - Mintemp) DESC, Maxtemp);
```

Use index expression and indexed column in query:

```
SELECT * FROM Weatherdata_tab
WHERE ((Maxtemp - Mintemp) < '20' AND Maxtemp > '75');
```

Result:

```
no rows selected
```

Example 9–3 creates an index that allows faster case-insensitive searches in the EMPLOYEES table and then uses the index expression in a query.

Example 9–3 Function-Based Index for Faster Case-Insensitive Searches

Create index:

```
CREATE INDEX emp_lastname ON EMPLOYEES (UPPER(LAST_NAME));
```

Use index expression in query:

```
SELECT first_name, last_name
FROM EMPLOYEES
WHERE UPPER(LAST_NAME) LIKE 'J%S_N';
```

Result:

FIRST_NAME	LAST_NAME
Charles	Johnson

```
1 row selected.
```

Example 9–4 creates a table with one column, NAME, and a function-based index to sort that column using the collation sequence GERMAN, and then selects all columns of the table, ordering them by NAME. Because the query can use the index, the query is faster. (Assume that the query is run in a German session, where NLS_SORT is GERMAN and NLS_COMP is ANSI. Otherwise, the query would have to specify the values of these Globalization Support parameters.)

Example 9–4 Function-Based Index for Language-Dependent Sorting

Create table on which to create index:

```
DROP TABLE nls_tab;
CREATE TABLE nls_tab (NAME VARCHAR2(80));
```

Create index:

```
CREATE INDEX nls_index
  ON nls_tab (NLSSORT(NAME, 'NLS_SORT = GERMAN'));
```

Select all table columns, ordered by NAME:

```
SELECT * FROM nls_tab
WHERE NAME IS NOT NULL
ORDER BY NAME;
```


Maintaining Data Integrity in Database Applications

In a database application, **maintaining data integrity** means ensuring that the data in the tables that the application manipulates conform to the appropriate business rules. A **business rule** specifies a condition or relationship that must always be true or must always be false. For example, a business rule might be that no employee can have a salary over \$100,000 or that every employee in the `EMPLOYEES` table must belong to a department in the `DEPARTMENTS` table. Business rules vary from company to company, because each company defines its own policies about salaries, employee numbers, inventory tracking, and so on.

As explained in *Oracle Database Concepts*, there are several ways to ensure data integrity, and the one to use whenever possible is the **integrity constraint** (or **constraint**).

This chapter supplements this information:

- The explanation of data integrity and constraints in *Oracle Database Concepts*
- The information about managing constraints in *Oracle Database Administrator's Guide*
- The syntactic and semantic information about constraints in *Oracle Database SQL Language Reference*

Note: This chapter applies to only to constraints on tables. Constraints on views do not help maintain data integrity or have associated indexes. Instead, they enable query rewrites on queries involving views, thereby improving performance when using materialized views and other data warehousing features.

For more information about constraints on views, see *Oracle Database SQL Language Reference*.

For information about using constraints in data warehouses, see *Oracle Database Data Warehousing Guide*.

Topics:

- [Enforcing Business Rules with Constraints](#)
- [Enforcing Business Rules with Both Constraints and Application Code](#)
- [Creating Indexes for Use with Constraints](#)
- [When to Use NOT NULL Constraints](#)

- When to Use Default Column Values
- Choosing a Primary Key for a Table (PRIMARY KEY Constraint)
- When to Use UNIQUE Constraints
- Enforcing Referential Integrity with FOREIGN KEY Constraints
- Minimizing Space and Time Overhead for Indexes Associated with Constraints
- Guidelines for Indexing Foreign Keys
- Referential Integrity in a Distributed Database
- When to Use CHECK Constraints
- Examples of Defining Constraints
- Enabling and Disabling Constraints
- Modifying Constraints
- Renaming Constraints
- Dropping Constraints
- Managing FOREIGN KEY Constraints
- Viewing Information About Constraints

See Also: [Section 7.1.1, "How the Correct Data Type Increases Data Integrity,"](#) for information about the role that data type plays in data integrity

10.1 Enforcing Business Rules with Constraints

Whenever possible, enforce business rules with constraints. In addition to the advantages explained in *Oracle Database Concepts*, constraints have the advantage of speed: Oracle Database can check that all the data in a table obeys a constraint faster than application code can do the equivalent checking.

[Example 10-1](#) creates a table of departments, a table of employees, a constraint to enforce the rule that all values in the department table are unique, and a constraint to enforce the rule that every employee must work for a valid department.

Example 10-1 Enforcing Business Rules with Constraints

Create table of departments:

```
DROP TABLE dept_tab;
CREATE TABLE dept_tab (
    deptname VARCHAR2(20),
    deptno    INTEGER
);
```

Create table of employees:

```
DROP TABLE emp_tab;
CREATE TABLE emp_tab (
    empname VARCHAR2(80),
    empno   INTEGER,
    deptno  INTEGER
);
```

Create constraint to enforce rule that all values in department table are unique:

```
ALTER TABLE dept_tab ADD PRIMARY KEY (deptno);
```

Create constraint to enforce rule that every employee must work for a valid department:

```
ALTER TABLE emp_tab ADD FOREIGN KEY (deptno) REFERENCES dept_tab(deptno);
```

Now, whenever you insert an employee record into emp_tab, Oracle Database checks that its deptno value appears in dept_tab.

Suppose that instead of using a constraint to enforce the rule that every employee must work for a valid department, you use a trigger that queries dept_tab to check that it contains the deptno value of the employee record to be inserted into emp_tab. Because the query uses consistent read (CR), it might miss uncommitted changes from other transactions. For more information about using triggers to enforce business rules, see *Oracle Database Concepts*.

See Also: *Oracle Database SQL Language Reference* for syntactic and semantic information about constraints

10.2 Enforcing Business Rules with Both Constraints and Application Code

Enforcing business rules with both constraints and application code is recommended when application code can determine that data values are invalid without querying tables. The application code can provide immediate feedback to the user and reduce the load on the database by preventing attempts to insert invalid data into tables.

For [Example 10–2](#), assume that [Example 10–1](#) was run and then this column was added to the table emp_tab:

```
empgender VARCHAR2(1)
```

The only valid values for empgender are 'M' and 'F'. When someone tries to insert a row into emp_tab or update the value of emp_tab.empgender, application code can determine whether the new value for emp_tab.empgender is valid without querying a table. If the value is invalid, the application code can notify the user instead of trying to insert the invalid value, as in [Example 10–2](#).

Example 10–2 Enforcing Business Rules with Both Constraints and Application Code

```
CREATE OR REPLACE PROCEDURE add_employee (
    e_name    emp_tab.empname%TYPE,
    e_gender  emp_tab.empgender%TYPE,
    e_number  emp_tab.empno%TYPE,
    e_dept    emp_tab.deptno%TYPE
) AUTHID DEFINER IS
BEGIN
    IF UPPER(e_gender) IN ('M','F') THEN
        INSERT INTO emp_tab VALUES (e_name, e_gender, e_number, e_dept);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Gender must be M or F.');
    END IF;
END;
/

BEGIN
    add_employee ('Smith', 'M', 356, 20);
END;
```

/

Result:

```
Gender must be M or F.
```

10.3 Creating Indexes for Use with Constraints

When a unique or primary key constraint is enabled, Oracle Database creates an index automatically, but Oracle recommends that you create these indexes explicitly. If you want to use an index with a foreign key constraint, then you must create the index explicitly. For information about creating indexes explicitly, see *Oracle Database Administrator's Guide* or *Oracle Database SQL Language Reference*.

When a constraint can use an existing index, Oracle Database does not create an index for that constraint. Note that:

- A unique or primary key constraint can use either a unique index, an entire nonunique index, or the first few columns of a nonunique index.
- If a unique or primary key constraint uses a nonunique index, then no other unique or primary key constraint can use that nonunique index.
- The column order in the constraint and index need not match.
- The object number of the index used by a unique or primary key constraint is stored in CDEF\$.ENABLED for that constraint. No static data dictionary view or dynamic performance view shows this information.

If an enabled unique or primary key constraint is using an index, you cannot drop only the index. To drop the index, you must either drop the constraint itself or disable the constraint and then drop the index. For information about disabling and dropping constraints, see *Oracle Database Administrator's Guide*.

See Also:

- *Oracle Database Administrator's Guide* for more information about indexes associated with constraints
- [Chapter 9, "Using Indexes in Database Applications"](#)

10.4 When to Use NOT NULL Constraints

By default, a column can contain a `NULL` value. To ensure that the column never contains a `NULL` value, use the NOT NULL constraint (described in *Oracle Database SQL Language Reference*).

Use a NOT NULL constraint in both of these situations:

- A column must contain a non-`NULL` value.

For example, in the table `HR.EMPLOYEES`, each employee must have an employee ID. Therefore, the column `HR.EMPLOYEES.EMPLOYEE_ID` has a NOT NULL constraint, and nobody can insert a new employee record into `HR.EMPLOYEES` without specifying a non-`NULL` value for `EMPLOYEE_ID`. You *can* insert a new employee record into `HR.EMPLOYEES` without specifying a salary; therefore, the column `HR.EMPLOYEES.SALARY` does *not* have a NOT NULL constraint.

- You want to allow index scans of the table, or allow an operation that requires indexing all rows.

Oracle Database indexes do not store keys whose values are all NULL. Therefore, for the preceding kinds of operations, at least one indexed column must have a NOT NULL constraint.

Example 10–3 uses the SQL*Plus command DESCRIBE to show which columns of the DEPARTMENTS table have NOT NULL constraints, and then shows what happens if you try to insert NULL values in columns that have NOT NULL constraints.

Example 10–3 Inserting NULL Values into Columns with NOT NULL Constraints

```
DESCRIBE DEPARTMENTS;
```

Result:

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

Try to insert NULL into DEPARTMENT_ID column:

```
INSERT INTO DEPARTMENTS (
  DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID
)
VALUES (NULL, 'Sales', 200, 1700);
```

Result:

```
VALUES (NULL, 'Sales', 200, 1700)
*
ERROR at line 4:
ORA-01400: cannot insert NULL into ("HR"."DEPARTMENTS"."DEPARTMENT_ID")
```

Omitting a value for a column that cannot be NULL is the same as assigning it the value NULL:

```
INSERT INTO DEPARTMENTS (
  DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID
)
VALUES ('Sales', 200, 1700);
```

Result:

```
INSERT INTO DEPARTMENTS (
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("HR"."DEPARTMENTS"."DEPARTMENT_ID")
```

You can prevent the preceding error by giving DEPARTMENT_ID a non-NULL default value. For more information, see [Section 10.5, "When to Use Default Column Values."](#)

You can combine NOT NULL constraints with other constraints to further restrict the values allowed in specific columns. For example, the combination of NOT NULL and UNIQUE constraints forces the input of values in the UNIQUE key, eliminating the possibility that data in a new conflicts with data in an existing row. For more information, see ["UNIQUE and NOT NULL Constraints on the Foreign Key"](#) on page 10-11.

10.5 When to Use Default Column Values

When an `INSERT` statement (described in *Oracle Database SQL Language Reference*) does not specify a value for a specific column, that column receives its default value. By default, that default value is `NULL`. You can change the default value when you define the column (with the `CREATE TABLE` statement, described in *Oracle Database SQL Language Reference*) or when you alter the column (with the `ALTER TABLE` statement, described in *Oracle Database SQL Language Reference*).

Note: Giving a column a non-`NULL` default value does not ensure that the value of the column will never have the value `NULL`, as the `NOT NULL` constraint does. For information about the `NOT NULL` constraint, see [Section 10.4, "When to Use NOT NULL Constraints."](#)

Use a default column value in these situations:

- The column has a `NOT NULL` constraint.

Giving the column a non-`NULL` default value prevents the error that would occur if someone inserted a row without specifying a value for the column.

- There is a most common value for the column.

For example, if most departments in the company are in New York, then set the default value of the column `DEPARTMENTS.LOCATION` to '`NEW YORK`'.

- There is a non-`NULL` value that signifies no entry.

For example, if the value zero in the column `EMPLOYEES.SALARY` means that the salary has not yet been determined, then set the default value of that column to zero.

A default column value that signifies no entry can simplify testing. For example, it lets you change this test:

```
IF (employees.salary IS NOT NULL) AND (employees.salary < 50000)
```

To this test:

```
IF employees.salary < 50000
```

- You want to automatically record the names of users who modify a table.

For example, suppose that you allow users to insert rows into a table through a view. You give the base table a column named `insertter` (which need not be included in the definition of the view), to store the name of the user who inserted the row. To record the user name automatically, define a default value that invokes the `USER` function. For example:

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

10.6 Choosing a Primary Key for a Table (PRIMARY KEY Constraint)

The primary key of a table uniquely identifies each row and ensures that no duplicate rows exist (and typically, this is its only purpose). Therefore, a primary key value cannot be `NULL`.

A table can have at most one primary key, but that key can have multiple columns (that is, it can be a composite key). To designate a primary key, use the PRIMARY KEY constraint.

Whenever practical, choose as the primary key a single column whose values are generated by a sequence. For information about sequences, see *Oracle Database SQL Language Reference*.

The second-best choice for a primary key is a single column whose values are all of the following:

- Unique
- Never changed
- Never NULL
- Short and numeric (and therefore easy to type)

Minimize your use of composite primary keys, whose values are long and cannot be generated by a sequence.

See Also:

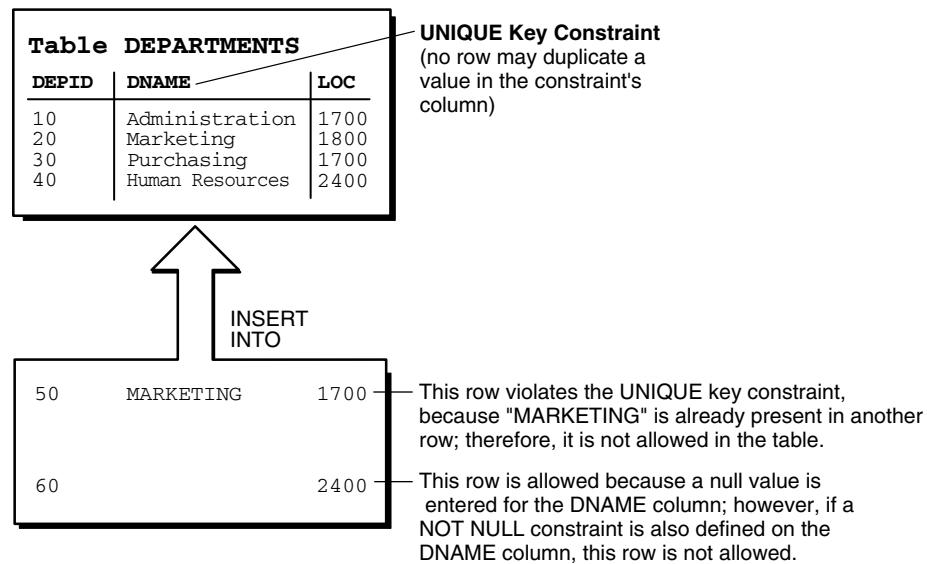
- *Oracle Database Concepts* for general information about primary key constraints
- *Oracle Database SQL Language Reference* for complete information about primary key constraints, including restrictions

10.7 When to Use UNIQUE Constraints

Use a UNIQUE constraint (which designates a unique key) on any column or combination of columns (except the primary key) where duplicate non-NULL values are not allowed. For example:

Unique Key	Primary Key
Employee Social Security Number	Employee number
Truck license plate number	Truck number
Customer phone number (country code column, area code column, and local phone number column)	Customer number
Department name column and location column	Department number

[Figure 10-1](#) shows a table with a UNIQUE constraint, a row that violates the constraint, and a row that satisfies it.

Figure 10–1 Rows That Violate and Satisfy a UNIQUE Constraint**See Also:**

- *Oracle Database Concepts* for general information about UNIQUE constraints
- *Oracle Database SQL Language Reference* for complete information about UNIQUE constraints, including restrictions

10.8 Enforcing Referential Integrity with FOREIGN KEY Constraints

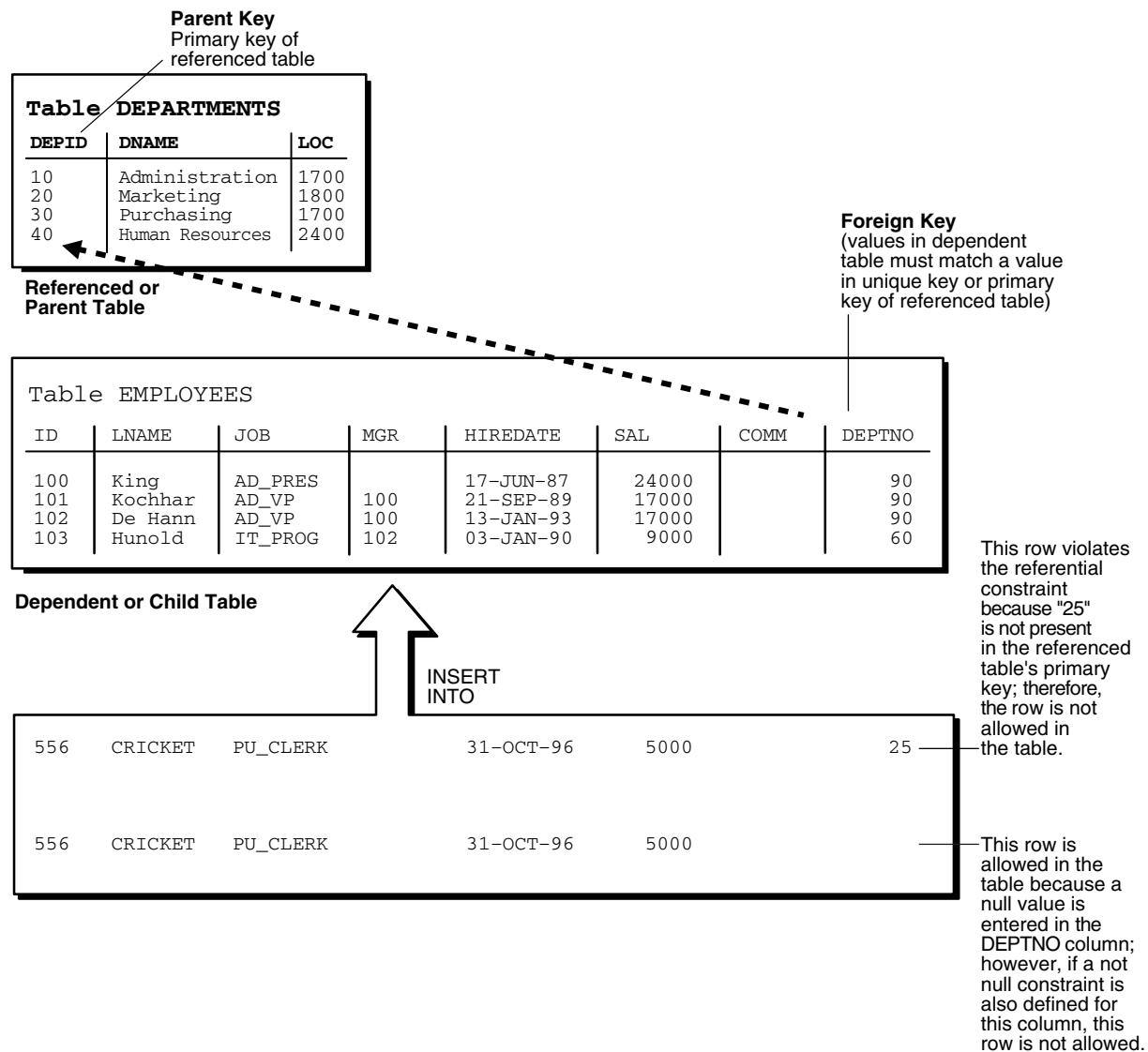
When two tables share one or more columns, you can use a FOREIGN KEY constraint to enforce referential integrity—that is, to ensure that the shared columns always have the same values in both tables.

Note: A FOREIGN KEY constraint is also called a **referential integrity constraint**, and its CONSTRAINT_TYPE is R in the static data dictionary views *_CONSTRAINTS.

Designate one table as the **referenced or parent table** and the other as the **dependent or child table**. In the parent table, define either a PRIMARY KEY or UNIQUE constraint on the shared columns. In the child table, define a FOREIGN KEY constraint on the shared columns. The shared columns now comprise a **foreign key**. Defining additional constraints on the foreign key affects the parent-child relationship—for details, see [Section 10.8.2, "Defining Relationships Between Parent and Child Tables."](#)

[Figure 10–2](#) 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.

[Figure 10–2](#) shows parent and child tables that share one column, a row that violates the FOREIGN KEY constraint, and a row that satisfies it.

Figure 10–2 Rows That Violate and Satisfy a FOREIGN KEY Constraint**Topics:**

- [FOREIGN KEY Constraints and NULL Values](#)
- [Defining Relationships Between Parent and Child Tables](#)
- [Rules for Multiple FOREIGN KEY Constraints](#)
- [Deferring Constraint Checks](#)

See Also:

- *Oracle Database Concepts* for general information about foreign key constraints
- *Oracle Database SQL Language Reference* for complete information about foreign key constraints, including restrictions

10.8.1 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 `NULL` or all non-`NULL` values 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*.

10.8.2 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 10-2](#) 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 illustrates 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.

10.8.3 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.

10.8.4 Deferring Constraint Checks

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

Note: You cannot use the `SET CONSTRAINTS` statement inside a trigger.

When deferring constraint checks:

- Select appropriate data.

You might want to defer constraint checks on `UNIQUE` and `FOREIGN` keys if the data you are working with has any of these characteristics:

- Tables are snapshots.
 - Some tables contain a large amount of data being manipulated by another application, which might not return the data in the same order.
 - Update cascade operations on foreign keys.
 - Ensure that constraints are deferrable.
- After identifying the appropriate tables, ensure that their `FOREIGN`, `UNIQUE` and `PRIMARY` key constraints are created `DEFERRABLE`.
- Within the application that manipulates the data, set all constraints deferred before you begin processing any data, as follows:

```
SET CONSTRAINTS ALL DEFERRED;
```

- (Optional) Check for constraint violations immediately before committing the transaction.

Immediately before the COMMIT statement, run the SET CONSTRAINTS ALL IMMEDIATE statement. If there are any problems with a constraint, this statement fails, and identifies the constraint that caused the error. If you commit while constraints are violated, the transaction rolls back and you get an error message.

In [Example 10-4](#), the PRIMARY and FOREIGN keys of the table emp are created DEFERRABLE and then deferred.

Example 10-4 Deferring Constraint Checks

```

DROP TABLE dept;
CREATE TABLE dept (
  deptno NUMBER PRIMARY KEY,
  dname  VARCHAR2 (30)
);

DROP TABLE emp;
CREATE TABLE emp (
  empno  NUMBER,
  ename  VARCHAR2(30),
  deptno NUMBER,
  CONSTRAINT pk_emp_empno PRIMARY KEY (empno) DEFERRABLE,
  CONSTRAINT fk_emp_deptno FOREIGN KEY (deptno) REFERENCES dept(deptno) DEFERRABLE
);

INSERT INTO dept (deptno, dname) VALUES (10, 'Accounting');
INSERT INTO dept (deptno, dname) VALUES (20, 'SALES');

INSERT INTO emp (empno, ename, deptno) VALUES (1, 'Corleone', 10);
INSERT INTO emp (empno, ename, deptno) VALUES (2, 'Costanza', 20);
COMMIT;

SET CONSTRAINTS ALL DEFERRED;

UPDATE dept
SET deptno = deptno + 10
WHERE deptno = 20;

```

Query:

```

SELECT * from dept
ORDER BY deptno;

```

Result:

```

DEPTNO DNAME
-----
10 Accounting
30 SALES

2 rows selected.

```

Update:

```

UPDATE emp
SET deptno = deptno + 10
WHERE deptno = 20;

```

Result:

```

1 row updated.

```

Query:

```
SELECT * from emp
ORDER BY deptno;
```

Result:

EMPNO	ENAME	DEPTNO
1	Corleone	10
2	Costanza	30

2 rows selected.

The `SET CONSTRAINTS` applies only to the current transaction, and its setting lasts for the duration of the transaction, or until another `SET CONSTRAINTS` statement resets the mode. The `ALTER SESSION SET CONSTRAINTS` statement applies only for the current session. The defaults specified when you create a constraint remain while the constraint exists.

See Also: *Oracle Database SQL Language Reference* for more information about the `SET CONSTRAINTS` statement

10.9 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 enforces 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 CONSTRAINT` statement.

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 nonunique indexes.

To use existing indexes when creating unique and primary key constraints, include `USING INDEX` in the `CONSTRAINT` clause. For details and examples, see *Oracle Database SQL Language Reference*.

10.10 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

10.11 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 data manipulation language (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.

10.12 When to Use CHECK Constraints

Use CHECK constraints when you must 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: [Section 10.12.4, "Choosing Between CHECK and NOT NULL Constraints"](#)

Examples of CHECK constraints include:

- 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.

10.12.1 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 these 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 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)

10.12.2 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 this 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 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*

10.12.3 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.

10.12.4 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 NULL or all non-NULL values, you must use a CHECK constraint. For example, this CHECK 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))
```

10.13 Examples of Defining Constraints

[Example 10–5](#) and [Example 10–6](#) show how to create simple constraints during the prototype phase of your database design. In these examples, each constraint is given a name. Naming the constraints prevents the database from creating multiple copies of the same constraint, with different system-generated names, if the data definition language (DDL) statement runs multiple times.

[Example 10–5](#) creates tables and their constraints at the same time, using the `CREATE TABLE` statement.

Example 10–5 Defining Constraints with the CREATE TABLE Statement

```
DROP TABLE DeptTab;
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')));

DROP TABLE EmpTab;
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 10–6](#) creates constraints for existing tables, using the `ALTER TABLE` statement.

You cannot create a validated constraint on a table if the table contains rows that violate the constraint.

Example 10–6 Defining Constraints with the ALTER TABLE Statement

```
-- Create tables without constraints:
```

```
DROP TABLE DeptTab;
CREATE TABLE DeptTab (
    Deptno  NUMBER(3),
    Dname   VARCHAR2(15),
    Loc     VARCHAR2(15)
);

DROP TABLE EmpTab;
CREATE TABLE EmpTab (
    Empno    NUMBER(5),
    Ename    VARCHAR2(15),
    Job      VARCHAR2(10),
    Mgr      NUMBER(5),
    Hiredate DATE,
    Sal      NUMBER(7,2),
```

```

    Comm      NUMBER(5,2),
    Deptno   NUMBER(3)
);

--Define constraints with the ALTER TABLE statement:

ALTER TABLE DeptTab
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);

```

See Also: *Oracle Database Administrator's Guide* for information about creating and maintaining constraints for a large production database

10.13.1 Privileges Needed to Define Constraints

If you have the CREATE TABLE or CREATE ANY TABLE system privilege, then you can define constraints on the tables that you create.

If you have the ALTER ANY TABLE system privilege, then you can define constraints on any existing table.

If you have the ALTER object privilege for a specific table, then you can define constraints on that table.

UNIQUE and PRIMARY KEY constraints require that the table owner has either the UNLIMITED TABLESPACE system privilege or a quota for the tablespace that contains the associated index.

You can define FOREIGN KEY constraints if the parent table or view is in your schema or you have the REFERENCES privilege on the columns of the referenced key in the parent table or view. For more information, see [Section 10.18.4, "Privileges Required to Create FOREIGN KEY Constraints."](#)

10.13.2 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 among the 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 Information About Constraints"](#) on page 10-25 for examples of static data dictionary views

10.14 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 might 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 Enabled Constraints \(Default\)](#)
- [Creating Disabled Constraints](#)
- [Enabling Existing Constraints](#)
- [Disabling Existing Constraints](#)
- [Guidelines for Enabling and Disabling Key Constraints](#)
- [Fixing Constraint Exceptions](#)

10.14.1 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.

10.14.2 Creating Enabled Constraints (Default)

When you define an integrity constraint (using either `CREATE TABLE` or `ALTER TABLE`), Oracle Database enables the constraint by default. For code clarity, you can explicitly enable the constraint by including the `ENABLE` clause in its definition, as in [Example 10-7](#).

Example 10-7 Creating Enabled Constraints

```
/* Use CREATE TABLE statement to create enabled constraint
   (ENABLE keyword is optional): */

DROP TABLE t1;
CREATE TABLE t1 (Empno NUMBER(5) PRIMARY KEY ENABLE);

/* Create table without constraint
```

and then use ALTER TABLE statement to add enabled constraint (ENABLE keyword is optional): */

```
DROP TABLE t2;
CREATE TABLE t2 (Empno NUMBER(5));

ALTER TABLE t2 ADD PRIMARY KEY (Empno) ENABLE;
```

Include the ENABLE clause when defining a constraint for a table to be populated a row at a time by individual transactions. This ensures that data is always consistent, and reduces the performance overhead of each DML statement.

An ALTER TABLE statement that tries to enable an integrity constraint fails if an existing row of the table violates the integrity constraint. The statement rolls back and the constraint definition is neither stored nor enabled.

See Also: [Section 10.14.7, "Fixing Constraint Exceptions,"](#) for more information about rows that violate constraints

10.14.3 Creating Disabled Constraints

You define and disable an integrity constraint (using either CREATE TABLE or ALTER TABLE), by including the DISABLE clause in its definition, as in [Example 10-8](#).

Example 10-8 Creating Disabled Constraints

```
/* Use CREATE TABLE statement to create disabled constraint */

DROP TABLE t1;
CREATE TABLE t1 (Empno NUMBER(5) PRIMARY KEY DISABLE);

/* Create table without constraint
   and then use ALTER TABLE statement to add disabled constraint */

DROP TABLE t2;
CREATE TABLE t2 (Empno NUMBER(5));

ALTER TABLE t2 ADD PRIMARY KEY (Empno) DISABLE;
```

Include the DISABLE clause when defining a constraint for a table to have large amounts of data inserted before anybody else accesses it, particularly if you must cleanse data after inserting it, or must fill empty columns with sequence numbers or parent/child relationships.

An ALTER TABLE statement that defines and disables a constraint never fails, because its rule is not enforced.

10.14.4 Enabling Existing Constraints

After you have cleansed the data and filled the empty columns, you can enable constraints that were disabled during data insertion.

To enable an existing constraint, use the ALTER TABLE statement with the ENABLE clause, as in [Example 10-9](#).

Example 10-9 Enabling Existing Constraints

-- Create table with disabled constraints:

```
DROP TABLE DeptTab;
```

```
CREATE TABLE DeptTab (
    Deptno  NUMBER(3) PRIMARY KEY DISABLE,
    Dname   VARCHAR2(15),
    Loc     VARCHAR2(15),
    CONSTRAINT uk_DeptTab_Dname_Loc UNIQUE (Dname, Loc) DISABLE,
    CONSTRAINT c_DeptTab_Loc
    CHECK (Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')) DISABLE
);

-- Enable constraints:

ALTER TABLE DeptTab
ENABLE PRIMARY KEY
ENABLE CONSTRAINT uk_DeptTab_Dname_Loc
ENABLE CONSTRAINT c_DeptTab_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: [Section 10.14.7, "Fixing Constraint Exceptions,"](#) for more information about rows that violate constraints

10.14.5 Disabling Existing Constraints

If you must perform a large insert or update when a table contains data, you can temporarily disable constraints to improve performance of the bulk operation.

To disable an existing constraint, use the ALTER TABLE statement with the DISABLE clause, as in [Example 10-10](#).

Example 10-10 Disabling Existing Constraints

-- Create table with enabled constraints:

```
DROP TABLE DeptTab;
CREATE TABLE DeptTab (
    Deptno  NUMBER(3) PRIMARY KEY ENABLE,
    Dname   VARCHAR2(15),
    Loc     VARCHAR2(15),
    CONSTRAINT uk_DeptTab_Dname_Loc UNIQUE (Dname, Loc) ENABLE,
    CONSTRAINT c_DeptTab_Loc
    CHECK (Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')) ENABLE
);

-- Disable constraints:

ALTER TABLE DeptTab
DISABLE PRIMARY KEY
DISABLE CONSTRAINT uk_DeptTab_Dname_Loc
DISABLE CONSTRAINT c_DeptTab_Loc;
```

10.14.6 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 [Section 10.18, "Managing FOREIGN KEY Constraints"](#)

10.14.7 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: [Section 10.14.7, "Fixing Constraint Exceptions,"](#) for more information about 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

10.15 Modifying Constraints

Starting with Oracle8i, you can modify an existing constraint with the MODIFY CONSTRAINT clause, as in [Example 10-11](#).

See Also: *Oracle Database SQL Language Reference* for information about the parameters you can modify

Example 10-11 Modifying Constraints

```
/* Create & then modify a CHECK constraint: */

DROP TABLE X1Tab;
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;

/* Create & then modify a PRIMARY KEY constraint: */

DROP TABLE t1;
```

```
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 ENABLE NOVALIDATE;
```

10.16 Renaming Constraints

One property of a constraint that you can modify is its name. Situations in which you would rename a constraint include:

- You want to clone a table and its constraints.
Constraint names must be unique, even across multiple schemas. Therefore, the constraints in the original table cannot have the same names as those in the cloned table.
- You created a constraint with a default system-generated name, and now you want to give it a name that is easy to remember, so that you can easily enable and disable it.

[Example 10-12](#) shows how to find the system-generated name of a constraint and change it.

Example 10-12 Renaming a Constraint

```
DROP TABLE T;
CREATE TABLE T (
  C1 NUMBER PRIMARY KEY,
  C2 NUMBER
);
```

Query:

```
SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS
WHERE TABLE_NAME = 'T'
AND CONSTRAINT_TYPE = 'P';
```

Result (system-generated name of constraint name varies):

```
CONSTRAINT_NAME
-----
SYS_C0013059
```

1 row selected.

Rename constraint from name reported in preceding query to T_C1_PK:

```
ALTER TABLE T
RENAME CONSTRAINT SYS_C0013059
TO T_C1_PK;
```

Query:

```
SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS
WHERE TABLE_NAME = 'T'
```

```
AND CONSTRAINT_TYPE = 'P';
```

Result:

```
CONSTRAINT_NAME
-----
T_C1_PK

1 row selected.
```

10.17 Dropping Constraints

You can drop a constraint using the `DROP` clause of the `ALTER TABLE` statement. Situations in which you would drop a constraint include:

- The constraint enforces a rule that is no longer true.
- The constraint is no longer needed.

To drop a constraint and all other integrity constraints that depend on it, specify `CASCADE`.

Example 10–13 Dropping Constraints

-- Create table with constraints:

```
DROP TABLE DeptTab;
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'))
);

-- Drop constraints:

ALTER TABLE DeptTab
DROP PRIMARY KEY
DROP CONSTRAINT uk_DeptTab_Dname_Loc
DROP CONSTRAINT c_DeptTab_Loc;
```

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 SQL Language Reference* for more information about the `DROP` clause of the `ALTER TABLE` statement.
- *Oracle Database Administrator's Guide* for more information about dropping constraints.
- *Oracle Database SQL Language Reference* for information about the `CASCADE CONSTRAINTS` clause of the `DROP TABLE` statement, which drops all referential integrity constraints that refer to primary and unique keys in the dropped table

10.18 Managing FOREIGN KEY Constraints

FOREIGN KEY constraints enforce relationships between columns in different tables. Therefore, they cannot be enabled if the constraint of the referenced primary or unique key is not present or not enabled.

10.18.1 Data Types and Names for Foreign Key Columns

You must use the same data type for corresponding columns in the dependent and referenced tables. The column names need not match.

10.18.2 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.

10.18.3 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.

10.18.4 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

10.18.5 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);
```

10.19 Viewing Information About 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`, as in [Example 10–14](#).

See Also: *Oracle Database Reference* for information about `*_CONSTRAINTS` and `*_CONS_COLUMNS`

Example 10–14 Viewing Information About Constraints

```
DROP TABLE DeptTab;
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'))
);

DROP TABLE EmpTab;
CREATE TABLE EmpTab (
  Empno   NUMBER(5) PRIMARY KEY,
  Ename   VARCHAR2(15) NOT NULL,
  Job     VARCHAR2(10),
  Mgr     NUMBER(5) CONSTRAINT r_EmpTab_Mgr
          REFERENCES EmpTab ON DELETE CASCADE,
  Hiredate DATE,
  Sal     NUMBER(7,2),
  Comm    NUMBER(5,2),
```

```

Deptno  NUMBER(3) NOT NULL
CONSTRAINT r_EmpTab_Deptno REFERENCES DeptTab
);

-- Format columns (optional):

COLUMN CONSTRAINT_NAME  FORMAT A20;
COLUMN CONSTRAINT_TYPE   FORMAT A4 HEADING 'TYPE';
COLUMN TABLE_NAME        FORMAT A10;
COLUMN R_CONSTRAINT_NAME FORMAT A17;
COLUMN SEARCH_CONDITION  FORMAT A40;
COLUMN COLUMN_NAME       FORMAT A12;

```

List accessible constraints in DeptTab and EmpTab:

```

SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME, R_CONSTRAINT_NAME
FROM USER_CONSTRAINTS
WHERE (TABLE_NAME = 'DEPTTAB' OR TABLE_NAME = 'EMPTAB')
ORDER BY CONSTRAINT_NAME;

```

Result:

CONSTRAINT_NAME	TYPE	TABLE_NAME	R_CONSTRAINT_NAME
C_DEPTTAB_LOC	C	DEPTTAB	
R_EMPTAB_DEPTNO	R	EMPTAB	SYS_C006286
R_EMPTAB_MGR	R	EMPTAB	SYS_C006290
SYS_C006286	P	DEPTTAB	
SYS_C006288	C	EMPTAB	
SYS_C006289	C	EMPTAB	
SYS_C006290	P	EMPTAB	
UK_DEPTTAB_DNAME_LOC	U	DEPTTAB	

8 rows selected.

Distinguish between NOT NULL and CHECK constraints in DeptTab and EmpTab:

```

SELECT CONSTRAINT_NAME, SEARCH_CONDITION
FROM USER_CONSTRAINTS
WHERE (TABLE_NAME = 'DEPTTAB' OR TABLE_NAME = 'EMPTAB')
AND CONSTRAINT_TYPE = 'C'
ORDER BY CONSTRAINT_NAME;

```

Result:

CONSTRAINT_NAME	SEARCH_CONDITION
C_DEPTTAB_LOC	Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')
SYS_C006288	"ENAME" IS NOT NULL
SYS_C006289	"DEPTNO" IS NOT NULL

3 rows selected.

For DeptTab and EmpTab, list columns that constitute constraints:

```

SELECT CONSTRAINT_NAME, TABLE_NAME, COLUMN_NAME
FROM USER_CONS_COLUMNS
WHERE (TABLE_NAME = 'DEPTTAB' OR TABLE_NAME = 'EMPTAB')
ORDER BY CONSTRAINT_NAME;

```

Result:

CONSTRAINT_NAME	TABLE_NAME	COLUMN_NAME
C_DEPTTAB_LOC	DEPTTAB	LOC
R_EMPTAB_DEPTNO	EMPTAB	DEPTNO
R_EMPTAB_MGR	EMPTAB	MGR
SYS_C006286	DEPTTAB	DEPTNO
SYS_C006288	EMPTAB	ENAME
SYS_C006289	EMPTAB	DEPTNO
SYS_C006290	EMPTAB	EMPNO
UK_DEPTTAB_DNAME_LOC	DEPTTAB	LOC
UK_DEPTTAB_DNAME_LOC	DEPTTAB	DNAME

9 rows selected.

Note that:

- Some constraint names are user specified (such as UK_DEPTTAB_DNAME_LOC), while others are system specified (such as SYS_C006290).
- Each constraint type is denoted with a different character in the CONSTRAINT_TYPE column. This 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.

These constraints are explicitly listed in the SEARCH_CONDITION column:

- NOT NULL constraints
- The conditions for user-defined CHECK constraints

Part III

PL/SQL for Application Developers

This part presents information that application developers need about PL/SQL, the Oracle procedural extension of SQL.

Chapters:

- [Chapter 11, "Coding PL/SQL Subprograms and Packages"](#)
- [Chapter 12, "Using PL/Scope"](#)
- [Chapter 13, "Using the PL/SQL Hierarchical Profiler"](#)
- [Chapter 14, "Developing PL/SQL Web Applications"](#)
- [Chapter 15, "Using Continuous Query Notification \(CQN\)"](#)

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

Coding PL/SQL Subprograms and Packages

PL/SQL subprograms and packages are the building blocks of Oracle Database applications. Oracle recommends that you implement your application as a package, for the reasons given in *Oracle Database PL/SQL Language Reference*.

Topics:

- [Overview of PL/SQL Subprograms](#)
- [Overview of PL/SQL Packages](#)
- [Overview of PL/SQL Units](#)
- [Creating PL/SQL Subprograms and Packages](#)
- [Altering PL/SQL Subprograms and Packages](#)
- [Dropping PL/SQL Subprograms and Packages](#)
- [Compiling PL/SQL Units for Native Execution](#)
- [Invoking Stored PL/SQL Subprograms](#)
- [Invoking Stored PL/SQL Functions from SQL Statements](#)
- [Debugging Stored Subprograms](#)
- [Package Invalidations and Session State](#)

See Also:

- *Oracle Database PL/SQL Language Reference* for information about handling errors in PL/SQL subprograms and packages
- *Oracle Database Data Cartridge Developer's Guide* for information about creating aggregate functions for complex data types such as multimedia data stored using object types, opaque types, and LOBs
- *Oracle Database SQL Tuning Guide* for information about application tracing tools, which can help you find problems in PL/SQL code

11.1 Overview of PL/SQL Subprograms

The basic unit of a PL/SQL source program is the **block**, which groups related declarations and statements. A block has an optional declarative part, a required executable part, and an optional exception-handling part. A block can be either anonymous or named. (For more information about PL/SQL blocks, see *Oracle Database PL/SQL Language Reference*.)

A PL/SQL **subprogram** is a named block that can be invoked repeatedly. If the subprogram has parameters, then their values can differ for each invocation. For the reasons to use subprograms, see *Oracle Database PL/SQL Language Reference*.

A subprogram is either a procedure or a function. Typically, you use a **procedure** to perform an action and a **function** to compute and return a value.

A subprogram is also either a **nested subprogram** (created inside a PL/SQL block, which can be another subprogram), a **package subprogram** (declared in a package specification and defined in the package body), or a **standalone subprogram** (created at schema level). Package subprograms and standalone programs are **stored subprograms**. A stored subprogram is compiled and stored in the database, where many applications can invoke it.

Stored subprograms are affected by the AUTHID and ACCESSIBLE BY clauses. The **AUTHID clause** affects the name resolution and privilege checking of SQL statements that the subprogram issues at runtime. The **ACCESSIBLE BY clause** specifies a white list of PL/SQL units that can access the subprogram. For information about PL/SQL units, see [Section 11.3](#).

A PL/SQL subprogram running 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. For information about external subprograms, see [Chapter 18, "Developing Applications with Multiple Programming Languages."](#)

PL/SQL lets you overload nested subprograms, package subprograms, and type methods. **Overloaded subprograms** have the same name but their formal parameters differ in either name, number, order, or data type family. For more information about overloaded subprograms, see *Oracle Database PL/SQL Language Reference*.

Like a stored procedure, a **trigger** is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is enabled, the database automatically invokes it—that is, the trigger fires—whenever its triggering event occurs. While a trigger is disabled, it does not fire. For more information about triggers, see *Oracle Database PL/SQL Language Reference*.

See Also: *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL subprograms

11.2 Overview of PL/SQL Packages

A PL/SQL **package** is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents. For the reasons to use packages, see *Oracle Database PL/SQL Language Reference*.

A package always has a **specification**, which declares the **public items** that can be referenced from outside the package. Public items can be used or invoked by external users who have the EXECUTE privilege for the package or the EXECUTE ANY PROCEDURE privilege.

If the public items include cursors or subprograms, then the package must also have a **body**. The body must define queries for public cursors and code for public subprograms. The body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package. Finally, the body can have an **initialization part**, whose statements initialize variables and do other one-time setup steps, and an exception-handling part. You can

change the body without changing the specification or the references to the public items; therefore, you can think of the package body as a black box.

In either the package specification or package body, you can map a package subprogram to an external Java or C subprogram by using a **call specification**, which maps the external subprogram name, parameter types, and return type to their SQL counterparts.

The **AUTHID clause** of the package specification determines whether the subprograms and cursors in the package run with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.

The **ACCESSIBLE BY clause** of the package specification lets you specify a white list" of PL/SQL units that can access the package. You use this clause in situations like these:

- You implement a PL/SQL application as several packages—one package that provides the application programming interface (API) and helper packages to do the work. You want clients to have access to the API, but not to the helper packages. Therefore, you omit the ACCESSIBLE BY clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package.
- You create a utility package to provide services to some, but not all, PL/SQL units in the same schema. To restrict use of the package to the intended units, you list them in the ACCESSIBLE BY clause in the package specification.

For information about PL/SQL units, see [Section 11.3](#).

Note: Before you create your own package, check *Oracle Database PL/SQL Packages and Types Reference* to see if Oracle supplies a package with the functionality that you need.

See Also: *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL packages

11.3 Overview of PL/SQL Units

A PL/SQL unit is one of these:

- PL/SQL anonymous block
- FUNCTION
- LIBRARY
- PACKAGE
- PACKAGE BODY
- PROCEDURE
- TRIGGER
- TYPE
- TYPE BODY

PL/SQL units are affected by PL/SQL compilation parameters (a category of database initialization parameters). Different PL/SQL units—for example, a package specification and its body—can have different compilation parameter settings. For

more information about PL/SQL units and compilation parameters, see *Oracle Database PL/SQL Language Reference*.

The AUTHID property of a PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at runtime. For details, see *Oracle Database PL/SQL Language Reference*.

11.3.1 PLSQL_OPTIMIZE_LEVEL Compilation Parameter

The PL/SQL optimize level determines how much the PL/SQL optimizer can rearrange code for better performance. This level is set with the compilation parameter PLSQL_OPTIMIZE_LEVEL (whose default value is 2).

To change the PL/SQL optimize level for your session, use the SQL command ALTER SESSION. Changing the level for your session affects only subsequently created PL/SQL units. To change the level for an existing PL/SQL unit, use an ALTER command with the COMPILE clause.

To display the current value of PLSQL_OPTIMIZE_LEVEL for one or more PL/SQL units, use the static data dictionary view ALL_PLSQL_OBJECT_SETTINGS.

[Example 11-1](#) creates two procedures, displays their optimize levels, changes the optimize level for the session, creates a third procedure, and displays the optimize levels of all three procedures. Only the third procedure has the new optimize level. Then the example changes the optimize level for only one procedure and displays the optimize levels of all three procedures again.

Example 11-1 Changing PLSQL_OPTIMIZE_LEVEL

Create two procedures:

```
CREATE OR REPLACE PROCEDURE p1 AUTHID DEFINER AS
BEGIN
  NULL;
END;
/
CREATE OR REPLACE PROCEDURE p2 AUTHID DEFINER AS
BEGIN
  NULL;
END;
/
```

Display the optimization levels of the two procedures:

```
SELECT NAME, PLSQL_OPTIMIZE_LEVEL
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME LIKE 'P%' AND TYPE='PROCEDURE'
ORDER BY NAME;
```

Result:

NAME	PLSQL_OPTIMIZE_LEVEL
P1	2
P2	2

2 rows selected.

Change the optimization level for the session and create a third procedure:

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL=1;
```

```
CREATE OR REPLACE PROCEDURE p3 AUTHID DEFINER AS
BEGIN
  NULL;
END;
/
```

Display the optimization levels of the three procedures:

```
SELECT NAME, PLSQL_OPTIMIZE_LEVEL
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME LIKE 'P%' AND TYPE='PROCEDURE'
ORDER BY NAME;
```

Result:

NAME	PLSQL_OPTIMIZE_LEVEL
P1	2
P2	2
P3	1

3 rows selected.

Change the optimization level of procedure p1 to 3:

```
ALTER PROCEDURE p1 COMPILE PLSQL_OPTIMIZE_LEVEL=3;
```

Display the optimization levels of the three procedures:

```
SELECT NAME, PLSQL_OPTIMIZE_LEVEL
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME LIKE 'P%' AND TYPE='PROCEDURE'
ORDER BY NAME;
```

Result:

NAME	PLSQL_OPTIMIZE_LEVEL
P1	3
P2	2
P3	1

3 rows selected.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about the PL/SQL optimizer
- *Oracle Database Reference* for more information about PLSQL_OPTIMIZE_LEVEL
- *Oracle Database SQL Language Reference* for more information about ALTER SESSION
- *Oracle Database PL/SQL Language Reference* for information about the ALTER commands for PL/SQL units
- *Oracle Database Reference* for more information about ALL_PLSQL_OBJECT_SETTINGS

11.4 Creating PL/SQL Subprograms and Packages

Topics:

- [Privileges Needed to Create Subprograms and Packages](#)
- [Creating Subprograms and Packages](#)
- [PL/SQL Object Size Limits](#)
- [PL/SQL Data Types](#)
- [Returning Result Sets to Clients](#)
- [Returning Large Amounts of Data from a Function](#)
- [PL/SQL Function Result Cache](#)
- [Overview of Bulk Binding](#)
- [PL/SQL Dynamic SQL](#)

11.4.1 Privileges Needed to Create Subprograms and Packages

To create a standalone subprogram or package in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create a standalone subprogram or package in another schema, you must have the `CREATE ANY PROCEDURE` system privilege.

If the subprogram or package that you create references schema objects, then you must have the necessary object privileges for those objects. These privileges must be granted to you explicitly, not through roles.

If the privileges of the owner of a subprogram or package change, then the subprogram or package must be reauthenticated before it is run. If a necessary object privilege for a referenced object is revoked from the owner of the subprogram or package, then the subprogram cannot run.

Granting the `EXECUTE` privilege on a subprogram lets users run that subprogram under the security domain of the subprogram owner, so that the user need not be granted privileges to the objects that the subprogram references. The `EXECUTE` privilege allows more disciplined and efficient security strategies for database applications and their users. Furthermore, it allows subprograms and packages to be stored in the data dictionary (in the `SYSTEM` tablespace), where no quota controls the amount of space available to a user who creates subprograms and packages.

See Also:

- [Oracle Database SQL Language Reference](#) for information about system and object privileges
- [Section 11.8, "Invoking Stored PL/SQL Subprograms"](#)

11.4.2 Creating Subprograms and Packages

This topic explains how to create standalone subprograms and packages, using SQL Data Definition Language (DDL) statements.

The DDL statements for creating standalone subprograms and packages are:

- `CREATE FUNCTION`, described in [Oracle Database PL/SQL Language Reference](#)
- `CREATE PROCEDURE`, described in [Oracle Database PL/SQL Language Reference](#)
- `CREATE PACKAGE`, described in [Oracle Database PL/SQL Language Reference](#)

- CREATE PACKAGE BODY, described in *Oracle Database PL/SQL Language Reference*

The name of a package and the names of its public objects must be unique within the package schema. The package specification and body must have the same name. Package constructs must have unique names within the scope of the package, except for overloaded subprograms.

Each of the preceding CREATE statements has an optional OR REPLACE clause. Specify OR REPLACE to re-create an existing PL/SQL unit—that is, to change its declaration or definition without dropping it, re-creating it, and regranting object privileges previously granted on it. If you redefine a PL/SQL unit, the database recompiles it.

Caution: A CREATE OR REPLACE statement does not issue a warning before replacing the existing PL/SQL unit.

Using any text editor, create a text file that contains DDL statements for creating any number of subprograms and packages.

To run the DDL statements, use an interactive tool such as SQL*Plus. The SQL*Plus command START or @ runs a script. For example, this SQL*Plus command runs the script `my_app.sql`:

```
@my_app
```

(For information about running scripts in SQL*Plus, see *SQL*Plus User's Guide and Reference*.)

Alternatively, you can create and run the DDL statements using SQL Developer. For information about SQL Developer, see *Oracle SQL Developer User's Guide*.

11.4.3 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 PARSED_SIZE, a column in the static data dictionary view `*_OBJECT_SIZE`. The column PARSED_SIZE 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.

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL program limits and PARSED_SIZE
- *Oracle Database Reference* for information about `*_OBJECT_SIZE`

11.4.4 PL/SQL Data Types

This topic introduces the PL/SQL data types and refers to other chapters or documents for more information.

Use the correct and most specific PL/SQL data type for each PL/SQL variable in your database application. For the reasons, see [Section 7.1, "Using the Correct and Most Specific Data Type."](#)

Topics:

- [PL/SQL Scalar Data Types](#)
- [PL/SQL Composite Data Types](#)
- [Abstract Data Types](#)

11.4.4.1 PL/SQL Scalar Data Types

Scalar data types store values that have no internal components.

A scalar data type can have subtypes. A **subtype** is a data type that is a subset of another data type, which is its **base type**. A subtype has the same valid operations as its base type. A data type and its subtypes comprise a **data type family**.

PL/SQL predefines many types and subtypes in the package STANDARD and lets you define your own subtypes.

Topics:

- [SQL Data Types](#)
- [BOOLEAN Data Type](#)
- [PLS_INTEGER and BINARY_INTEGER Data Types](#)
- [REF CURSOR Data Type](#)
- [User-Defined PL/SQL Subtypes](#)

See Also:

- [Oracle Database PL/SQL Language Reference](#) for a complete description of scalar PL/SQL data types
- [Oracle Database PL/SQL Language Reference](#) for the predefined PL/SQL data types and subtypes, grouped by data type family

11.4.4.1.1 SQL Data Types The PL/SQL data types include the SQL data types. For general information about the SQL data types, see *Oracle Database SQL Language Reference*—all information there about data types and subtypes, data type comparison rules, data conversion, literals, and format models applies to both SQL and PL/SQL, except as noted in *Oracle Database PL/SQL Language Reference*.

For information about how to use the SQL data types in database applications, see [Chapter 7, "Using SQL Data Types in Database Applications."](#)

11.4.4.1.2 BOOLEAN Data Type The BOOLEAN data type stores **logical values**, which are the Boolean values TRUE and FALSE and the value NULL. NULL represents an unknown value. For more information about the BOOLEAN data type, see *Oracle Database PL/SQL Language Reference*.

11.4.4.1.3 PLS_INTEGER and BINARY_INTEGER Data Types The PL/SQL data types PLS_INTEGER and BINARY_INTEGER are identical. For simplicity, this guide uses PLS_INTEGER to mean both PLS_INTEGER and BINARY_INTEGER.

The PLS_INTEGER data type stores signed integers in the range -2,147,483,648 through 2,147,483,647, represented in 32 bits.

The PLS_INTEGER data type has these advantages over the NUMBER data type and NUMBER subtypes:

- PLS_INTEGER values require less storage.

- PLS_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic.

For efficiency, use PLS_INTEGER values for all calculations in its range.

For more information about the PLS_INTEGER data type, see *Oracle Database PL/SQL Language Reference*.

11.4.4.1.4 REF CURSOR Data Type

REF CURSOR is the data type of a cursor variable. A cursor variable is like an explicit cursor, except that:

- It is not limited to one query.
You can open a cursor variable for a query, process the result set, and then use the cursor variable for another query.
- You can assign a value to it.
- You can use it in an expression.
- It can be a subprogram parameter.
You can use cursor variables to pass query result sets between subprograms.
- It can be a host variable.
You can use cursor variables to pass query result sets between PL/SQL stored subprograms and their clients.
- It cannot accept parameters.
You cannot pass parameters to a cursor variable, but you can pass whole queries to it.

A cursor variable has this flexibility because it is a pointer; that is, its value is the address of an item, not the item itself.

For more information about the REF CURSOR data type and cursor variables, see *Oracle Database PL/SQL Language Reference*.

11.4.4.1.5 User-Defined PL/SQL Subtypes

PL/SQL lets you define your own subtypes. The base type can be any scalar PL/SQL type, including a previously defined user-defined subtype.

Subtypes can:

- Provide compatibility with ANSI/ISO data types
- Show the intended use of data items of that type
- Detect out-of-range values

For more information about user-defined PL/SQL subtypes, see *Oracle Database PL/SQL Language Reference*.

11.4.4.2 PL/SQL Composite Data Types

Composite data types have internal components. The PL/SQL composite data types are collections and records.

In a **collection**, the internal components always have the same data type, and are called **elements**. You can access each element of a collection variable by its unique index. PL/SQL has three collection types—associative array, VARRAY (variable-size array), and nested table.

In a **record**, the internal components can have different data types, and are called **fields**. You can access each field of a record variable by its name.

You can create a collection of records, and a record that contains collections.

For more information about PL/SQL composite data types, see *Oracle Database PL/SQL Language Reference*.

11.4.4.3 Abstract Data Types

An Abstract Data Type (ADT) consists of a data structure and subprograms that manipulate the data. In the static data dictionary view `*_OBJECTS`, the `OBJECT_TYPE` of an ADT is `TYPE`. In the static data dictionary view `*_TYPES`, the `TYPECODE` of an ADT is `OBJECT`.

For more information about ADTs, see *Oracle Database PL/SQL Language Reference*.

11.4.5 Returning Result Sets to Clients

In PL/SQL, as in traditional database programming, you use cursors to process query result sets. A **cursor** is a pointer to a private SQL area that stores information about processing a specific `SELECT` or DML statement.

Note: The cursors that this section discusses are session cursors. A **session cursor** lives in session memory until the session ends, when it ceases to exist. Session cursors are different from the cursors in the private SQL area of the program global area (PGA), which are explained in *Oracle Database Concepts*.

A cursor that is constructed and managed by PL/SQL is an **implicit cursor**. A cursor that you construct and manage is an **explicit cursor**. The only advantage of an explicit cursor over an implicit cursor is that with an explicit cursor, you can limit the number of fetched rows (for details, see *Oracle Database PL/SQL Language Reference*). For general information about cursors, see *Oracle Database PL/SQL Language Reference*. For information about query set result processing with implicit and explicit cursors, see *Oracle Database PL/SQL Language Reference*.

A **cursor variable** is a pointer to a cursor. That is, its value is the address of a cursor, not the cursor itself. Therefore, a cursor variable has more flexibility than an explicit cursor. However, a cursor variable also has costs that an explicit cursor does not.

Topics:

- [Advantages of Cursor Variables](#)
- [Disadvantages of Cursor Variables](#)
- [Returning Query Results Implicitly](#)

See Also:

- *Oracle Database PL/SQL Language Reference* for general information about cursor variables
- *Oracle Call Interface Programmer's Guide* for information about using cursor variables in OCI
- *Pro*C/C++ Programmer's Guide* for information about using cursor variables in Pro*C/C++
- *Pro*COBOL Programmer's Guide* for information about using cursor variables in Pro*COBOL
- *Oracle Database JDBC Developer's Guide* for information about using cursor variables in JDBC
- [Section 11.4.6, "Returning Large Amounts of Data from a Function"](#)

11.4.5.1 Advantages of Cursor Variables

A cursor variable is like an explicit cursor except that:

- It is not limited to one query.
You can open a cursor variable for a query, process the result set, and then use the cursor variable for another query.
- You can assign a value to it.
- You can use it in an expression.
- It can be a subprogram parameter.
You can use cursor variables to pass query result sets between subprograms.
- It can be a host variable.
You can use cursor variables to pass query result sets between PL/SQL stored subprograms and their clients.
- It cannot accept parameters.
You cannot pass parameters to a cursor variable, but you can pass whole queries to it. The queries can include variables.

The preceding characteristics give cursor variables these advantages:

- Encapsulation
Queries are centralized in the stored subprogram that opens the cursor variable.
- Easy maintenance
If you must change the cursor, then you must change only the stored subprogram, not every application that invokes the stored subprogram.
- Convenient security
The application connects to the server with the user name of the application user. The application user must have EXECUTE permission on the stored subprogram that opens the cursor, but need not have READ permission on the queried tables.

11.4.5.2 Disadvantages of Cursor Variables

If you need not use a cursor variable, then use an implicit or explicit cursor, for both better performance and ease of programming.

Topics:

- [Parsing Penalty for Cursor Variable](#)
- [Multiple-Row-Fetching Penalty for Cursor Variable](#)

Note: The examples in these topics include TKPROF reports. For instructions for producing TKPROF reports, see *Oracle Database SQL Tuning Guide*.

11.4.5.2.1 Parsing Penalty for Cursor Variable When you close an explicit cursor, the cursor closes from your perspective—that is, you cannot use it where an open cursor is required—but PL/SQL caches the explicit cursor in an open state. If you reexecute the statement associated with the cursor, then PL/SQL uses the cached cursor, thereby avoiding a parse.

Avoiding a parse can significantly reduce CPU use, and the caching of explicit cursors is transparent to you; it does not affect your programming. PL/SQL does not reduce your supply of available open cursors. If your program must open another cursor but doing so would exceed the init.ora setting of OPEN_CURATORS, then PL/SQL closes cached cursors.

PL/SQL cannot cache a cursor variable in an open state. Therefore, a cursor variable has a parsing penalty.

In [Example 11–2](#), the procedure opens, fetches from, and closes an explicit cursor and then does the same with a cursor variable. The anonymous block calls the procedure 10 times. The TKPROF report shows that both queries were run 10 times, but the query associated with the explicit cursor was parsed only once, while the query associated with the cursor variable was parsed 10 times.

Example 11–2 Parsing Penalty for Cursor Variable

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  CURSOR e_c IS SELECT * FROM DUAL d1;  -- explicit cursor
  c_v SYS_REFCURSOR;                   -- cursor variable
  rec DUAL%ROWTYPE;
BEGIN
  OPEN e_c;                           -- explicit cursor
  FETCH e_c INTO rec;
  CLOSE e_c;

  OPEN c_v FOR SELECT * FROM DUAL d2; -- cursor variable
  FETCH c_v INTO rec;
  CLOSE c_v;
END;
/
BEGIN
  FOR i IN 1..10 LOOP                -- execute p 10 times
    p;
  END LOOP;
```

TKPROF report is similar to:

```
SELECT * FROM DUAL D1;
```

call	count
Parse	1
Execute	10

```

Fetch      10
-----
total      21
*****
SELECT * FROM DUAL D2;

call      count
-----
Parse     10
Execute   10
Fetch     10
-----
total     30

```

11.4.5.2.2 Multiple-Row-Fetching Penalty for Cursor Variable [Example 11–3](#) creates a table that has more than 7,000 rows and fetches all of those rows twice, first with an implicit cursor (fetching arrays) and then with a cursor variable (fetching individual rows). The code for the implicit cursor is simpler than the code for the cursor variable, and the TKPROF report shows that it also performs better.

Although you could use the cursor variable to fetch arrays, you would need much more code. Specifically, you would need code to do the following:

- Define the types of the collections into which you will fetch the arrays
- Explicitly bulk collect into the collections
- Loop through the collections to process the fetched data
- Close the explicitly opened cursor variable

Example 11–3 Array Fetching Penalty for Cursor Variable

Create table to query and display its number of rows:

```

CREATE TABLE t AS
  SELECT * FROM ALL_OBJECTS;

SELECT COUNT(*) FROM t;

```

Result is similar to:

```

COUNT(*)
-----
70788

```

Perform equivalent operations with an implicit cursor and a cursor variable:

```

DECLARE
  c_v SYS_REFCURSOR;
  rec t%ROWTYPE;
BEGIN
  FOR x IN (SELECT * FROM t exp_cur) LOOP -- implicit cursor
    NULL;
  END LOOP;

  OPEN c_v FOR SELECT * FROM t cur_var; -- cursor variable

  LOOP
    FETCH c_v INTO rec;
    EXIT WHEN c_v%NOTFOUND;
  END LOOP;

```

```

CLOSE c_v;
END;
/

```

TKPROF report is similar to:

```

SELECT * FROM T EXP_CUR

call      count      cpu      elapsed      disk      query      current      rows
-----  -----  -----  -----  -----  -----  -----  -----
Parse      1      0.00      0.00      0      0      0      0
Execute     1      0.00      0.00      0      0      0      0
Fetch    722      0.23      0.23      0    1748      0  72198
-----  -----  -----  -----  -----  -----  -----  -----
total    724      0.23      0.23      0    1748      0  72198
*****
```

```

SELECT * FROM T CUR_VAR

call      count      cpu      elapsed      disk      query      current      rows
-----  -----  -----  -----  -----  -----  -----  -----
Parse      1      0.00      0.00      0      0      0      0
Execute     1      0.00      0.00      0      0      0      0
Fetch  72199      0.40      0.42      0    72203      0  72198
-----  -----  -----  -----  -----  -----  -----  -----
total   72201      0.40      0.42      0    72203      0  72198

```

11.4.5.3 Returning Query Results Implicitly

A stored subprogram can return a query result implicitly to either the client program or the subprogram's immediate caller by invoking the `DBMS_SQL.RETURN_RESULT` procedure. After `DBMS_SQL.RETURN_RESULT` returns the result, only the recipient can access it. For details, see *Oracle Database PL/SQL Language Reference*.

Note: To return implicitly the result of a query executed with dynamic SQL, the subprogram must execute the query with `DBMS_SQL` procedures, not the `EXECUTE IMMEDIATE` statement. The reason is that the cursors that the `EXECUTE IMMEDIATE` statement returns to the subprogram are closed when the `EXECUTE IMMEDIATE` statement completes. For information about using `DBMS_SQL` procedures for dynamic SQL, see *Oracle Database PL/SQL Language Reference*.

11.4.6 Returning Large Amounts of Data from a Function

In a data warehousing environment, you might use PL/SQL functions to transform large amounts of data. You might pass the data 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.

See Also: *Oracle Database PL/SQL Language Reference* for more information about performing multiple transformations with pipelined table functions

11.4.7 PL/SQL Function Result Cache

Using the PL/SQL 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 function result cache, see *Oracle Database PL/SQL Language Reference*.

11.4.8 Overview of Bulk Binding

Oracle Database uses two engines to run PL/SQL units. The PL/SQL engine runs the procedural statements and the SQL engine runs the SQL statements. Every SQL statement causes a context switch between the two engines. You can greatly improve the performance of your database application by minimizing the number of context switches for each PL/SQL unit.

When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required can cause poor performance. Collections include:

- Associative arrays
- Variable-size arrays
- Nested 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 between the two engines in a single operation.

Typically, bulk binding 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 binding. Consider using bulk binding to improve the performance of DML and `SELECT INTO` statements that reference collections and `FOR` loops that reference collections and return DML.

Note: Parallel DML statements are disabled with bulk binding. For information about parallel DML statements, see *Oracle Database PL/SQL Language Reference*.

Topics:

- [DML Statements that Reference Collections](#)
- [SELECT Statements that Reference Collections](#)
- [FOR Loops that Reference Collections and Return DML](#)

See Also: *Oracle Database PL/SQL Language Reference* for more information about bulk binding, including how to handle exceptions that occur during bulk binding operations

11.4.8.1 DML Statements that Reference Collections

A bulk bind, which uses the FORALL keyword, can improve the performance of INSERT, UPDATE, or DELETE statements that reference collection elements.

The PL/SQL block in [Example 11-4](#) increases the salary for employees whose manager's ID number is 7902, 7698, or 7839, with and without bulk binds. Without bulk bind, PL/SQL sends a SQL statement to the SQL engine for each updated employee, leading to context switches that slow performance.

Example 11-4 DML Statements that Reference Collections

```
DECLARE
  TYPE numlist IS VARRAY (100) OF NUMBER;
  id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN
  -- Efficient method, using bulk bind:

  FORALL i IN id.FIRST..id.LAST
    UPDATE EMPLOYEES
      SET SALARY = 1.1 * SALARY
        WHERE MANAGER_ID = id(i);

  -- Slower method:

  FOR i IN id.FIRST..id.LAST LOOP
    UPDATE EMPLOYEES
      SET SALARY = 1.1 * SALARY
        WHERE MANAGER_ID = id(i);
  END LOOP;
END;
/
```

See Also: *Oracle Database PL/SQL Language Reference* for more information about the FORALL statement

11.4.8.2 SELECT Statements that Reference Collections

The BULK COLLECT clause can improve the performance of queries that reference collections. You can use BULK COLLECT with tables of scalar values, or tables of %TYPE values.

The PL/SQL block in [Example 11-5](#) queries multiple values into PL/SQL tables, with and without bulk binds. Without bulk bind, PL/SQL sends a SQL statement to the SQL engine for each selected employee, leading to context switches that slow performance.

Example 11-5 SELECT Statements that Reference Collections

```
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 EMPLOYEE_ID, LAST_NAME
  FROM EMPLOYEES
  WHERE MANAGER_ID = 7698;
BEGIN
  -- Efficient method, using bulk bind:

  SELECT EMPLOYEE_ID, LAST_NAME BULK COLLECT
  INTO empno, ename
  FROM EMPLOYEES
  WHERE MANAGER_ID = 7698;

  -- Slower method:

  counter := 1;

  FOR rec IN c LOOP
    empno(counter) := rec.EMPLOYEE_ID;
    ename(counter) := rec.LAST_NAME;
    counter := counter + 1;
  END LOOP;
END;
/

```

See Also: *Oracle Database PL/SQL Language Reference* for more information about the BULK COLLECT clause

11.4.8.3 FOR Loops that Reference Collections and Return DML

You can use the FORALL keyword with the BULK COLLECT keywords to improve the performance of FOR loops that reference collections and return DML.

The PL/SQL block in [Example 11–6](#) updates the EMPLOYEES table by computing bonuses for a collection of employees. Then it returns the bonuses in a column called bonus_list_inst. The actions are performed with and without bulk binds. Without bulk bind, PL/SQL sends a SQL statement to the SQL engine for each updated employee, leading to context switches that slow performance.

Example 11–6 FOR Loops that Reference Collections and Return DML

```

DECLARE
  TYPE emp_list IS VARRAY(100) OF EMPLOYEES.EMPLOYEE_ID%TYPE;
  empids emp_list := emp_list(182, 187, 193, 200, 204, 206);

  TYPE bonus_list IS TABLE OF EMPLOYEES.SALARY%TYPE;
  bonus_list_inst bonus_list;

BEGIN
  -- Efficient method, using bulk bind:

  FORALL i IN empids.FIRST..empids.LAST
  UPDATE EMPLOYEES
  SET SALARY = 0.1 * SALARY
  WHERE EMPLOYEE_ID = empids(i)
  RETURNING SALARY BULK COLLECT INTO bonus_list_inst;

  -- Slower method:

  FOR i IN empids.FIRST..empids.LAST LOOP

```

```

UPDATE EMPLOYEES
SET SALARY = 0.1 * SALARY
WHERE EMPLOYEE_ID = empids(i)
RETURNING SALARY INTO bonus_list_inst(i);
END LOOP;
END;
/

```

See Also: *Oracle Database PL/SQL Language Reference* for more information about use the BULK COLLECT clause with the RETURNING INTO clause

11.4.9 PL/SQL Dynamic SQL

Dynamic SQL is a programming methodology for generating and running SQL statements at runtime. It is useful when writing general-purpose and flexible programs like dynamic query systems, when writing programs that must run database definition language (DDL) statements, or when you do not know at compile time the full text of a SQL statement or the number or data types of its input and output variables.

If you do not need dynamic SQL, then use static SQL, which has these advantages:

- Successful compilation verifies that static SQL statements reference valid database objects and that the necessary privileges are in place to access those objects.
- Successful compilation creates schema object dependencies.

For information about schema object dependency, see [Chapter 23, "Understanding Schema Object Dependency."](#)

For information about dynamic SQL, see *Oracle Database PL/SQL Language Reference*.

For information about static SQL, see *Oracle Database PL/SQL Language Reference*.

11.5 Altering PL/SQL Subprograms and Packages

To alter the name of a stored standalone subprogram or package, you must drop it (as instructed in [Section 11.6](#)) and then create it with the new name. For example:

```

CREATE PROCEDURE p IS BEGIN NULL; END;
/
DROP PROCEDURE p
/
CREATE PROCEDURE p1 IS BEGIN NULL; END;
/

```

To alter a stored standalone subprogram or package without changing its name, you can replace it with a new version with the same name by including OR REPLACE in the CREATE statement. For example:

```

CREATE OR REPLACE PROCEDURE p1 IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, world!');
END;
/

```

Note: ALTER statements (such as ALTER FUNCTION, ALTER PROCEDURE, and ALTER PACKAGE) do not alter the declarations or definitions of existing PL/SQL units, they recompile only the units. For information about ALTER statements, see *Oracle Database PL/SQL Language Reference*.

11.6 Dropping PL/SQL Subprograms and Packages

To drop stored standalone subprograms, use these statements:

- DROP FUNCTION, described in *Oracle Database PL/SQL Language Reference*
- DROP PROCEDURE, described in *Oracle Database PL/SQL Language Reference*

To drop a package (specification and body) or only its body, use the statement DROP PACKAGE, described in *Oracle Database PL/SQL Language Reference*.

11.7 Compiling PL/SQL Units for Native Execution

You can usually speed up PL/SQL units—your own and those that Oracle supplies—by compiling them into native code (processor-dependent system code), which is stored in the SYSTEM tablespace.

PL/SQL units compiled into native code run in all server environments, including the shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC).

Whether to compile a PL/SQL unit into native code depends on where you are in the development cycle and what the PL/SQL unit does.

For more information about compiling PL/SQL units for native execution, see *Oracle Database PL/SQL Language Reference*.

Note: To compile Java packages and classes for native execution, use the ncomp tool. For more information, see *Oracle Database Java Developer's Guide*.

11.8 Invoking Stored PL/SQL Subprograms

Stored PL/SQL subprograms can be invoked from many different environments. For example:

- Interactively, using an Oracle Database tool
- From the body of another subprogram
- From the body of a trigger
- From within an application (such as a SQL*Forms or a precompiler)

Stored PL/SQL functions (but not procedures) can also be invoked from within SQL statements. For details, see [Section 11.9](#).

When you invoke a subprogram owned by another user:

- You must include the name of the owner in the invocation. For example:

```
EXECUTE jdoe.Fire_emp (1043);
EXECUTE jdoe.Hire_fire.Fire_emp (1043);
```

- The AUTHID property of the subprogram affects the name resolution and privilege checking of SQL statements that the subprogram issues at runtime. For details, see *Oracle Database PL/SQL Language Reference*.

Topics:

- [Privileges Required to Invoke a Stored Subprogram](#)
- [Invoking a Subprogram Interactively from Oracle Tools](#)
- [Invoking a Subprogram from Another Subprogram](#)
- [Invoking a Remote Subprogram](#)

See Also:

- *Oracle Database PL/SQL Language Reference* for information about subprogram invocation, parameters, and definer's and invoker's rights
- *Oracle Database PL/SQL Language Reference* for information about coding the body of a trigger
- [Section 11.9, "Invoking Stored PL/SQL Functions from SQL Statements"](#)
- *Oracle Call Interface Programmer's Guide* for information about invoking PL/SQL subprograms from OCI applications
- *Pro*C/C++ Programmer's Guide* for information about invoking PL/SQL subprograms from Pro*C/C++
- *Pro*COBOL Programmer's Guide* for information about invoking PL/SQL subprograms from Pro*COBOL
- *Oracle Database JDBC Developer's Guide* for information about invoking PL/SQL subprograms from JDBC applications

11.8.1 Privileges Required to Invoke a Stored Subprogram

You do not need privileges to invoke:

- Standalone subprograms that you own
- Subprograms in packages that you own
- Public standalone subprograms
- Subprograms in public packages

To invoke a stored subprogram owned by another user, you must have the EXECUTE privilege for the standalone subprogram or for the package containing the package subprogram, or you must have the EXECUTE ANY PROCEDURE system privilege. If the subprogram is remote, then you must be granted the EXECUTE privilege or EXECUTE ANY PROCEDURE system privilege directly, not through a role.

See Also: *Oracle Database SQL Language Reference* for information about system and object privileges

11.8.2 Invoking a Subprogram Interactively from Oracle Tools

You can invoke a subprogram interactively from an Oracle Database tool, such as SQL*Plus. [Example 11-7](#) uses SQL*Plus to create a procedure and then invokes it in two different ways.

Example 11–7 Invoking a Subprogram Interactively with SQL*Plus

```
CREATE OR REPLACE PROCEDURE salary_raise (
    employee  EMPLOYEES.EMPLOYEE_ID%TYPE,
    increase   EMPLOYEES.SALARY%TYPE
)
IS
BEGIN
    UPDATE EMPLOYEES
    SET SALARY = SALARY + increase
    WHERE EMPLOYEE_ID = employee;
END;
/
```

Invoke procedure from within anonymous block:

```
BEGIN
    salary_raise(205, 200);
END;
/
```

Result:

PL/SQL procedure successfully completed.

Invoke procedure with EXECUTE statement:

```
EXECUTE salary_raise(205, 200);
```

Result:

PL/SQL procedure successfully completed.

Some interactive tools allow you to create session variables, which you can use for the duration of the session. Using SQL*Plus, [Example 11–8](#) creates, uses, and prints a session variable.

Example 11–8 Creating and Using a Session Variable with SQL*Plus

-- Create function for later use:

```
CREATE OR REPLACE FUNCTION get_job_id (
    emp_id  EMPLOYEES.EMPLOYEE_ID%TYPE
) RETURN EMPLOYEES.JOB_ID%TYPE
IS
    job_id  EMPLOYEES.JOB_ID%TYPE;
BEGIN
    SELECT JOB_ID INTO job_id
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;

    RETURN job_id;
END;
/
-- Create session variable:

VARIABLE job VARCHAR2(10);

-- Run function and store returned value in session variable:

EXECUTE :job := get_job_id(204);
```

PL/SQL procedure successfully completed.

SQL*Plus command:

```
PRINT job;
```

Result:

```
JOB
-----
PR REP
```

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

11.8.3 Invoking a Subprogram from Another Subprogram

A subprogram or a trigger can invoke another stored subprogram. In [Example 11–9](#), the procedure `print_mgr_name` invokes the procedure `print_emp_name`.

Recursive subprogram invocations are allowed (that is, a subprogram can invoke itself).

Example 11–9 Invoking a Subprogram from Within Another Subprogram

-- Create procedure that takes employee's ID and prints employee's name:

```
CREATE OR REPLACE PROCEDURE print_emp_name (
  emp_id  EMPLOYEES.EMPLOYEE_ID%TYPE
)
IS
  fname  EMPLOYEES.FIRST_NAME%TYPE;
  lname  EMPLOYEES.LAST_NAME%TYPE;
BEGIN
  SELECT FIRST_NAME, LAST_NAME
  INTO fname, lname
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  DBMS_OUTPUT.PUT_LINE (
    'Employee #' || emp_id || ': ' || fname || ' ' || lname
  );
END;
/
```

-- Create procedure that takes employee's ID and prints manager's name:

```
CREATE OR REPLACE PROCEDURE print_mgr_name (
  emp_id  EMPLOYEES.EMPLOYEE_ID%TYPE
)
IS
  mgr_id  EMPLOYEES.MANAGER_ID%TYPE;
BEGIN
  SELECT MANAGER_ID
  INTO mgr_id
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;
```

```

DBMS_OUTPUT.PUT_LINE (
  'Manager of employee #' || emp_id || ' is: '
);

print_emp_name(mgr_id);
END;
/

```

Invoke procedures:

```

BEGIN
  print_emp_name(200);
  print_mgr_name(200);
END;
/

```

Result:

```

Employee #200:  Jennifer Whalen
Manager of employee #200 is:
Employee #101:  Neena Kochhar

```

11.8.4 Invoking a Remote Subprogram

A **remote subprogram** is stored on a different database from its invoker. A remote subprogram invocation must include the subprogram name, a database link to the database on which the subprogram is stored, and an actual parameter for every formal parameter (even if the formal parameter has a default value).

For example, this SQL*Plus statement invokes the stored standalone procedure `fire_emp1`, which is referenced by the local database link named `boston_server`:

```
EXECUTE fire_emp1@boston_server(1043);
```

Note: Although you can invoke remote package subprograms, you cannot directly access remote package variables and constants.

Caution:

- Remote subprogram invocations use runtime binding. The user account to which you connect depends on the database link. (Stored subprograms use compile-time binding.)
 - If a local subprogram invokes a remote subprogram, and a time-stamp mismatch is found during execution of the local subprogram, then the remote subprogram is not run, and the local subprogram is invalidated. For more information, see [Section 23.9.1, "Dependencies Among Local and Remote Database Procedures."](#)
-

Topics:

- [Synonyms for Remote Subprograms](#)
- [Transactions That Invoke Remote Subprograms](#)

See Also:

- [Section 23.9.1, "Dependencies Among Local and Remote Database Procedures"](#)
- *Oracle Database PL/SQL Language Reference* for information about handling errors in subprograms

11.8.4.1 Synonyms for Remote Subprograms

A **synonym** is an alias for a schema object. You can create a synonym for a remote subprogram name and database link, and then use the synonym to invoke the subprogram. For example:

```
CREATE SYNONYM synonym1 FOR fire_emp1@boston_server;
EXECUTE synonym1(1043);
```

Note: You cannot create a synonym for a package subprogram, because it is not a schema object (its package is a schema object).

Synonyms provide both data independence and location transparency. Using the synonym, a user can invoke the subprogram without knowing who owns it or where it is. However, a synonym is not a substitute for privileges—to use the synonym to invoke the subprogram, the user still needs the necessary privileges for the subprogram.

Granting a privilege on a synonym is equivalent to granting the privilege on the base object. Similarly, granting a privilege on a base object is equivalent to granting the privilege on all synonyms for the object.

You can create both private and public synonyms. A private synonym is in your schema and you control its availability to others. A public synonym belongs to the user group `PUBLIC` and is available to every database user.

Use public synonyms sparingly because they make database consolidation more difficult (for an example, see *Oracle Database Concepts*).

If you do not want to use a synonym, you can create a local subprogram to invoke the remote subprogram. For example:

```
CREATE OR REPLACE PROCEDURE local_procedure
  (arg IN NUMBER)
AS
BEGIN
  fire_emp1@boston_server(arg);
END;
/
DECLARE
  arg NUMBER;
BEGIN
  local_procedure(arg);
END;
/
```

See Also:

- *Oracle Database Concepts* for general information about synonyms
- *Oracle Database SQL Language Reference* for information about the CREATE SYNONYM statement
- *Oracle Database SQL Language Reference* for information about the GRANT statement

11.8.4.2 Transactions That Invoke Remote Subprograms

A remote subprogram invocation is assumed to update a database. Therefore, a transaction that invokes a remote subprogram requires a two-phase commit (even if the remote subprogram does not update a database). If the transaction is rolled back, then the work done by the remote subprogram is also rolled back.

With respect to the statements COMMIT, ROLLBACK, and SAVEPOINT, a remote subprogram differs from a local subprogram in these ways:

- If the transaction starts on a database that is not an Oracle database, then the remote subprogram cannot run these statements.
This situation can occur in Oracle XA applications, which are not recommended. For details, see [Chapter 19, "Developing Applications with Oracle XA."](#)
- After running one of these statements, the remote subprogram cannot start its own distributed transactions.
A **distributed transaction** updates two or more databases. Statements in the transaction are sent to the different databases, and the transaction succeeds or fails as a unit. If the transaction fails on any database, then it must be rolled back (either to a savepoint or completely) on all databases. Consider this when creating subprograms that perform distributed updates.
- If the remote subprogram does not commit or roll back its work, then the work is implicitly committed when the database link is closed. Until then, the remote subprogram is considered to be performing a transaction. Therefore, further invocations to the remote subprogram are not allowed.

11.9 Invoking Stored PL/SQL Functions from SQL Statements

Caution: Because SQL is a declarative language, rather than an imperative (or procedural) one, you cannot know how many times a function invoked by a SQL statement will run—even if the function is written in PL/SQL, an imperative language.

If your application requires that a function be executed a certain number of times, do not invoke that function from a SQL statement. Use a cursor instead.

For example, if your application requires that a function be called for each selected row, then open a cursor, select rows from the cursor, and call the function for each row. This technique guarantees that the number of calls to the function is the number of rows fetched from the cursor.

For general information about cursors, see *Oracle Database PL/SQL Language Reference*.

These SQL statements can invoke PL/SQL stored functions:

- INSERT
- UPDATE
- DELETE
- SELECT

(SELECT can also invoke a PL/SQL function declared and defined in its WITH clause. For more information, see *Oracle Database SQL Language Reference*.)

- CALL

(CALL can also invoke a PL/SQL stored procedure.)

To invoke a PL/SQL function from a SQL statement, you must either own or have the EXECUTE privilege on the function. To select from a view defined with a PL/SQL function, you must have READ or SELECT privilege on the view. No separate EXECUTE privileges are needed to select from the view.

Note: The AUTHID property of the PL/SQL function can also affect the privileges that you need to invoke the function from a SQL statement, because AUTHID affects the name resolution and privilege checking of SQL statements that the unit issues at runtime. For details, see *Oracle Database PL/SQL Language Reference*.

For general information about invoking subprograms, including passing parameters, see *Oracle Database PL/SQL Language Reference*.

Topics:

- [Why Invoke PL/SQL Functions from SQL Statements?](#)
- [Where PL/SQL Functions Can Appear in SQL Statements](#)
- [When PL/SQL Functions Can Appear in SQL Expressions](#)
- [Controlling Side Effects of PL/SQL Functions Invoked from SQL Statements](#)

11.9.1 Why Invoke PL/SQL Functions from SQL Statements?

Invoking PL/SQL functions in SQL statements can:

- 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 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 data types (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 run in parallel (using the parallel query option).

11.9.2 Where PL/SQL Functions Can Appear in SQL Statements

A PL/SQL function can appear in a SQL statement wherever a SQL function or an expression can appear in a SQL statement. For example:

- 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 instead of:

- Column name in the `SELECT` list
- Table name in the `FROM` clause

A PL/SQL function cannot appear in these contexts, which require unchanging definitions:

- `CHECK` constraint clause of a `CREATE` or `ALTER TABLE` statement
- Default value specification for a column

11.9.3 When PL/SQL Functions Can Appear in SQL Expressions

To be invoked from a SQL expression, a PL/SQL function must satisfy these requirements:

- It must be either a user-defined aggregate function or a row function.
- Its formal parameters must be `IN` parameters, not `OUT` or `IN OUT` parameters.

The function in [Example 11–10](#) satisfies the preceding requirements.

Example 11–10 PL/SQL Function in SQL Expression (Follows Rules)

```
DROP TABLE payroll; -- in case it exists
CREATE TABLE payroll (
    srate NUMBER,
    orate NUMBER,
    acctno NUMBER
);

CREATE OR REPLACE FUNCTION gross_pay (
    emp_id IN NUMBER,
    st_hrs IN NUMBER := 40,
    ot_hrs IN NUMBER := 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;
```

/

11.9.4 Controlling Side Effects of PL/SQL Functions Invoked from SQL Statements

A subprogram has **side effects** if it changes anything except the values of its own local variables. For example, a subprogram that changes any of the following has side effects:

- Its own OUT or IN OUT parameter
- A global variable
- A public variable in a package
- A database table
- The database
- The external state (by invoking DBMS_OUTPUT or sending e-mail, for example)

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.

Some side effects are not allowed in a function invoked from a SQL query or DML statement.

Before Oracle Database 8g Release 1 (8.1), application developers used PRAGMA RESTRICT_REFERENCES to assert the **purity** (freedom from side effects) of a function. This pragma remains available for backward compatibility, but do not use it in new applications. Instead, specify the optimizer hints DETERMINISTIC and PARALLEL_ENABLE when you create the function. For information about DETERMINISTIC and PARALLEL_ENABLE, see *Oracle Database PL/SQL Language Reference*.

Topics:

- [Restrictions on Functions Invoked from SQL Statements](#)
- [PL/SQL Functions Invoked from Parallelized SQL Statements](#)
- [PRAGMA RESTRICT_REFERENCES \(deprecated\)](#)

11.9.4.1 Restrictions on Functions Invoked from SQL Statements

Note: The restrictions on functions invoked from SQL statements also apply to triggers fired by SQL statements.

If a SQL statement invokes a function, and the function runs a new SQL statement, then the execution of the new statement is logically embedded in the context of the statement that invoked the function. To ensure that the new statement is safe in this context, Oracle Database enforces these restrictions on the function:

- If the SQL statement that invokes the function is a query or DML statement, then the function cannot end the current transaction, create or rollback to a savepoint, or ALTER the system or session.
- If the SQL statement that invokes the function is a query or parallelized DML statement, then the function cannot run a DML statement or otherwise modify the database.

- If the SQL statement that invokes the function is a DML statement, then the function can neither read nor modify the table being modified by the SQL statement that invoked the function.

The restrictions apply regardless of how the function runs the new SQL statement. For example, they apply to new SQL statements that the function:

- Invokes from PL/SQL, whether embedded directly in the function body, run using the `EXECUTE IMMEDIATE` statement, or run using the `DBMS_SQL` package
- Embeds in Java with SQLJ syntax or runs using JDBC
- Runs with OCI using the callback context from within an external C function

To avoid these restrictions, ensure that the execution of the new SQL statement is not logically embedded in the context of the SQL statement that invokes the function. For example, put the new SQL statement in an autonomous transaction or, in OCI, create a new connection for the external C function rather than using the handle provided by the `OCIExtProcContext` argument. For information about autonomous transactions, see [Section 6.8](#).

11.9.4.2 PL/SQL Functions Invoked from Parallelized SQL Statements

When Oracle Database runs a **parallelized** SQL statement, multiple processes work simultaneously to run the single SQL statement. When a parallelized SQL statement invokes a function, each process might invoke its own copy of the function, for only the subset of rows that the process handles.

Each process has its own copy of package variables. When parallel execution begins, the package variables are initialized for each process as if a user were logging into the system; the package variable values are not copied from the original login session. Changes that one process makes to package variables do not automatically propagate to the other processes or to the original login session. Java `STATIC` class attributes are similarly initialized and modified independently in each process. A function can use package and Java `STATIC` variables to accumulate a value across the various rows that it encounters. Therefore, Oracle Database does not parallelize the execution of user-defined functions by default.

Before Oracle Database 8g Release 1 (8.1):

- If a parallelized query invoked a user-defined function, then the execution of the function could be parallelized if `PRAGMA RESTRICT_REFERENCES` asserted both `RNPS` and `WNPS` for the function—that is, that the function neither referenced package variables nor changed their values.

Without this assertion, the execution of a standalone PL/SQL function (but not a C or Java function) could be parallelized if Oracle Database determined that the function neither referenced package variables nor changed their values.

- If a parallelized DML statement invoked a user-defined function, then the execution of the function could be parallelized if `PRAGMA RESTRICT_REFERENCES` asserted `RNDS`, `WNDS`, `RNPS` and `WNPS` for the function—that is, that the function neither referenced nor changed the values of either package variables or database tables.

Without this assertion, the execution of a standalone PL/SQL function (but not a C or Java function) could be parallelized if Oracle Database determined that the function neither referenced nor changed the values of either package variables or database tables.

As of Oracle Database 8g Release 1 (8.1), if a parallelized SQL statement invokes a user-defined function, then the execution of a function can be parallelized in these situations:

- The function was created with PARALLEL_ENABLE.
- Before Oracle Database 8g Release 1 (8.1), the database recognized the function as parallelizable.

11.9.4.3 PRAGMA RESTRICT_REFERENCES

Note: PRAGMA RESTRICT_REFERENCES is deprecated. In new applications, Oracle recommends using DETERMINISTIC and PARALLEL_ENABLE (explained in *Oracle Database SQL Language Reference*) instead of RESTRICT_REFERENCES.

In existing PL/SQL applications, you can either remove PRAGMA RESTRICT_REFERENCES or continue to use it, even with new functionality, to ease integration with the existing code. For example:

- When it is impossible or impractical to completely remove PRAGMA RESTRICT_REFERENCES from existing code.

For example, if subprogram S1 depends on subprogram S2, and you do not remove the pragma from S1, then you might need the pragma in S2 to compile S1.
- When replacing PRAGMA RESTRICT_REFERENCES with PARALLEL_ENABLE and DETERMINISTIC in existing code would negatively affect the action of new, dependent code.

To used PRAGMA RESTRICT_REFERENCES to assert the purity of a function: In the package specification (not the package body), anywhere after the function declaration, use this syntax:

```
PRAGMA RESTRICT_REFERENCES (function_name, assertion [, assertion]... );
```

Where *assertion* is one of the following:

Assertion	Meaning
RNPS	The function reads no package state (does not reference the values of package variables)
WNPS	The function writes no package state (does not change the values of package variables).
RNDS	The function reads no database state (does not query database tables).
WNDS	The function writes no database state (does not modify database tables).
TRUST	Trust that no SQL statement in the function body violates any assertion made for the function. For more information, see Section 11.9.4.3.1 .

If you do not specify TRUST, and a SQL statement in the function body violates an assertion that you do specify, then the PL/SQL compiler issues an error message when it parses a violating statement.

Assert the highest purity level (the most assertions) that the function allows, so that the PL/SQL compiler never rejects the function unnecessarily.

Note: If the function invokes subprograms, then either specify PRAGMA RESTRICT_REFERENCES for those subprograms also or specify TRUST in either the invoking function or the invoked subprograms.

See Also: *Oracle Database PL/SQL Language Reference* for more information about PRAGMA RESTRICT_REFERENCES

Example 11-11 creates a function that neither reads nor writes database or package state, and asserts that it has the maximum purity level.

Example 11-11 PRAGMA RESTRICT_REFERENCES

```

DROP TABLE accounts; -- in case it exists
CREATE TABLE accounts (
    acctno    INTEGER,
    balance   NUMBER
);

INSERT INTO accounts (acctno, balance)
VALUES (12345, 1000.00);

CREATE OR REPLACE PACKAGE finance AS
    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
    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;
/
DECLARE
    interest NUMBER;
BEGIN
    SELECT finance.compound_(5, 1000, 6)
    INTO interest
    FROM accounts
    WHERE acctno = 12345;
END;
/

```

Topics:

- [Specifying the Assertion TRUST](#)
- [Differences between Static and Dynamic SQL Statements](#)

11.9.4.3.1 Specifying the Assertion TRUST When PRAGMA RESTRICT REFERENCES specifies TRUST, the PL/SQL compiler does not check the subprogram body for violations.

TRUST makes it easier for a subprogram that uses PRAGMA RESTRICT REFERENCES to invoke subprograms that do not use it.

If your PL/SQL subprogram invokes a C or Java subprogram, then you must specify TRUST for either the PL/SQL subprogram (as in [Example 11-12](#)) or the C or Java subprogram (as in [Example 11-13](#)), because the PL/SQL compiler cannot check a C or Java subprogram for violations at runtime.

Example 11-12 PRAGMA RESTRICT REFERENCES with TRUST on Invoker

```
CREATE OR REPLACE PACKAGE p IS
  PROCEDURE java_sleep (milli_seconds IN NUMBER)
  AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';

  FUNCTION f (n NUMBER) RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES(f,WNDS,TRUST);
END p;
/
CREATE OR REPLACE PACKAGE BODY p IS
  FUNCTION f (
    n NUMBER
  ) RETURN NUMBER
  IS
  BEGIN
    java_sleep(n);
    RETURN n;
  END f;
END p;
/
```

Example 11-13 PRAGMA RESTRICT REFERENCES with TRUST on Invokee

```
CREATE OR REPLACE PACKAGE p IS
  PROCEDURE java_sleep (milli_seconds IN NUMBER)
  AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';
  PRAGMA RESTRICT_REFERENCES(java_sleep,WNDS,TRUST);

  FUNCTION f (n NUMBER) RETURN NUMBER;
END p;
/
CREATE OR REPLACE PACKAGE BODY p IS
  FUNCTION f (
    n NUMBER
  ) RETURN NUMBER
  IS
  BEGIN
    java_sleep(n);
    RETURN n;
  END f;
END p;
/
```

11.9.4.3.2 Differences between Static and Dynamic SQL Statements A static INSERT, UPDATE, or DELETE statement does not violate RNDS if it does not explicitly read a database state (such as a table column). A dynamic INSERT, UPDATE, or DELETE statement always violate RNDS, regardless of whether it explicitly reads a database state.

The following `INSERT` statement violates RNDS if it is executed dynamically, but not if it is executed statically:

```
INSERT INTO my_table values(3, 'BOB');
```

The following `UPDATE` statement always violates RNDS, whether it is executed statically or dynamically, because it explicitly reads the column name of `my_table`:

```
UPDATE my_table SET id=777 WHERE name='BOB';
```

11.10 Debugging Stored Subprograms

To compile a stored subprogram, you must fix any syntax errors in the code. To ensure that the subprogram works correctly, performs well, and recovers from errors, you might need to do additional debugging. Such debugging might involve:

- Adding extra output statements to verify execution progress and check data values at certain points within the subprogram.
- To output the value of variables and expressions, use the `PUT` and `PUT_LINE` subprograms in the Oracle package `DBMS_OUTPUT`, described in *Oracle Database PL/SQL Packages and Types Reference*.
- Analyzing execution in greater detail by running PL/Scope, the PL/SQL hierarchical profiler, or a debugger

Topics:

- [PL/Scope](#)
- [PL/SQL Hierarchical Profiler](#)
- [Compiling Code for Debugging](#)
- [Privileges for Debugging PL/SQL and Java Stored Subprograms](#)
- [DBMS_DEBUG Package](#)

See Also:

- *Oracle Database PL/SQL Language Reference* for information about handling errors in PL/SQL subprograms and packages
- *Oracle Database Java Developer's Guide* for information about debugging Java applications

11.10.1 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 lets you develop 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 12, "Using PL/Scope."](#)

11.10.2 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 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 13, "Using the PL/SQL Hierarchical Profiler."](#)

11.10.3 Compiling Code for Debugging

A debugger can stop on individual code lines and access variables only in code compiled with debug information generated.

To compile a PL/SQL unit with debug information generated, set the compilation parameter `PLSQL_OPTIMIZE_LEVEL` to 1 (the default value is 2). For more information about `PLSQL_OPTIMIZE_LEVEL`, see *Oracle Database Reference*. For information about PL/SQL units, see [Section 11.3](#).

Note: The PL/SQL compiler never generates debug information for code hidden with the PL/SQL wrap utility. For information about the wrap utility, see *Oracle Database PL/SQL Language Reference*.

11.10.4 Privileges for Debugging PL/SQL and Java Stored Subprograms

As of Oracle Database 10g, a new privilege model applies to debugging PL/SQL and Java code running in the database. This model applies to all development environments.

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. The effective user might be the owner of a DR subprogram involved in making the connect call.

When a session connects to a debugger, the session login user and the enabled session-level roles are fixed as the privilege environment for that debugging connection. The privileges needed for debugging must be granted to that combination of user and roles on the relevant code. The privileges are:

- To display and change variables declared in a PL/SQL package specification or Java public variables: either `EXECUTE` or `DEBUG`.
- To display and change private variables, or to breakpoint and run code lines step by step: `DEBUG`

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.

Granting the `DEBUG ANY PROCEDURE` system privilege is equivalent to granting the `DEBUG` privilege on all objects in the database. Objects owned by `SYS` are included if the value of the `07_DICTIONARY_ACCESSIBILITY` parameter is `TRUE`.

Caution: Granting the DEBUG ANY PROCEDURE privilege, or granting the DEBUG privilege on any object owned by SYS, grants complete rights to the database.

See Also:

- *Oracle Database SQL Language Reference* for information about system and object privileges
- *Oracle Database Java Developer's Guide* for information about privileges for debugging Java subprograms

11.10.5 DBMS_DEBUG Package

The DBMS_DEBUG package is a PL/SQL interface to the PL/SQL debugger layer, Probe, in the Oracle server. This API is primarily intended to implement server-side debuggers and it provides a way to debug server-side PL/SQL program units.

Several debuggers, such as Oracle Procedure Builder and various third-party vendor solutions, use this API.

If you are writing low-level debugging code (for example, code for part of a debugger), you might need to use DBMS_DEBUG.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about DBMS_DEBUG

11.11 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. Therefore, the session state is lost for all package instances in the session.

When a package in a given session is invalidated, the session receives ORA-04068 the first time it tries to use any object of the invalid package instance. The second time a session makes such a package call, the package is reinstated for the session without error. However, if you handle this error in your application, be aware of the following:

- For optimal performance, Oracle Database returns this error message only when the package state is discarded. When a subprogram in one package invokes a subprogram in another package, the session state is lost for both packages.
- If a server session traps ORA-04068, then ORA-04068 is not raised for the client session. Therefore, when the client session tries to use an object in the package, the package is not reinstated. To reinstantiate the package, the client session must either reconnect to the database or recompile the package.

In [Example 11-14](#), the RAISE statement raises the current exception, ORA-04068, which is the cause of the exception being handled, ORA-06508. ORA-04068 is not trapped.

Example 11-14 Raising ORA-04068

```
PROCEDURE p IS
  package_exception EXCEPTION;
  PRAGMA EXCEPTION_INIT (package_exception, -6508);
BEGIN
  ...

```

```
EXCEPTION
  WHEN package_exception THEN
    RAISE;
END;
/
```

In [Example 11-15](#), the RAISE statement raises the exception ORA-20001 in response to ORA-06508, instead of the current exception, ORA-04068. ORA-04068 is trapped. When this happens, the ORA-04068 error is masked, which stops the package from being reinstated.

Example 11-15 Trapping ORA-04068

```
PROCEDURE p IS
  package_exception EXCEPTION;
  other_exception   EXCEPTION;
  PRAGMA EXCEPTION_INIT (package_exception, -6508);
  PRAGMA EXCEPTION_INIT (other_exception, -20001);
BEGIN
  ...
EXCEPTION
  WHEN package_exception THEN
    ...
    RAISE other_exception;
END;
/
```

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.

12

Using PL/Scope

PL/Scope is a compiler-driven tool that collects data about identifiers in 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, assignment) and the location of each usage in the source code.

PL/Scope lets you develop powerful and effective PL/Scope source code browsers that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

PL/Scope is intended for application developers, and is usually used in the environment of a development database.

Note: PL/Scope cannot collect data for a PL/SQL 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](#)
- [PL/Scope Identifier Data for STANDARD and DBMS_STANDARD](#)
- [How Much Space is PL/Scope Data Using?](#)
- [Viewing PL/Scope Data](#)
- [Identifier Types that PL/Scope Collects](#)
- [Usages that PL/Scope Reports](#)
- [Sample PL/Scope Session](#)

12.1 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, set the PL/SQL compilation parameter PLSCOPE_SETTINGS to 'IDENTIFIERS:ALL'.

Note: Collecting all identifiers might generate large amounts of data and slow compile time.

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`.

See Also:

- *Oracle Database Reference* for information about `PLSCOPE_SETTINGS`
- *Oracle Database PL/SQL Language Reference* for information about PL/SQL compilation parameters

12.2 PL/Scope Identifier Data for STANDARD and DBMS_STANDARD

The packages `STANDARD` and `DBMS_STANDARD` declare and define base types, such as `VARCHAR2` and `NUMBER`, and subprograms such as `RAISE_APPLICATION_ERROR`. If your database has PL/Scope identifier data for these packages, PL/Scope can track your usage of the identifiers that these packages create.

Topics:

- [Do You Need STANDARD and DBMS_STANDARD Identifier Data?](#)
- [Does Your Database Have STANDARD and DBMS_STANDARD Identifier Data?](#)
- [Recompiling STANDARD and DBMS_STANDARD](#)
- [Running utlrp.sql to Recompile Invalid PL/SQL Objects](#)

12.2.1 Do You Need STANDARD and DBMS_STANDARD Identifier Data?

You can use PL/Scope without `STANDARD` and `DBMS_STANDARD` identifier data. You need this data only if you must know where your code uses the base types or subprograms that these packages create—for example, to know where your code uses the base type `BINARY_INTEGER`, so that you can substitute `PLS_INTEGER`.

12.2.2 Does Your Database Have STANDARD and DBMS_STANDARD Identifier Data?

A newly created Oracle 11.1.0.7 database, or a database that was upgraded to 11.1.0.7 from 10.2, has PL/Scope identifier data for the packages `STANDARD` and `DBMS_STANDARD`. A database that was upgraded to 11.1.0.7 from 11.1.0.6 does not have this data.

To see if your database has this data, use the query in [Example 12-1](#).

[Example 12-1](#) shows what the query returns when the database has PL/Scope identifier data for `STANDARD` and `DBMS_STANDARD`.

Example 12-1 STANDARD and DBMS_STANDARD PL/Scope Identifier Data Available?

Query:

```
COLUMN OBJECT_NAME FORMAT A14

SELECT UNIQUE OBJECT_NAME
FROM ALL_IDENTIFIERS
WHERE OBJECT_NAME IN ('STANDARD', 'DBMS_STANDARD')
AND OWNER='SYS'
ORDER BY OBJECT_NAME;
```

Result:

```
OBJECT_NAME
```

```
-----
DBMS_STANDARD
STANDARD
```

2 rows selected.

If the query in [Example 12-1](#) selects no rows, then the database does not have PL/Scope identifier data for the packages STANDARD and DBMS_STANDARD. To collect this data, a DBA must recompile the packages STANDARD and DBMS_STANDARD, as explained in [Section 12.2.3](#).

12.2.3 Recompiling STANDARD and DBMS_STANDARD

A DBA can use this procedure to recompile the packages STANDARD and DBMS_STANDARD:

Note: This procedure invalidates and revalidates (by recompiling) every PL/SQL object in the database.

1. Connect to the database, shut it down, and then start it in UPGRADE mode:

```
CONNECT / AS SYSDBA;
SHUTDOWN IMMEDIATE;
STARTUP PFILE=parameter_initialization_file UPGRADE;
```

2. Have PL/Scope collect data for all identifiers in the packages STANDARD and DBMS_STANDARD:

```
ALTER SESSION SET PLSCOPE_SETTINGS='IDENTIFIERS:ALL';
```

3. Invalidate and recompile the database:

```
@?/rdbms/admin/utlirp.sql
```

Now all PL/SQL objects in the database are invalid except STANDARD and DBMS_STANDARD, which were recompiled with PLSCOPE_SETTINGS='IDENTIFIERS:ALL'.

4. (Optional) Invalidate any other PL/SQL objects to be recompiled with PLSCOPE_SETTINGS='IDENTIFIERS:ALL', using a script similar to this.

Customize the query on lines 5 through 9 to invalidate only those objects for which you need PL/Scope identifier data. Collecting all identifiers for all objects, as this script does, might generate large amounts of data and slow compile time:

```
DECLARE
  TYPE ObjIDArray IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  ObjIDs  ObjIDArray;
BEGIN
  SELECT object_id BULK COLLECT INTO ObjIDs
  FROM ALL_OBJECTS
  WHERE object_type IN
    (SELECT DISTINCT TYPE
     FROM ALL_PLSQL_OBJECT_SETTINGS);
  FOR i IN 1..SQL%ROWCOUNT LOOP
    BEGIN
      DBMS_UTILITY.INVALIDATE(ObjIDs(i),
      'PLSCOPE_SETTINGS=IDENTIFIERS:ALL REUSE SETTINGS');
      NULL;
    END;
  END;
```

```

    END LOOP;
END;
/

```

Notes: In the preceding script:

- Do not substitute ObjIDs.LAST for SQL%ROWCOUNT, because ObjIDs attributes are dependent on a package that is locked by the anonymous block.
 - If your database is large, do not substitute a cursor FOR LOOP for the BULK COLLECT statement, or you will run out of resources.
-

5. Shut down the database, and then start it in NORMAL mode:

```

SHUTDOWN IMMEDIATE;
STARTUP PFILE=parameter_initialization_file;

```

6. For any remaining invalid PL/SQL objects, do either of these:

- Allow them to be recompiled automatically, as they are referenced.
(This can be slow if there are complex dependencies.)
- Run the script utlrp.sql to recompile the invalid PL/SQL objects, as explained in [Section 12.2.4](#).

12.2.4 Running utlrp.sql to Recompile Invalid PL/SQL Objects

If the database was restarted in NORMAL mode (step 5 on page 12-4), then a DBA, or a user who has been granted the DBA role, can use this procedure:

1. Connect to the database as SYS:

```
CONNECT / AS SYS;
```

2. Run the script utlrp.sql:

```
@?/rdbms/admin/utlrp.sql
```

If the script gives you any instructions, follow them, and then run the script again.

If the script terminates abnormally without giving any instructions, run it again.

12.3 How Much Space is PL/Scope Data Using?

PL/Scope stores its data in the SYSAUX tablespace. If you are logged on as SYSDBA, you can use the query in [Example 12-2](#) to display the amount of space that PL/Scope data is using.

Example 12-2 How Much Space is PL/Scope Data Using?

Query:

```

SELECT SPACE_USAGE_KBYTES
FROM V$SYSAUX_OCCUPANTS
WHERE OCCUPANT_NAME= 'PL/SCOPE';

```

Result:

```
SPACE_USAGE_KBYTES
```

1600

1 row selected.

For information about managing the SYSAUX tablespace, see *Oracle Database Administrator's Guide*.

12.4 Viewing PL/Scope Data

To view the data that PL/Scope has collected, you can use either:

- [Section 12.4.1, "Static Data Dictionary Views"](#)
- [Section 12.4.2, "Demo Tool"](#)
- [Section 12.4.3, "SQL Developer"](#)

12.4.1 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](#)

12.4.1.1 Unique Keys

Each row of a `*_IDENTIFIERS` view represents a unique usage of an identifier in the PL/SQL unit. In each of these views, these are equivalent unique keys within a compilation unit:

- LINE, COL, and USAGE
- USAGE_ID

For the usages in the `*_IDENTIFIERS` views, see [Section 12.6](#).

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).

12.4.1.2 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 12-3](#) shows.

Example 12-3 `USAGE_CONTEXT_ID` and `USAGE_ID`

```

ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';

CREATE OR REPLACE PROCEDURE a (p1 IN BOOLEAN) AUTHID DEFINER 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 or REPLACE PROCEDURE b (
  p2 OUT PLS_INTEGER,
  p3 IN OUT VARCHAR2
) AUTHID DEFINER
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
/

```

Result:

```
IDENTIFIER_USAGE_CONTEXTS
-----
B.....procedure declaration
B.....procedure definition
P2.....formal out declaration
  Pls_Integer... subtype reference
P3.....formal in out declaration
  Varchar2..... character datatype reference
N.....variable declaration
  Number..... number datatype reference
Q.....variable declaration
  Q.....variable assignment
  Boolean..... boolean datatype reference
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
    Char..... subtype reference
  N.....variable assignment
  D.....constant reference
```

24 rows selected.

12.4.1.3 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 12-4](#), 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 12-4 Program Unit with Two Identifiers Named p

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  PROCEDURE p IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Inner p');
    END p;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Outer p');
    p();
  END p;
/
```

12.4.2 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.

12.4.3 SQL Developer

PL/Scope is a feature of SQL Developer. For information about using PL/Scope from SQL Developer, see the SQL Developer online help or documentation.

12.5 Identifier Types that PL/Scope Collects

[Table 12–1](#) shows the identifier types that PL/Scope collects, in alphabetical order. The identifier types in [Table 12–1](#) 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 12–1 Identifier Types that PL/Scope Collects

TYPE Column	Value	Comment
ASSOCIATIVE ARRAY		
CONSTANT		
CURSOR		
BFILE	DATATYPEBLOB	Each DATATYPE is a base type declared in package STANDARD.
DATATYPEBOOLEAN		
DATATYPECHARACTER		
DATATYPECLOB		
DATATYPEDATE		
DATATYPEINTERVAL		
DATATYPENUMBER		
DATATYPETIME		
DATATYPETIMESTAMP		
DATATYPE		
EXCEPTION		
FORMAL INFORMAL IN		
OUTFORMAL OUT		
FUNCTION		
INDEX TABLE		
ITERATOR		An iterator is the index of a FOR loop.
LABEL		A label declaration also acts as a context.
LIBRARY		
NESTED TABLE		
OBJECT		
OPAQUE		Examples of internal opaque types are ANYDATA and XMLType.
PACKAGE		
PROCEDURE		

Table 12-1 (Cont.) Identifier Types that PL/Scope Collects

TYPE	Column Value	Comment
	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.

12.6 Usages that PL/Scope Reports

[Table 12-2](#) shows the usages that PL/Scope reports, in alphabetical order. The identifier types in [Table 12-2](#) appear in the USAGE column of the *_IDENTIFIER static data dictionary views, which are described in *Oracle Database Reference*.

Table 12-2 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: <ul style="list-style-type: none"> ▪ 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 variable 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 call onto the call stack; that is: <ul style="list-style-type: none"> ▪ A call to a FUNCTION or PROCEDURE ▪ Running 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 call onto the call stack.	
DECLARATION	A DECLARATION tells the compiler that an identifier exists, and each identifier has exactly one DECLARATION. Each DECLARATION can have an associated data type. <p>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.</p> <p>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 >>.</p>	

Table 12–2 (Cont.) Usages that PL/Scope Reports

USAGE Column Value	Description
DEFINITION	<p>A DEFINITION tells the compiler how to implement or use a previously declared identifier.</p> <p>Each of these types of identifiers has a DEFINITION:</p> <ul style="list-style-type: none"> ■ EXCEPTION (can have multiple definitions) ■ FUNCTION ■ OBJECT ■ PACKAGE ■ PROCEDURE ■ TRIGGER
REFERENCE	<p>A REFERENCE uses an identifier without changing its value. Examples of references are:</p> <ul style="list-style-type: none"> ■ 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 <code>myrecordvar.myfield := 1</code>, a reference is made to <code>myrecordvar</code>, and an assignment is made to <code>myfield</code>. ■ Using a cursor for any purpose except FETCH ■ Passing an identifier to a subprogram by value (IN mode) ■ Using an identifier as the bind variable in the USING clause of an EXECUTE IMMEDIATE statement in either IN or IN OUT mode <p>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).</p>

12.7 Sample PL/Scope Session

In this sample session, assume that you are logged in as HR.

1. Set the session parameter:

```
ALTER SESSION SET PLSCOPE_SETTINGS='IDENTIFIERS:ALL';
```

2. Create this package:

```
CREATE OR REPLACE PACKAGE PACK1 AUTHID DEFINER 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;
/

```

3. Verify that PL/Scope collected all identifiers for the package body:

```

SELECT PLSCOPE_SETTINGS
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME='PACK1' AND TYPE='PACKAGE BODY'

```

Result:

```
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 these SQL*Plus formatting commands and this query:

```

COLUMN NAME FORMAT A6
COLUMN SIGNATURE FORMAT A32
COLUMN TYPE FORMAT A9

```

```

SELECT NAME, SIGNATURE, TYPE
FROM USER_IDENTIFIERS
WHERE NAME LIKE '%1' AND USAGE='DECLARATION'
ORDER BY OBJECT_TYPE, USAGE_ID;

```

Result is similar to:

NAME	SIGNATURE	TYPE
PACK1	41820FA4D5EF6BE707895178D0C5C4EF	PACKAGE
R1	EEBB6849DEE31BC77BF186EBAE5D4E2D	RECORD
RF1	41D70040337349634A7F547BC83517C7	VARIABLE
F1	D51E825FF334817C977174423E3D0765	FUNCTION
FP1	CAC3474C112DBEC67AB926978D9A16C1	FORMAL IN
P1	B7C0576BA4D00C33A65CC0C64CADAB89	PROCEDURE
PP1	6B74CF95A5B7377A735925DFAA280266	FORMAL IN
FP1	98EB63B8A4AFEB5EF94D50A20165D6CC	FORMAL IN
PP1	62D8463A314BE1F996794723402278CF	FORMAL IN
PR1	BDB1CB26C97562CCC20CD1F32D341D7C	VARIABLE

10 rows selected.

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:

```

COLUMN VARIABLE_NAME FORMAT A13
COLUMN CONTEXT_NAME FORMAT A12

```

```

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;

```

Result is similar to:

VARIABLE_NAME	CONTEXT_NAME	SIGNATURE
A	F1	1691C6B3C951FCAA2CBEEB47F85CF128
PR1	P1	BDB1CB26C97562CCC20CD1F32D341D7C

2 rows selected.

6. Find all usages performed on the local variable A:

```

COLUMN USAGE FORMAT A11
COLUMN USAGE_ID FORMAT 999
COLUMN OBJECT_NAME FORMAT A11
COLUMN OBJECT_TYPE FORMAT A12

SELECT USAGE, USAGE_ID, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='1691C6B3C951FCAA2CBEEB47F85CF128' -- signature of A
ORDER BY OBJECT_TYPE, USAGE_ID;

```

Result:

USAGE	USAGE_ID	OBJECT_NAME	OBJECT_TYPE
DECLARATION	6	PACK1	PACKAGE BODY
ASSIGNMENT	8	PACK1	PACKAGE BODY
REFERENCE	9	PACK1	PACKAGE BODY

3 rows selected.

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:

```

COLUMN NAME FORMAT A6
COLUMN TYPE FORMAT A15

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 = 'D51E825FF334817C977174423E3D0765' -- signature of F1
AND a.OBJECT_TYPE = b.OBJECT_TYPE
AND a.OBJECT_NAME = b.OBJECT_NAME;

```

Result:

NAME	TYPE
NUMBER	NUMBER DATATYPE

```
1 row selected.
```

Note: This query produces the output shown only if your database has PL/Scope identifier data for the packages STANDARD and DBMS_STANDARD. For more information, see [Section 12.2](#).

8. Find out where the assignment to local identifier A occurred:

```
SELECT LINE, COL, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='1691C6B3C951FCAA2CBEEB47F85CF128' -- signature of A
AND USAGE='ASSIGNMENT';
```

Result:

LINE	COL	OBJECT_NAME	OBJECT_TYPE
3	5	PACK1	PACKAGE BODY

```
1 row selected.
```


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](#)
- [Collecting Profile Data](#)
- [Understanding Raw Profiler Output](#)
- [Analyzing Profile Data](#)
- [plshprof Utility](#)

13.1 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:

- 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 and write the **raw profiler output** 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.

13.2 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 these 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 this PL/SQL procedure, `test`:

```
CREATE OR REPLACE PROCEDURE test AUTHID DEFINER 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;
/

```

The SQL script in [Example 13-1](#) profiles the execution of the PL/SQL procedure test.

Example 13-1 Profiling a PL/SQL Procedure

```

BEGIN
  /* Start profiling.
  Write raw profiler output to file test.trc in a directory
  that is mapped to directory object PLSHPROF_DIR
  (see note after example). */

  DBMS_HPROF.START_PROFILING('PLSHPROF_DIR', 'test.trc');
END;
/
-- Run 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 path name. For example, if you are connected to the database AS SYSDBA, this CREATE DIRECTORY statement creates the directory object PLSHPROF_DIR and maps it to the file system directory /private/plshprof/results:

```

CREATE DIRECTORY PLSHPROF_DIR AS '/private/plshprof/results';
To run the SQL script in Example 13-1, you must have READ and WRITE
privileges on both PLSHPROF_DIR and the directory to which it is
mapped. If you are connected to the database AS SYSDBA, this GRANT
statement grants READ and WRITE privileges on PLSHPROF_DIR to HR:

```

```
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*.

13.3 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 in [Example 13-1](#) wrote this 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 1
P#C PLSQL.""".__anonymous_block"
P#X 66
P#C PLSQL."SYS"."DBMS_OUTPUT":11."GET_LINES"#660bd56a1b1640db #180
P#X 5
P#R
P#X 29
P#R
P#X 2
P#R
P#C PLSQL.""".__plsql_vm"
P#X 1
P#C PLSQL.""".__anonymous_block"
P#X 42
P#C PLSQL."HR"."TEST":7."TEST"#980980e97e42f8ec #1
P#X 2
P#C PLSQL."HR"."TEST":7."TEST.FOO"#980980e97e42f8ec #4
P#X 24
P#C SQL."HR"."TEST":7."__static_sql_exec_line6" #6
P#X 165
P#R
P#X 13
P#R
P#X 2
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 75
P#R
P#X 2
P#R
P#X 2
P#C PLSQL."HR"."TEST":7."TEST.FOO"#980980e97e42f8ec #4
P#X 2
P#C SQL."HR"."TEST":7."__static_sql_exec_line6" #6
P#X 73
P#R
P#X 3
P#R
P#X 1
P#R
P#X 1
P#R
P#X 1
P#R
P#C PLSQL.""".__plsql_vm"
P#X 2
P#C PLSQL.""".__anonymous_block"
P#X 69
P#C PLSQL."SYS"."DBMS_OUTPUT":11."GET_LINES"#660bd56a1b1640db #180

```

```

P#X 5
P#R
P#X 35
P#R
P#X 1
P#R
P#C PLSQL. .... ."__plsql_vm"
P#X 1
P#C PLSQL. .... ."__anonymous_block"
P#X 58
P#C PLSQL. "SYS". "DBMS_HPROF" ::11. "STOP_PROFILING" #980980e97e42f8ec #63
P#R
P#R
P#R
P#! PL/SQL Timer Stopped

```

Table 13–1 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 properly nested (like matched parentheses). If an unhandled exception causes a called subprogram to exit, the profiler still reports a matching return event.

Each call event (P#C) entry in the raw profiler output includes this information:

- **Namespace** to which the called subprogram belongs (see [Section 13.3.1](#))
- **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 in [Section 13.3.2](#).
- Hexadecimal value that represents an MD5 hash of the **signature** of the called subprogram
The DBMS_HPROF.analyze PL/SQL API (described in [Section 13.4](#)) 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 this 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 these 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.
#4	4 is the line number in the PL/SQL module HR.TEST at which TEST.FOO is defined.

Note: When a subprogram is inlined, it is not reported in the profiler output. For information about subprogram inlining, see *Oracle Database PL/SQL Language Reference*.

When a call to a DETERMINISTIC function is "optimized away," it is not reported in the profiler output. For information about DETERMINISTIC functions, see *Oracle Database PL/SQL Language Reference*.

13.3.1 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, data manipulation language (DML) statements, data definition language (DDL) statements, and native dynamic SQL statements

13.3.2 Special Function Names

PL/SQL hierarchical profiler tracks certain operations as if they were functions with the names and namespaces shown in [Table 13-2](#).

Table 13-2 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
Package initialization block	__pkg_init	PL/SQL
Static SQL statement at line <i>line#</i>	__static_sql_exec_line <i>line#</i>	SQL

Table 13–2 (Cont.) Function Names of Operations that the PL/SQL Hierarchical Profiler

Tracked Operation	Function Name	Namespace
Dynamic SQL statement at line <i>line#</i>	<code>__dyn_sql_exec_line<i>line#</i></code>	SQL
SQL <code>FETCH</code> statement at line <i>line#</i>	<code>__sql_fetch_line<i>line#</i></code>	SQL

13.4 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 13–3](#).

Table 13–3 PL/SQL Hierarchical Profiler Database Tables

Table	Description
<code>DBMSHP_RUNS</code>	Top-level information for this run of <code>DBMS_HPROF.analyze</code> . For column descriptions, see Table 13–4 on page 13–8.
<code>DBMSHP_FUNCTION_INFO</code>	Information for each subprogram profiled in this run of <code>DBMS_HPROF.analyze</code> . For column descriptions, see Table 13–5 on page 13–9.
<code>DBMSHP_PARENT_CHILD_INFO</code>	Parent-child information for each subprogram profiled in this run of <code>DBMS_HPROF.analyze</code> . For column descriptions, see Table 13–6 on page 13–8.

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 [Section 13.5](#).

13.4.1 Creating Hierarchical Profiler Tables

To create the hierarchical profiler tables in [Table 13–3](#) 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 13–3](#) 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 these 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 [Section 13.3](#).)

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 anonymous block in [Example 13-2](#):

- Invokes the function DBMS_HPROF.analyze function, which:
 - Analyzes the profile data in the raw profiler output file test.trc (from [Section 13.3](#)), 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 13-3](#) on page 13-7.
 - Returns a unique identifier that you can use to query the hierarchical profiler tables in [Table 13-3](#) on page 13-7. (By querying these hierarchical profiler tables, you can produce customized reports.)
- Stores the unique identifier in the variable runid, which it prints.

Example 13-2 Invoking DBMS_HPROF.analyze

```
DECLARE
  runid NUMBER;
BEGIN
  runid := DBMS_HPROF.analyze(LOCATION=>'PLSHPROF_DIR',
                               FILENAME=>'test.trc');
  DBMS_OUTPUT.PUT_LINE('runid = ' || runid);
END;
/
```

13.4.2 Understanding Hierarchical Profiler Tables

These topics explain how to use the hierarchical profiler tables in [Table 13-3](#):

- [Section 13.4.2.1, "Hierarchical Profiler Database Table Columns"](#)
- [Section 13.4.2.2, "Distinguishing Between Overloaded Subprograms"](#)
- [Section 13.4.2.3, "Hierarchical Profiler Tables for Sample PL/SQL Procedure"](#)
- [Section 13.4.2.4, "Examples of Calls to DBMS_HPROF.analyze with Options"](#)

13.4.2.1 Hierarchical Profiler Database Table Columns

[Table 13-4](#) 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 13-4 DBMSHP_RUNS Table Columns

Column Name	Column Data Type	Column Contents
RUNID	NUMBER	Unique identifier for this run of DBMS_HPROF.analyze, generated from DBMSHP_RUNNUMBER sequence.
RUN_TIMESTAMP	TIMESTAMP (6)	Time stamp for this run of DBMS_HPROF.analyze.
RUN_COMMENT	VARCHAR2 (2047)	Comment that you provided for this run of DBMS_HPROF.analyze.
TOTAL_ELAPSED_TIME	NUMBER (38)	Total elapsed time for this run of DBMS_HPROF.analyze.

[Table 13–5](#) 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 13–5](#) has a row for each variation of that subprogram. Each variation has its own LINE# and HASH (see [Section 13.4.2.2, "Distinguishing Between Overloaded Subprograms."](#))

The primary key for the hierarchical profiler table DBMSHP_FUNCTION_INFO is RUNID, SYMBOLID.

Table 13–5 DBMSHP_FUNCTION_INFO Table Columns

Column Name	Column Data Type	Column Contents
RUNID	NUMBER	References RUNID column of DBMSHP_RUNS table. For a description of that column, see Table 13–4 .
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 (32)	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 in Section 13.3.2 .
		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 Section 13.3.1 .
SUBTREE_ELAPSED_TIME	NUMBER (38)	Elapsed time, in microseconds, for subprogram, including time spent in descendant subprograms.
FUNCTION_ELAPSED_TIME	NUMBER (38)	Elapsed time, in microseconds, for subprogram, excluding time spent in descendant subprograms.
CALLS	NUMBER (38)	Number of calls to subprogram.

Table 13–6 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 13–6 DBMSHP_PARENT_CHILD_INFO Table Columns

Column Name	Column Data Type	Column Contents
RUNID	NUMBER	References RUNID column of DBMSHP_FUNCTION_INFO table. For a description of that column, see Table 13–5 .
PARENTSYMID	NUMBER	Parent symbol ID. RUNID, PARENTSYMID references DBMSHP_FUNCTION_INFO(RUNID, SYMBOLID).
CHILDSYMID	NUMBER	Child symbol ID. RUNID, CHILDSYMID references DBMSHP_FUNCTION_INFO(RUNID, SYMBOLID).
SUBTREE_ELAPSED_TIME	NUMBER(38)	Elapsed time, in microseconds, for RUNID, CHILDSYMID when called from RUNID, PARENTSYMID, including time spent in descendant subprograms.
FUNCTION_ELAPSED_TIME	NUMBER(38)	Elapsed time, in microseconds, for RUNID, CHILDSYMID when called from RUNID, PARENTSYMID, excluding time spent in descendant subprograms.
CALLS	NUMBER(38)	Number of calls to RUNID, CHILDSYMID from RUNID, PARENTSYMID.

13.4.2.2 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.

13.4.2.3 Hierarchical Profiler Tables for Sample PL/SQL Procedure

The hierarchical profiler tables for the PL/SQL procedure `test` in [Section 13.2](#) are shown in [Example 13–3](#) through [Example 13–5](#).

Example 13–3 DBMSHP_RUNS Table for Sample PL/SQL Procedure

RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME	RUN_COMMENT
1	20-DEC-12 11.37.26.688381 AM	7	
2	20-DEC-12 11.37.26.700523 AM	9	
3	20-DEC-12 11.37.26.706824 AM	123	

Example 13–4 DBMSHP_FUNCTION_INFO Table for Sample PL/SQL Procedure

RUNID	SYMBOLID	OWNER	MODULE	TYPE	FUNCTION
1	1	HR	PKG	PACKAGE BODY	MYPROC
2	1	HR	PKG	PACKAGE BODY	MYFUNC
2	2	HR	PKG	PACKAGE BODY	MYPROC
3	1				<u>anonymous_block</u>
3	2				<u>plsql_vm</u>
3	3	HR	PKG	PACKAGE BODY	MYFUNC
3	4	HR	PKG	PACKAGE BODY	MYPROC
3	5	HR	TEST2	PROCEDURE	TEST2

LINE#	CALLS	HASH	NAMESPACE
2	1	9689BA467A19CD19	PLSQL
7	1	28DC3402BAEB2B0D	PLSQL
2	1	9689BA467A19CD19	PLSQL
0	1		PLSQL
0	1		PLSQL
7	1	28DC3402BAEB2B0D	PLSQL
2	2	9689BA467A19CD19	PLSQL
1	1	980980E97E42F8EC	PLSQL

NAMESPACE	SUBTREE_ELAPSED_TIME	FUNCTION_ELAPSED_TIME
PLSQL	7	7
PLSQL	9	8
PLSQL	1	1
PLSQL	121	44
PLSQL	123	2
PLSQL	9	8
PLSQL	8	8
PLSQL	77	61

Example 13–5 DBMSHP_PARENT_CHILD_INFO Table for Sample PL/SQL Procedure

RUNID	PARENTSYMID	CHILDSYMID	SUBTREE_ELAPSED_TIME	FUNCTION_ELAPSED_TIME	CALLS
2	1	2		1	1
3	1	5		77	61
3	2	1		121	44
3	3	4		1	1
3	5	3		9	8
3	5	4		7	1

Consider the third row of the table DBMSHP_PARENT_CHILD_INFO (Example 13–5). The RUNID column shows that this row corresponds to the third run. The columns PARENTSYMID and CHILDSYMID show that the symbol IDs of the parent (caller) and child (called subprogram) are 2 and 1, respectively. The table DBMSHP_FUNCTION_INFO (Example 13–4) shows that for the third run, the symbol IDs 2 and 1 correspond to procedures plsql_vm and anonymous_block, respectively. Therefore, the information in this row is about calls from the procedure plsql_vm to the anonymous_block (defined within plsql_vm) in the module HR.TEST. This row shows that, when called from the procedure plsql_vm, the function time for the procedure anonymous_block is 44 microseconds, and the time spent in the anonymous_block subtree (including descendants) is 121 microseconds.

13.4.2.4 Examples of Calls to DBMS_HPROF.analyze with Options

For an example of a call to DBMS_HPROF.analyze without options, see [Example 13-2](#).

[Example 13-6](#) creates a package, creates a procedure that invokes subprograms in the package, profiles the procedure, and uses DBMS_HPROF.analyze to analyze the raw profiler output file. The raw profiler output file is in the directory corresponding to the PLSHPROF_DIR directory object.

Example 13-6 Invoking DBMS_HPROF.analyze with Options

```
-- Create package

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER IS
  PROCEDURE myproc (n IN out NUMBER);
  FUNCTION myfunc (v VARCHAR2) RETURN VARCHAR2;
  FUNCTION myfunc (n PLS_INTEGER) RETURN PLS_INTEGER;
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg IS
  PROCEDURE myproc (n 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 (n IN out PLS_INTEGER) IS
  BEGIN
    n := n + 1;
  END;
  BEGIN
    i := n;
    myproc(i);
    RETURN i;
  END;
END pkg;
/

-- Create procedure that invokes package subprograms

CREATE OR REPLACE PROCEDURE test2 AUTHID DEFINER IS
  x NUMBER := 5;
  y VARCHAR2(32767);
BEGIN
  pkg.myproc(x);
  y := pkg.myfunc('hello');
END;
```

```

-- Profile test2

BEGIN
  DBMS_HPROF.START_PROFILING('PLSHPROF_DIR', 'test2.trc');
END;
/
BEGIN
  test2;
END;
/
BEGIN
  DBMS_HPROF.STOP_PROFILING;
END;
/
-- If not done, create hierarchical profiler tables (see Section 13.4.1).

-- Call DBMS_HPROF.analyze with options

DECLARE
  runid NUMBER;
BEGIN
  -- Analyze only subtrees rooted at trace entry "HR"."PKG"."MYPLOC"

  runid := DBMS_HPROF.analyze('PLSHPROF_DIR', 'test2.trc',
                               trace => '"HR"."PKG"."MYPLOC"');

  -- Analyze up to 20 calls to subtrees rooted at trace entry
  -- "HR"."PKG"."MYFUNC". Because "HR"."PKG"."MYFUNC" is overloaded,
  -- both overloads are considered for analysis.

  runid := DBMS_HPROF.analyze('PLSHPROF_DIR', 'test2.trc',
                               collect => 20,
                               trace => '"HR"."PKG"."MYFUNC"');

  -- Analyze second call to PL/SQL virtual machine

  runid := DBMS_HPROF.analyze('PLSHPROF_DIR', 'test2.trc',
                               skip => 1, collect => 1,
                               trace => '""."__plsql_vm"');

END;
/

```

13.5 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 [Section 13.2](#).)

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](#)

13.5.1 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 these:

Option	Description	Default
<code>-skip count</code>	Skips first <i>count</i> calls. Use only with <code>-trace symbol</code> .	0
<code>-collect count</code>	Collects information for <i>count</i> calls. Use only with <code>-trace symbol</code> .	1
<code>-output filename</code>	Specifies name of output file	<i>symbol.html</i> or <i>tracefile1.html</i>
<code>-summary</code>	Prints only elapsed time	None
<code>-trace symbol</code>	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 this command (% is the prompt):

```
% plshprof -output report test.trc
```

13.5.2 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](#)

13.5.2.1 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 output log in a variety of formats. These 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)

Also, 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

13.5.2.2 Function-Level Reports

The function-level reports provide a flat view of the profile information. Each function-level report includes this 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.

This 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)

13.5.2.3 Module-Level Reports

Each module-level report includes this 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.

This 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

13.5.2.4 Namespace-Level Reports

Each namespace-level report includes this 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.

This 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

13.5.2.5 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 [Section 13.5.2.2, "Function-Level Reports"](#)).

This [Sample Report](#) is a fragment of a Parents and Children Report that corresponds to a function named HR.P.UPLOAD. The first row has this 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:

- 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 function 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:

- 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 microseconds (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)
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)

13.5.3 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](#)

13.5.3.1 Difference Report Conventions

Difference reports use these 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 run.

13.5.3.2 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) #

Function	Rel%	Ind%	Calls	Rel%	Function Name
-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

Also, 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

13.5.3.3 Function-Level Difference Reports

Each function-level difference report includes, for each function, the change in these values 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 that:

- 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 run.

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)
-2223737	-2223737		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)
-2223737	-2223737		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)

13.5.3.4 Module-Level Difference Reports

Each module-level report includes, for each module, the change in these values 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

13.5.3.5 Namespace-Level Difference Reports

Each namespace-level report includes, for each namespace, the change in these values 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

13.5.3.6 Parents and Children Difference Report for a Function

The Parents and Children Difference Report for a function shows changes in the execution profiles of these 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 this 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 sample report 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. This 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%

Developing PL/SQL Web Applications

Note: The use of the technologies described in this chapter is suitable for applications that require tight control of the HTTP communication and HTML generation. For others applications, you are encouraged to use Oracle Application Express, which provides more features and a convenient graphical interface to ease application development.

For information about using Oracle Application Express, see *Oracle Database 2 Day + Application Express Developer's Guide*.

This chapter explains how to develop PL/SQL web applications, which let you make your database available on the intranet.

Topics:

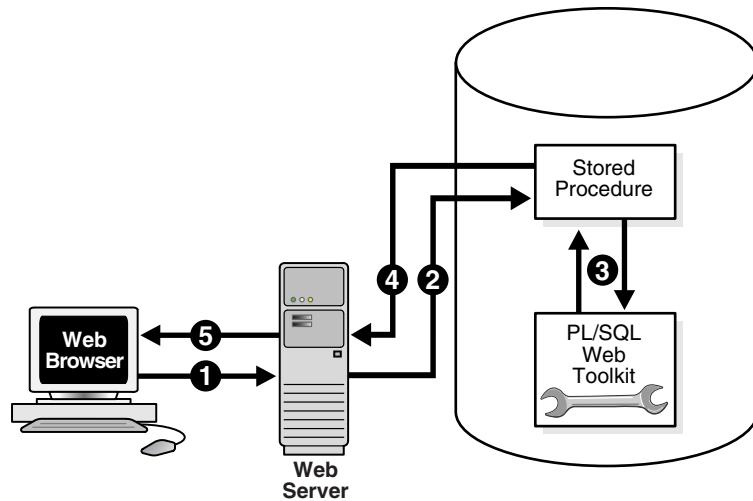
- [Overview of PL/SQL Web Applications](#)
- [Implementing PL/SQL Web Applications](#)
- [Using mod_plsql Gateway to Map Client Requests to a PL/SQL Web Application](#)
- [Using Embedded PL/SQL Gateway](#)
- [Generating HTML Output with PL/SQL](#)
- [Passing Parameters to PL/SQL Web Applications](#)
- [Performing Network Operations in PL/SQL Subprograms](#)

14.1 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 the database. Delivering web content with PL/SQL stored subprograms provides the power and flexibility of database processing. For example, you can use data manipulation language (DML) statements, dynamic SQL statements, and cursors. You also eliminate the process overhead of forking a new CGI process to handle each HTTP request.

[Figure 14-1](#) illustrates the generic process for a PL/SQL web application.

Figure 14–1 PL/SQL Web Application

14.2 Implementing PL/SQL Web Applications

You can implement a web browser-based application entirely in PL/SQL with these Oracle Database components:

- [PL/SQL Gateway](#)
- [PL/SQL Web Toolkit](#)

14.2.1 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.

14.2.1.1 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 the 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 advantages of using `mod_plsql` over the embedded form of the PL/SQL gateway are:

- 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.

14.2.1.2 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 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 advantages of using the embedded gateway instead of `mod_plsql` are:

- You can invoke PL/SQL web applications like Application Express without installing Oracle HTTP Server, thereby simplifying installation, configuration, and administration of PL/SQL based web applications.
- You use the same configuration approach that is used to deliver content from Oracle XML DB in response to FTP and HTTP requests.

14.2.2 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 PL/SQL Web Toolkit API enables stored subprograms to perform actions such as:

- Obtain information about an HTTP request
- Generate HTTP headers such as content-type and mime-type
- Set browser cookies
- Generate HTML pages

[Table 14-1](#) describes commonly used PL/SQL Web Toolkit packages.

Table 14-1 Commonly Used Packages in the PL/SQL Web Toolkit

Package	Description of Contents
HTF	Function versions of the subprograms in the <code>htp</code> 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 must nest function calls.
HTP	Subprograms that generate HTML tags. For example, the procedure <code>htp.anchor</code> generates the HTML anchor tag, <code><A></code> .
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.

Table 14–1 (Cont.) Commonly Used Packages in the PL/SQL Web Toolkit

Package	Description of Contents
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	These types of utility subprograms: <ul style="list-style-type: none"> ▪ 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 data type.
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

14.3 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* for information about the mod_plsql module

14.4 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:

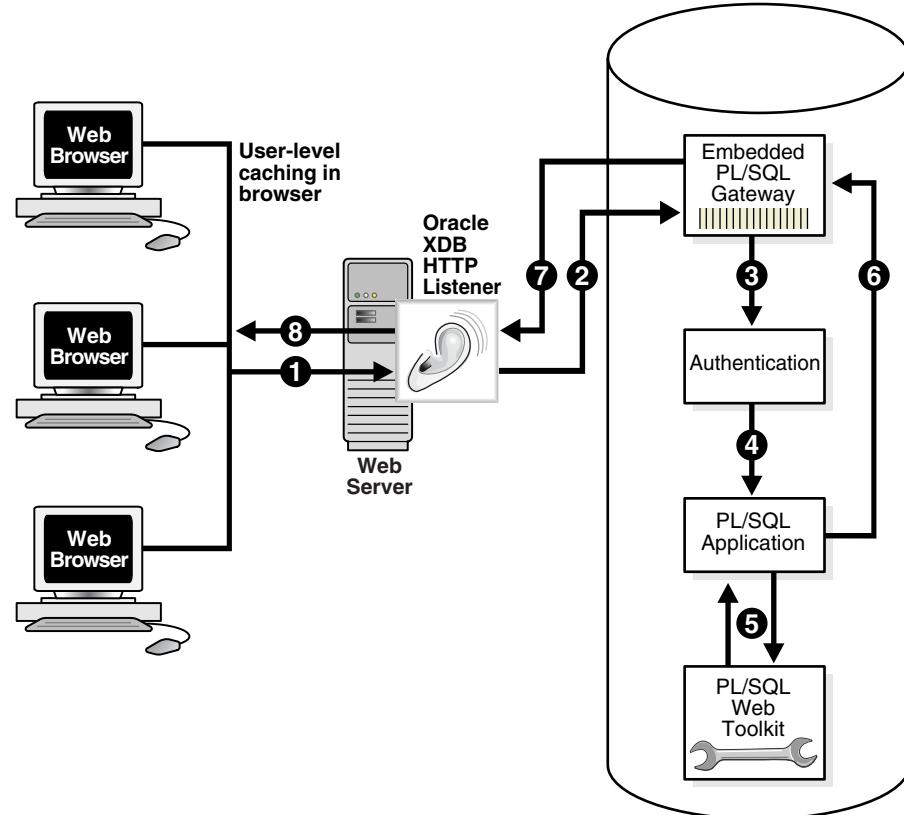
- [How Embedded PL/SQL Gateway Processes Client Requests](#)
- [Installing Embedded PL/SQL Gateway](#)
- [Configuring Embedded PL/SQL Gateway](#)
- [Invoking PL/SQL Stored Subprograms Through Embedded PL/SQL Gateway](#)

- Securing Application Access with Embedded PL/SQL Gateway
- Restrictions in Embedded PL/SQL Gateway
- Using Embedded PL/SQL Gateway: Scenario

14.4.1 How Embedded PL/SQL Gateway Processes Client Requests

Figure 14-2 illustrates the process by which the embedded gateway handles client HTTP requests.

Figure 14-2 Processing Client Requests with Embedded PL/SQL Gateway



The explanation of the steps in Figure 14-2 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 (for instructions, see *Oracle XML DB Developer's Guide*). Configure the HTTP listener either by using Oracle Enterprise Manager Cloud Control (Cloud Control) 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.

14.4.2 Installing Embedded PL/SQL Gateway

The embedded gateway requires these 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 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 the database, so no supplementary installation is necessary.

14.4.3 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.

Topics:

- [Configuring Embedded PL/SQL Gateway: Overview](#)
- [Configuring User Authentication for Embedded PL/SQL Gateway](#)

14.4.3.1 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>. DAD attribute names are case-sensitive.

Use the DBMS_EPG package to perform these embedded PL/SQL gateway configurations:

1. Create a DAD with the DBMS_EPG.CREATE_DAD procedure.
2. Set DAD attributes with the DBMS_EPG.SET_DAD_ATTRIBUTE procedure.

All DAD attributes are optional. If you do not specify an attribute, it has its initial value.

Table 14-2 lists the embedded PL/SQL gateway attributes and the corresponding mod_plsql DAD parameters. Enumeration values in the "Legal Values" column are case-sensitive.

Table 14-2 Mapping Between mod_plsql and Embedded PL/SQL Gateway DAD Attributes

mod_plsql DAD Attribute	Embedded PL/SQL Gateway DAD Attribute	Multiple Occurrences	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
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
PlsqlRequestValidationFunction	request-validation-function	No	String
PlsqlSessionCookieName	session-cookie-name	No	String

Table 14–2 (Cont.) Mapping Between mod_plsql and Embedded PL/SQL Gateway DAD Attributes

mod_plsql DAD Attribute	Embedded PL/SQL Gateway DAD Attribute	Multiple Occurrences	Legal Values
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 default values of the DAD attributes are sufficient for most users of the embedded gateway. mod_plsql users do not need these attributes:

- PlsqlDatabasePassword
- PlsqlDatabaseConnectionString (because the embedded gateway does not support logon to external databases)

Like the DAD attributes, the global configuration parameters are optional. [Table 14–3](#) describes the DBMS_EPG global attributes and the corresponding mod_plsql global parameters.

Table 14–3 Mapping Between mod_plsql and Embedded PL/SQL Gateway Global Attributes

mod_plsql DAD Attribute	Embedded PL/SQL Gateway DAD Attribute	Multiple Occurrences	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

14.4.3.2 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: To serve a PL/SQL web application on the Internet but maintain the database behind a firewall, do not use the embedded PL/SQL gateway to run the application; use mod_plsql.

Use the DBMS_EPG package to configure database authentication.

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: Examples](#)
- [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](#)

14.4.3.2.1 Configuring Static Authentication with DBMS_EPG Static authentication is for the `mod_plsql` user who stores database user names and passwords in the DAD so that the browser user is not required to enter database authentication information.

To configure 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).
2. Create the DAD. For example, this 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, this 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');
```

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 use this command to assign these privileges to the DAD:

```
EXEC DBMS_EPG.AUTHORIZE_DAD('HR_DAD');
```

Note: 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. To use static authentication with the embedded PL/SQL gateway, first unlock this account.

14.4.3.2.2 Configuring Dynamic Authentication with DBMS_EPG

Dynamic authentication is for the `mod_plsql` user who does not store database user names 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).
2. Create the DAD. For example, this 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 prompts the user to enter the username and password for this account when accessing the DAD. For example, this 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');
```

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.

14.4.3.2.3 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.
2. Create the DAD. For example, this procedure creates a DAD invoked `HR_DAD` and maps the virtual path to `/hrweb/`:

```
EXEC DBMS_EPG.CREATE_DAD('HR_DAD', '/hrweb/*');
```

- Set the DAD attribute database-username to ANONYMOUS. For example:

```
EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('HR_DAD', 'database-username', 'ANONYMOUS');
```

Both database-username and ANONYMOUS are case-sensitive.

You need not authorize the embedded gateway to use ANONYMOUS privileges to access database objects, because ANONYMOUS has no system privileges and owns no database objects.

14.4.3.2.4 Determining the Authentication Mode of a DAD If you know the name of a DAD, then the authentication mode for this DAD depends on these 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 one that the user authorized to use the DAD?

[Table 14-4](#) shows how the answers to the preceding questions determine the authentication mode.

Table 14-4 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 run 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. The DAD authorized might not exist.

See Also: *Oracle Database Reference* for more information about the DBA_EPG_DAD_AUTHORIZATION view

14.4.3.2.5 Creating and Configuring DADs: Examples [Example 14-1](#) does this:

- Creates a DAD with static authentication for database user HR and assigns it the privileges of the HR account, which then authorizes it.
- Creates a DAD with dynamic authentication that is not restricted to any user.
- Creates a DAD with dynamic authentication that is restricted to the HR account.

Example 14-1 Creating and Configuring DADs

```
-----
-- DAD with static authentication
-----

CONNECT SYSTEM AS SYSDBA
PASSWORD: password
EXEC DBMS_EPG.CREATE_DAD('Static_Auth_DAD', '/static/*');
EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('Static_Auth_DAD', 'database-username', 'HR');
GRANT EXECUTE ON DBMS_EPG TO HR;

-- Authorization
CONNECT HR
PASSWORD: password
EXEC DBMS_EPG.AUTHORIZE_DAD('Static_Auth_DAD');

-----
-- DAD with dynamic authentication
-----

CONNECT SYSTEM AS SYSDBA
PASSWORD: password
EXEC DBMS_EPG.CREATE_DAD('Dynamic_Auth_DAD', '/dynamic/*');

-----
-- DAD with dynamic authentication restricted
-----

EXEC DBMS_EPG.CREATE_DAD('Dynamic_Auth_DAD_Restricted', '/dynamic/*');
EXEC DBMS_EPG.SET_DAD_ATTRIBUTE
  ('Dynamic_Auth_DAD_Restricted', 'database-username', 'HR');
```

The creation and authorization of a DAD are independent; therefore you can:

- Authorize a DAD that does not exist (it can be created later)
- Authorize a DAD for which you are not the user (however, the authorization does not take effect until the DAD database-user attribute is changed to your username)

Example 14-2 creates a DAD with static authentication for database user HR and assigns it the privileges of the HR account. Then:

- Instead of authorizing that DAD, the database user HR authorizes a nonexistent DAD.
Although the user might have done this by mistake, no error occurs, because the nonexistent DAD might be created later.
- The database user OE authorizes the DAD (whose database-user attribute is set to HR).
No error occurs, but the authorization does not take effect until the DAD database-user attribute is changed to OE.

Example 14-2 Authorizing DADs to be Created or Changed Later

```
REM Create DAD with static authentication for database user HR

CONNECT SYSTEM AS SYSDBA
PASSWORD: password
EXEC DBMS_EPG.CREATE_DAD('Static_Auth_DAD', '/static/*');
```

```

EXEC DBMS_EPG.SET_DAD_ATTRIBUTE('Static_Auth_DAD', 'database-username', 'HR');

GRANT EXECUTE ON DBMS_EPG TO HR;

REM Database user HR authorizes DAD that does not exist

CONNECT HR
PASSWORD: password
EXEC DBMS_EPG.AUTHORIZE_DAD('Static_Auth_DAD_Typo');

REM Database user OE authorizes DAD with database-username 'HR'

CONNECT OE
PASSWORD: password
EXEC DBMS_EPG.AUTHORIZE_DAD('Static_Auth_DAD');

```

14.4.3.2.6 Determining the Authentication Mode for a DAD: Example Example 14-3 creates a PL/SQL procedure, `show_dad_auth_status`, which accepts the name of a DAD and reports its authentication mode. If the specified DAD does not exist, the procedure exits with an error.

Example 14-3 Determining the Authentication Mode for a DAD

```

CREATE OR REPLACE PROCEDURE show_dad_auth_status (p_dadname VARCHAR2) IS
  v_daduser VARCHAR2(32);
  v_cnt      PLS_INTEGER;
BEGIN
  -- Determine DAD user
  v_daduser := DBMS_EPG.GET_DAD_ATTRIBUTE(p_dadname, 'database-username');

  -- Determine whether DAD authorization exists for 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 DAD user, authentication mode is static
  IF (v_cnt > 0) THEN
    DBMS_OUTPUT.PUT_LINE (
      '|| p_dadname ||
      '|| p_dadname ||
      ' is set up for static authentication for user '|| v_daduser || '.
    );
    RETURN;
  END IF;

  -- If no DAD authorization exists for DAD user, authentication mode is dynamic

  -- Determine whether dynamic authentication is restricted to particular user
  IF (v_daduser IS NOT NULL) THEN
    DBMS_OUTPUT.PUT_LINE (
      '|| p_dadname ||
      '|| p_dadname ||
      ' is set up for dynamic authentication for user '|| v_daduser || ' only.');
  ELSE
    DBMS_OUTPUT.PUT_LINE (
      '|| p_dadname ||
      '|| p_dadname ||
      ' is set up for dynamic authentication for any user.');
  END IF;
END;
/

```

Assume that you have run the script in [Example 14-1](#) to create and configure various DADs. The output is:

```
SET SERVEROUTPUT ON;
BEGIN
    show_dad_auth_status('Static_Auth_DAD');
END;
/
'Static_Auth_DAD' is set up for static authentication for user 'HR'.
```

14.4.3.2.7 Determining the Authentication Mode for All DADs: Example The anonymous block in [Example 14-4](#) reports the authentication modes of all registered DADs. It invokes the show_dad_auth_status procedure from [Example 14-3](#).

Example 14-4 Showing the Authentication Mode for All DADs

```
DECLARE
    v_dad_names DBMS_EPG.VARCHAR2_TABLE;
BEGIN
    DBMS_OUTPUT.PUT_LINE
        ('----- Authorization Status for All DADs -----');
    DBMS_EPG.GET_DAD_LIST(v_dad_names);

    FOR i IN 1..v_dad_names.count LOOP
        show_dad_auth_status(v_dad_names(i));
    END LOOP;
END;
/
```

If you have run the script in [Example 14-1](#) to create and configure various DADs, the output of [Example 14-4](#) is:

```
----- Authorization Status for All DADs -----
'Static_Auth_DAD' is set up for static auth for user 'HR'.
'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.
```

14.4.3.2.8 Showing DAD Authorizations that Are Not in Effect: Example The anonymous block in [Example 14-5](#) reports DAD authorizations that are *not* in effect. A DAD authorization is not in effect in either of these situations:

- The user who authorizes the DAD is not the user specified by the database-username attribute of the DAD
- The user authorizes a DAD that does not exist

Example 14-5 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
    DBMS_OUTPUT.PUT_LINE
        ('----- DAD Authorizations Not in Effect -----');
    DBMS_EPG.GET_DAD_LIST(v_dad_names);

    FOR r IN (SELECT * FROM DBA_EPG_DAD_AUTHORIZATION) LOOP  -- Outer loop
        v_dad_found := FALSE;
        FOR i IN 1..v_dad_names.count LOOP  -- Inner loop
```

```

        IF (r.DAD_NAME = v_dad_names(i)) THEN
            v_dad_user :=
                DBMS_EPG.GET_DAD_ATTRIBUTE(r.DAD_NAME, 'database-username');

            -- Is database-username the user for whom DAD is authorized?
            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 DAD user is ' ||
                    ''' || v_dad_user || ''');
            END IF;
            v_dad_found := TRUE;
            EXIT; -- Inner loop
        END IF;
    END LOOP; -- Inner loop

    -- Does 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; -- Outer loop
END;
/

```

If you have run the script in [Example 14-2](#) to create and configure various DADs, the output of [Example 14-5](#) (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 DAD user is 'HR'.
DAD authorization of 'Static_Auth_DAD_Typo' by user 'HR' is not in effect
    because DAD does not exist.

```

14.4.3.2.9 Examining Embedded PL/SQL Gateway Configuration When you are connected to the database as a user with system privileges, this script helps you examine the configuration of the embedded PL/SQL gateway:

```
$ORACLE_HOME/rdbms/admin/epgstat.sql
```

[Example 14-6](#) shows the output of the epgstat.sql script for [Example 14-1](#) when the ANONYMOUS account is locked.

Example 14-6 epgstat.sql Script Output for Example 14-1

Command to run script:

```
@$ORACLE_HOME/rdbms/admin/epgstat.sql
```

Result:

```
+-----+
| XDB protocol ports:          |
| XDB is listening for the protocol |
| when the protocol port is nonzero. |
+-----+
```

HTTP Port FTP Port

```

----- -----
0      0

1 row selected.

+-----+
| DAD virtual-path mappings |
+-----+

Virtual Path          DAD Name
-----
/dynamic/*           Dynamic_Auth_DAD_Restricted
/static/*            Static_Auth_DAD

2 rows selected.

+-----+
| DAD attributes |
+-----+

DAD Name      DAD Param          DAD Value
-----
Dynamic_Auth  database-username  HR
_DAD_Restricted

Static_Auth_   database-username  HR
DAD

2 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   HR                Dynamic
Dynamic_Auth_DAD_Res  HR             Dynamic Restricted
tricted

Static_Auth_DAD   HR                Static

3 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 & LOCKED

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?
-----
false

1 row selected.

```

14.4.4 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 must 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 this 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`.

These topics documented in *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

14.4.5 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.

14.4.6 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. Also, the embedded version of the gateway does not support these features:

- Dynamic HTML caching
- System monitoring
- Authentication modes other than Basic

For information about authentication modes, see *Oracle HTTP Server mod_plsql User's Guide*.

14.4.7 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 run the program follow these steps:

1. Log on to the database as a user with `ALTER USER` privileges and ensure 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 this 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.

To determine which users and roles were granted the `XDADMIN` role, query the data dictionary:

```
SELECT *
  FROM DBA_ROLE_PRIVS
 WHERE GRANTED_ROLE = 'XDBADMIN';
```

3. Create the DAD. For example, this procedure creates a DAD invoked `HR_DAD` and maps the virtual path to `/plsql/`:

```
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, this 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');
```

The attribute `database-username` is case-sensitive.

5. Grant `EXECUTE` privilege to the database user whose privileges must be used by the DAD (so that he or she can authorize the DAD). For example:

```
GRANT EXECUTE ON DBMS_EPG TO HR;
```

6. 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, `HR`).

7. 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');
```

8. Create a sample PL/SQL stored procedure invoked `print_employees`. This 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;
  /

```

9. Ensure that the Oracle Net listener can accept HTTP requests. You can determine the status of the listener on Linux and UNIX by running this command at the system prompt:

```
lsnrctl status | grep HTTP
```

Output (reformatted from a single line to multiple lines from page size constraints):

```
(DESCRIPTION=
  (ADDRESS=(PROTOCOL=tcp) (HOST=example.com) (PORT=8080))
  (Presentation=HTTP)
  (Session=RAW)
)
```

If you do not see the HTTP service started, then you can add these lines to your 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
```

10. Run the `print_employees` program from your web browser. For example, you can use this 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://example.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.

14.5 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 14-7](#) shows how to generate a simple HTML page by calling the HTP functions that correspond to each HTML tag.

Example 14-7 Using HTP Functions to Generate HTML Tags

```
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 both text and tags. [Example 14-8](#) illustrates this technique.

Example 14-8 Using HTP.PRINT to Generate HTML Tags

```
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;
/
```

14.6 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 Option and Check Box 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](#)

14.6.1 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 these 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 14-9](#) shows a simple drop-down list.

Example 14-9 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>
```

14.6.2 Passing Option and Check Box Parameters from an HTML Form

Options pass either a null value (if none of the options in a group is checked), or the value specified on the option that is checked.

To specify a default value for a set of options, you can include the `CHECKED` attribute in an `INPUT` tag, or include a `DEFAULT` clause on the parameter within the stored subprogram. When setting up a group of options, be sure to include a choice that indicates "no preference", because after selecting a option, the user can select a different one, but 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.

Check boxes need special handling, because your stored subprogram might receive a null value, a single value, or multiple values:

All the check boxes with the same `NAME` attribute comprise a check box group. If none of the check boxes in a group is checked, the stored subprogram receives a null value for the corresponding parameter.

If one check box in a group is checked, the stored subprogram receives a single `VARCHAR2` parameter.

If multiple check boxes in a group are checked, the stored subprogram receives a parameter with the PL/SQL type `TABLE OF VARCHAR2`. You must declare a type like `TABLE OF VARCHAR2`, 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
    http.print('<p>Check Box value: ' || checkboxes(i));
  END LOOP;
END;
/
```

14.6.3 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 a client-side JavaScript function, 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 after reaching a length limit.
- 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 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.

The procedure in [Example 14-10](#) accepts two strings as input. The first time the procedure 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 input, filling in the original values for the user.

Example 14-10 Passing Entry-Field Parameters from an HTML Form

```

DROP TABLE name_zip_table;
CREATE TABLE name_zip_table (
    name      VARCHAR2(100),
    zipcode   NUMBER
);

-- Store a name and associated zip code in the database.

CREATE OR REPLACE PROCEDURE associate_name_with_zipcode
    (name VARCHAR2 := NULL,
     zip  VARCHAR2 := NULL)
AS
BEGIN
    -- Each entry field must contain a value. 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
        INSERT INTO name_zip_table (name, zipcode) VALUES (name, zip);

        HTP.PRINT('<p>The person ' || HTP.ESCAPE_SC(name) ||
                  ' has the zip code ' || HTP.ESCAPE_SC(zip) || '.');
    END IF;

    -- If user entered incomplete or incorrect data, show 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 reenter data. Fill all fields,
                  and use 6-digit zip code.</b>');
    END IF;

    -- If user entered no data or incorrect data, show error message
    -- & make form invoke same procedure to check input values.

    HTP.FORMOPEN('HR.associate_name_with_zipcode', 'GET');
    HTP.PRINT('<p>Enter your name:</td>');

    HTP.PRINT('<td valign=center><input type=text name=name value=' || |
              HTP.ESCAPE_SC(name) || '>');

    HTP.PRINT('<p>Enter your zip code:</td>');

    HTP.PRINT('<td valign=center><input type=text name=zip value=' || |
              HTP.ESCAPE_SC(zip) || '>');

```

```
  HTP.FORMSUBMIT(NULL, 'Submit');
  HTP.FORMCLOSE;
END;
/
```

14.6.4 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 option 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.

14.6.5 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*

14.6.6 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 eliminate 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 makes a single selection, and the confirmation step of the `Submit` button is not essential.

14.6.7 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 check boxes, options, or list items with none checked, or a `VALUE` parameter of "" (empty quotation marks).

Regardless of any validation you do on the client side, use code stored subprograms to handle the possibility that some parameters are null:

- Specify an initial value in all parameter declarations, to prevent an exception when the stored subprogram is invoked with a missing form parameter. You can set the initial value to zero for numeric values (when that makes sense), and to `NULL` when you want to check whether the user specifies a value.
- Before using an input parameter value that has the initial value `NULL`, 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.

14.6.8 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 repeatedly.

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 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 after 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 website.

See Also: The Oracle Application Server documentation set at:

<http://www.oracle.com/technetwork/indexes/documentation/index.html>

14.7 Performing Network Operations in PL/SQL Subprograms

Oracle provides packages that allow PL/SQL subprograms to perform these network operations:

- [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](#)

Internet Protocol version 6 (IPv6) Support

As of Oracle Database 11g Release 2 (11.2.0.1), PL/SQL network utility packages support IPv6 addresses. The package interfaces have not changed: Any interface parameter that expects a network host accepts an IPv6 address in string form, and any interface that returns an IP address can return an IPv6 address.

However, applications that use network addresses might need small changes, and recompilation, to accommodate IPv6 addresses. An IPv6 address has 128 bits, while an IPv4 address has 32 bits. In a URL, an IPv6 address must be enclosed in brackets. For example:

`http://[2001:0db8:85a3:08d3:1319:8a2e:0370:7344]/`

See Also:

- [Oracle Database Net Services Administrator's Guide](#) for detailed information about IPv6 support in Oracle Database
- [Oracle Database PL/SQL Packages and Types Reference](#) for information about IPv6 support in specific PL/SQL network utility packages

14.7.1 Sending E-Mail from PL/SQL

Using the UTL_SMTP package, a PL/SQL subprogram can send e-mail, as in [Example 14-11](#).

Example 14-11 Sending E-Mail from PL/SQL

```
CREATE OR REPLACE PROCEDURE send_test_message
IS
  mailhost  VARCHAR2(64) := 'mailhost.example.com';
  sender    VARCHAR2(64) := 'me@example.com';
  recipient VARCHAR2(64) := 'you@example.com';
  mail_conn UTL_SMTP.CONNECTION;
BEGIN
  mail_conn := UTL_SMTP.OPEN_CONNECTION(mailhost, 25); -- 25 is the port
  UTL_SMTP.HELO(mail_conn, mailhost);
  UTL_SMTP.MAIL(mail_conn, sender);
  UTL_SMTP.RCPT(mail_conn, recipient);

  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);

  /* If message were in single string, open_data(), write_data(),
   and close_data() could be in a single call to data(). */

  UTL_SMTP.QUIT(mail_conn);
EXCEPTION
  WHEN OTHERS THEN
    -- Insert error-handling code here
    RAISE;
END;
```

/

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

14.7.2 Getting a Host Name or Address from PL/SQL

Using the UTL_INADDR package, a PL/SQL subprogram can determine the host name of the local system or the IP address of a given host name.

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

14.7.3 Using TCP/IP Connections from PL/SQL

Using the UTL_TCP package, a PL/SQL subprogram can open TCP/IP connections to systems on the network, and read or write to the corresponding sockets.

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

14.7.4 Retrieving HTTP URL Contents from PL/SQL

Using the UTL_HTTP package, a PL/SQL subprogram can:

- Retrieve the contents of an HTTP URL
The contents are usually in the form of HTML-tagged text, but might be any kind of file that can be downloaded from a web server (for example, plain text or a JPEG image).
- Control HTTP session details (such as headers, cookies, redirects, proxy servers, IDs and passwords for protected sites, and CGI parameters)
- Speed up multiple accesses to the same website, using HTTP 1.1 persistent connections

A PL/SQL subprogram can construct and interpret URLs for use with the UTL_HTTP package by using the functions UTL_URL.ESCAPE and UTL_URL.UNESCAPE.

The PL/SQL procedure in [Example 14–12](#) uses the UTL_HTTP package to retrieve the contents of an HTTP URL.

Example 14–12 Retrieving HTTP URL Contents from PL/SQL

```
CREATE OR REPLACE PROCEDURE show_url
  (url      IN VARCHAR2,
   username IN VARCHAR2 := NULL,
   password IN VARCHAR2 := 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 do not need the proxy host.
```

```
UTL_HTTP.SET_PROXY('proxy.example.com', 'corp.example.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 yourself.
-- Some sites serve special pages for particular browsers.
UTL_HTTP.SET_HEADER(req, 'User-Agent', 'Mozilla/4.0');

-- Specify user ID and password for pages that require them.
IF (username IS NOT NULL) THEN
    UTL_HTTP.SET_AUTHENTICATION(req, username, password);
END IF;

-- Start receiving the HTML text.
resp := UTL_HTTP.GET_RESPONSE(req);

-- Show status codes and reason phrase of 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 page is password protected
    -- and you 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 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 for realm ' || my_realm ||
                ' for the web page.');
        END IF;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Check the URL.');
    END IF;
ELSE
    DBMS_OUTPUT.PUT_LINE('Check if the website is up.');
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 website is up.');
    UTL_HTTP.END_RESPONSE(resp);
    RETURN;
END IF;
```

```

-- 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;

-- Read lines until none 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;
/

```

This block shows examples of calls to the procedure in [Example 14-12](#), but the URLs are for nonexistent pages. Substitute URLs from your own web server.

```

BEGIN
  show_url('http://www.oracle.com/no-such-page.html');
  show_url('http://www.oracle.com/protected-page.html');
  show_url
    ('http://www.oracle.com/protected-page.html', 'username', 'password');
END;
/

```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the UTL_HTTP package
- *Oracle Database PL/SQL Packages and Types Reference* for detailed information about UTL_URL.ESCAPE and UTL_URL.UNESCAPE

14.7.5 Using Tables, Image Maps, Cookies, and CGI Variables from PL/SQL

Using packages supplied by Oracle, and the `mod_plsql` plug-in of Oracle HTTP Server (OHS), a PL/SQL subprogram 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 perform other typical web operations.

Documentation for these packages is not part of the database documentation library. The location of the documentation depends on your application server. 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;

```


Using Continuous Query Notification (CQN)

Continuous Query Notification (CQN) lets an application 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, regardless of whether 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 Oracle Call Interface (OCI). If you use the PL/SQL interface, the notification handler is a server-side PL/SQL stored procedure; if you use OCI, 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:

- [About Object Change Notification \(OCN\)](#)
- [About Query Result Change Notification \(QRCN\)](#)
- [Events that Generate Notifications](#)
- [Notification Contents](#)
- [Good Candidates for CQN](#)
- [Creating CQN Registrations](#)
- [Using PL/SQL to Create CQN Registrations](#)
- [Using OCI to Create CQN Registrations](#)
- [Querying CQN Registrations](#)
- [Interpreting Notifications](#)

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*.

15.1 About 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, regardless of whether the result of the query changed.

For example, if an application registers the query in [Example 15-1](#) 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 15-1 Query to be Registered for Change Notification

```
SELECT EMPLOYEE_ID, SALARY
  FROM EMPLOYEES
 WHERE DEPARTMENT_ID = 10;
```

15.2 About 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 15-1](#) for QRCN, the database sends the application a QRCN only if the query result set changes; that is, if one of these data manipulation language (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 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](#)

15.2.1 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 15-1](#) for QRCN, that employee 201 is in department 10, and that these 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 [Section 15.7.5.1](#).

15.2.2 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 15-2](#) is too complex for QRCN in guaranteed mode because it contains the aggregate function `SUM`.

Example 15-2 Query Too Complex for QRCN in Guaranteed Mode

```
SELECT SUM(SALARY)
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 20;
```

In best-effort mode, CQN registers this simpler version of the query in [Example 15-2](#):

```
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 from 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:

- Minimizes the OLTP response overhead that is from 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.

The overhead minimization of best-effort mode infrequently causes false positives, even for queries that CQN does not simplify. For example, consider the query in [Example 15-1](#) and the transaction in [Section 15.2.1](#). 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 15-3](#) is too complex for QRCN in guaranteed mode because it includes a subquery.

Example 15-3 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 15-3](#) 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 [Section 15.7.5.2](#).

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.

15.3 Events that Generate Notifications

These events generate notifications:

- [Committed DML Transactions](#)

- [Committed DDL Statements](#)
- [Deregistration](#)
- [Global Events](#)

15.3.1 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 [Section 15.7.2.3](#).

15.3.2 Committed DDL Statements

For both OCN and QRCN, these data definition language (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 an object is created with the same name, and in the same schema, as the dropped object, the created object is not associated with OCN registrations that were associated with the dropped object.

When the notification type is QRCN:

- The notification includes:
 - 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 deregistered.

For example, suppose that this query is registered for QRCN:

```
SELECT COL1 FROM TEST_TABLE
  WHERE COL2 = 1;
```

Suppose that TEST_TABLE has this schema:

```
(COL1 NUMBER, COL2 NUMBER, COL3 NUMBER)
```

This DDL statement, when committed, invalidates the query and causes it to be removed from the registration:

```
ALTER TABLE DROP COLUMN COL2;
```

15.3.3 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 [Section 15.7.3](#).

A notification is not generated when a client application performs an explicit deregistration.

15.3.4 Global Events

The global events EVENT_STARTUP and EVENT_SHUTDOWN generate notifications.

In an Oracle RAC environment, these events generate notifications:

- EVENT_STARTUP when the first instance of the database starts
- 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

15.4 Notification Contents

A notification contains some or all of this information:

- Type of event, which is one of:
 - Startup
 - Object change
 - Query result change
 - Deregistration
 - Shutdown
- Registration ID of affected registration
- Names of changed objects
- If ROWID option was specified, ROWIDs 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.

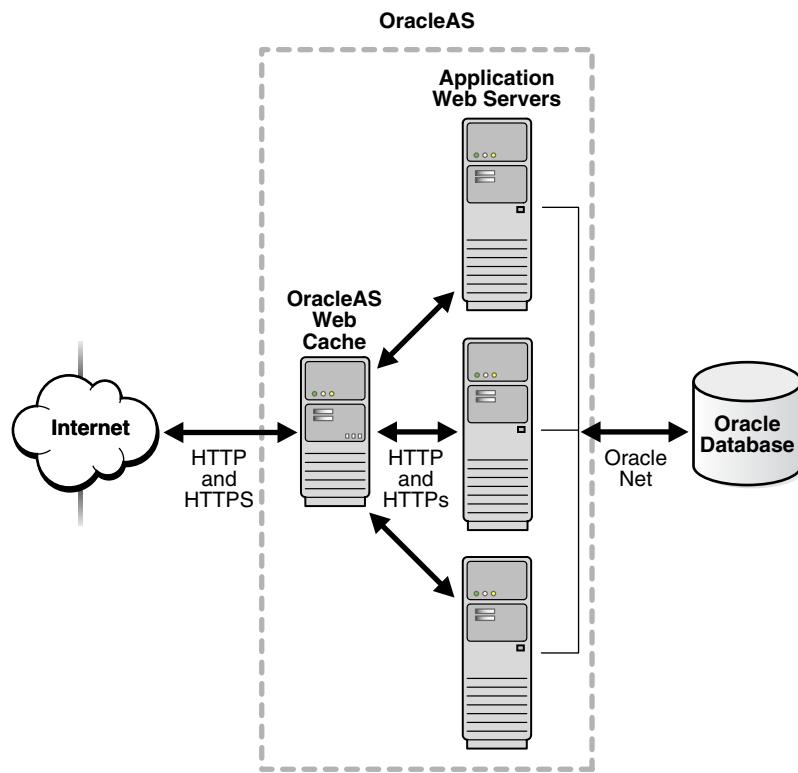
15.5 Good Candidates for CQN

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 rerunning the registered queries.

An example of such an application is a web forum. Because its users need not view 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 to refresh the cache.

[Figure 15-1](#) illustrates a typical scenario in which the database serves data that is cached in the middle tier and then accessed over the Internet.

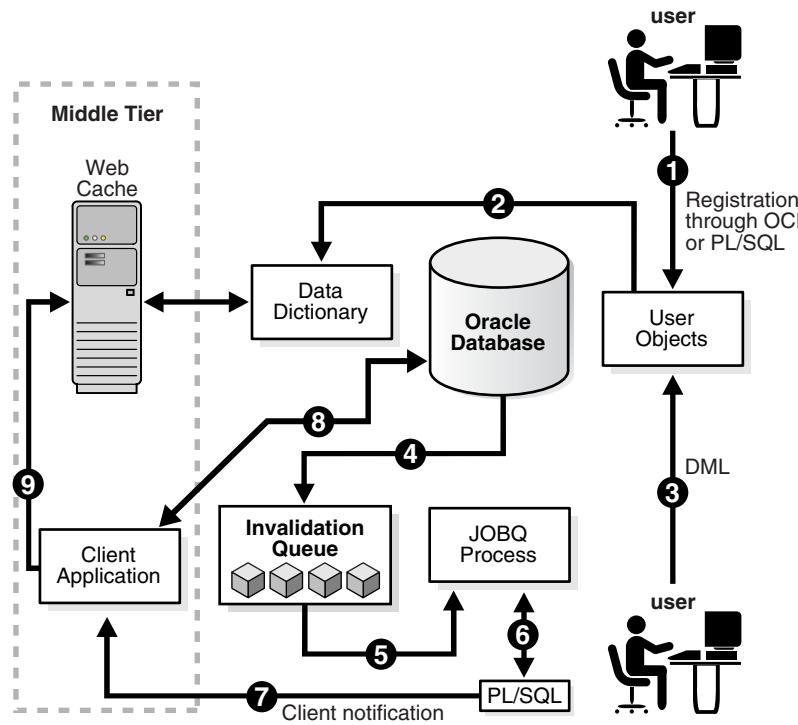
Figure 15–1 Middle-Tier Caching



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, the 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 15–2](#) illustrates the process by which middle-tier web clients receive and process notifications.

Figure 15–2 Basic Process of Continuous Query Notification (CQN)



Explanation of steps in Figure 15–2 (if 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. The database adds a message that describes the change to an internal queue.
5. The database notifies a JOBJQ background process of a notification message.
6. The JOBJQ process runs the stored procedure specified by the client application. In this example, JOBJQ 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 data.

15.6 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 [Section 15.1](#) and [Section 15.2](#).

To create a CQN registration, you can use one of two interfaces:

- PL/SQL interface

If you use the PL/SQL interface, the notification handler is a server-side PL/SQL stored procedure. See [Section 15.7](#).

- Oracle Call Interface (OCI)

If you use OCI, the notification handler is a client-side C callback procedure. See [Section 15.8](#).

After being created, a registration is stored in the 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 (from loss of privileges, for example).

15.7 Using PL/SQL to Create CQN Registrations

This section describes using PL/SQL to create CQN registrations. When you use the PL/SQL interface, the notification handler is a server-side PL/SQL stored procedure.

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](#)
- [Deleting Registrations](#)
- [Configuring CQN: Scenario](#)

15.7.1 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 run 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 [Section 15.7.6](#).

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_CQ_NOTIFICATION` package

15.7.2 CQN Registration Options

You can change the CQN registration defaults with the options summarized in [Table 15-1](#).

Table 15-1 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 value of the <code>ROWID</code> pseudocolumn for 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](#)

15.7.2.1 Notification Type Option

The notification types are OCN (described in [Section 15.1](#)) and QRCN (described in [Section 15.2](#)).

15.7.2.2 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 [Section 15.7.2.1](#).

The QRCN modes are guaranteed (described in [Section 15.2.1](#)) and best-effort (described in [Section 15.2.2](#)).

The default is guaranteed mode. For best-effort mode, specify `QOS_BEST EFFORT` in the `QOSFLAGS` attribute of the `CQ_NOTIFICATION$_REG_INFO` object.

15.7.2.3 ROWID Option

The `ROWID` option includes the value of the `ROWID` pseudocolumn (the rowid of the row) for each changed row in the notification. 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.

Note: When you update a row in a table compressed with Hybrid Columnar Compression (HCC), the `ROWID` of the row changes. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

From the `ROWID` information in the notification, the application can retrieve the contents of the changed rows by performing queries of this 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 from `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 [Section 15.7.2.6](#).

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.

15.7.2.4 Operations Filter Option (OCN Notification Type Only)

The Operations Filter option applies only when the notification type is OCN.

The Operations Filter option enables 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_NOTIFICATION` package, as follows:

Operation	Constant
INSERT	<code>DBMS_CQ_NOTIFICATION.INSERTOP</code>
UPDATE	<code>DBMS_CQ_NOTIFICATION.UPDATEOP</code>
DELETE	<code>DBMS_CQ_NOTIFICATION.DELETEOP</code>
ALTEROP	<code>DBMS_CQ_NOTIFICATION.ALTEROP</code>
DROPOP	<code>DBMS_CQ_NOTIFICATION.DROPOP</code>
UNKNOWNOP	<code>DBMS_CQ_NOTIFICATION.UNKNOWNOP</code>
All (default)	<code>DBMS_CQ_NOTIFICATION.ALL_OPERATIONS</code>

To specify multiple operations, use bitwise OR. For example:

```
DBMS_CQ_NOTIFICATION.INSERTOP + DBMS_CQ_NOTIFICATION.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

15.7.2.5 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 [Section 15.7.2.6](#).

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 `ROWIDS`, even if you also specified `QOS_ROWIDS`.

15.7.2.6 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:

Attribute	Description
NTFN_GROUPING_CLASS	Specifies the class by which to group notifications. 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.
NTFN_GROUPING_TYPE	<p>Specifies the type of grouping, which is either of:</p> <ul style="list-style-type: none"> <li data-bbox="768 572 1380 635">■ DBMS_CQ_NOTIFICATION.NTFN_GROUPING_TYPE_SUMMARY: All notifications in the group are summarized into a single notification. <li data-bbox="768 650 1380 713">Note: The single notification does not include ROWIDS, even if you specified the ROWID option. <li data-bbox="768 728 1380 794">■ 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 .

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.

15.7.2.7 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.

15.7.2.8 Purge-on-Notify and Timeout Options

By default, a CQN registration survives until the application that created it explicitly deregisters it or until the database implicitly purges it (from 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.

15.7.3 Prerequisites for Creating CQN Registrations

These are prerequisites for creating CQN registrations:

- You must have these privileges:
 - EXECUTE privilege on the DBMS_CQ_NOTIFICATION package, whose subprograms you use to create a registration
 - CHANGE NOTIFICATION system privilege
 - READ or SELECT privilege on each object to be registered
- Loss of privileges on an object associated with a registered query generates a notification—see [Section 15.3.3](#).
- 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.

15.7.4 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:

- 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.

15.7.5 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 [Section 15.2.1](#) and [Section 15.2.2](#).)

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](#)

15.7.5.1 Queries that Can Be Registered for QRCN in Guaranteed Mode

To be registered for QRCN in guaranteed mode, a query must conform to these rules:

- Every column that it references is either a NUMBER data type or a VARCHAR2 data type.
- Arithmetic operators in column expressions are limited to these binary operators, and their operands are columns with numeric data types:
 - + (addition)
 - - (subtraction, not unary minus)
 - * (multiplication)
 - / (division)
- Comparison operators in the predicate are limited to:
 - < (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 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 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 [Section 15.2.2](#).
-

15.7.5.2 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 generates 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 these:
 - String functions (such as SUBSTR, LTRIM, and RTRIM)
 - Arithmetic functions (such as TRUNC, ABS, and SQRT)

For a list of 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 these:
 - Bitmap join, domain, or function-based 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

15.7.5.3 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 [Section 15.7.5.1](#)
- 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 SQL Tuning Guide*.

15.7.6 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. Run the queries to register. (Do not run 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

15.7.6.1 Creating a PL/SQL Notification Handler

The PL/SQL stored procedure that you create to serve as the notification handler must have this 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, *ntfnd\$,* to the notification handler, *proc_name*, which handles the notification according to its application requirements. (This is step 6 in [Figure 15–2](#).)

Note: The notification handler runs 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.

15.7.6.2 Creating a CQ_NOTIFICATION\$_REG_INFO Object

An object of type CQ_NOTIFICATION\$_REG_INFO specifies the notification handler that the database runs when a registered objects changes. In SQL*Plus, you can view its type attributes by running this statement:

```
DESC CQ_NOTIFICATION$_REG_INFO
```

[Table 15–2](#) describes the attributes of SYS.CQ_NOTIFICATION\$_REG_INFO.

Table 15–2 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 <i>schema_name.procedure_name</i> , for example, <i>hr.dcn_callback</i> .
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 15–3 . To specify multiple quality-of-service flags, 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 deregisters it. Can be combined with the QOSFLAGS attribute with its QOS_DEREG_NFY flag.

Table 15–2 (Cont.) Attributes of CQ_NOTIFICATION\$_REG_INFO

Attribute	Description
OPERATIONS_FILTER	<p>Applies only to OCN (described in Section 15.1). Has no effect if you specify the QOS_FLAGS attribute with its QOS_QUERY flag.</p> <p>Filters messages based on types of SQL statement. You can specify these constants in the DBMS_CQ_NOTIFICATION package:</p> <ul style="list-style-type: none"> ▪ 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 <p>You can specify a combination of operations with a bitwise OR. For example: DBMS_CQ_NOTIFICATION.INSERTOP + DBMS_CQ_NOTIFICATION.DELETEOP.</p>
TRANSACTION_LAG	<p>Deprecated. To implement flow-of-control notifications, use the NTFN_GROUPING_* attributes.</p> <p>Applies only to OCN (described in Section 15.1). Has no effect if you specify the QOS_FLAGS attribute with its QOS_QUERY flag.</p> <p>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. The database tracks intervening changes at an object granularity and bundles the changes along with the notification. Thus, the client does not lose intervening changes.</p> <p>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, Oracle recommends 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.</p> <p>If you specify TRANSACTION_LAG, then the ROWID level granularity is unavailable in the notification messages even if you specified QOS_ROWIDS during registration.</p>
NTFN_GROUPING_CLASS	Specifies the class by which to group notifications. 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.

Table 15-2 (Cont.) Attributes of CQ_NOTIFICATION\$_REG_INFO

Attribute	Description
NTFN_GROUPING_TYPE	Specifies either of these types of grouping: <ul style="list-style-type: none"> ▪ 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 <code>NULL</code> , it defaults to the current system-generated time.
NTFN_GROUPING_REPEAT_COUNT	Specifies how many times to repeat the notification. Set to <code>DBMS_CQ_NOTIFICATION.NTFN_GROUPING_FOREVER</code> to receive notifications for the life of the registration. To receive at most <i>n</i> notifications during the life of the registration, set to <i>n</i> .

The quality-of-service flags in [Table 15-3](#) are constants in the `DBMS_CQ_NOTIFICATION` package. You can specify them with the `QOS_FLAGS` attribute of `CQ_NOTIFICATION$_REG_INFO` (see [Table 15-2](#)).

Table 15-3 Quality-of-Service Flags

Flag	Description
<code>QOS_DEREG_NFY</code>	Purges the registration after the first notification.
<code>QOS_RELIABLE</code>	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.
<code>QOS_ROWIDS</code>	Includes the <code>ROWID</code> of each changed row in the notification.
<code>QOS_QUERY</code>	Registers queries for QRCN, described in Section 15.2 . If a query cannot be registered for QRCN, an error is generated at registration time, unless you also specify <code>QOS_BEST EFFORT</code> . Default: Queries are registered for OCN, described in Section 15.1
<code>QOS_BEST EFFORT</code>	Used with <code>QOS_QUERY</code> . 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 Section 15.2.2 . To see which queries were simplified, query the static data dictionary view <code>DBA_CQ_NOTIFICATION_QUESTIONS</code> or <code>USER_CQ_NOTIFICATION_QUESTIONS</code> . These views give the <code>QUERYID</code> and the text of each registered query. Default: Queries are registered for QRCN in guaranteed mode, described in Section 15.2.1

Suppose that you must invoke the procedure `HR.dcn_callback` whenever a registered object changes. In [Example 15-4](#), 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 15–4 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;
/

```

15.7.6.3 Identifying Individual Queries in a Notification

Any query in a registered list of queries can cause a continuous query notification. 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;

```

Result:

EMPLOYEE_ID	SALARY	CQ_NOTIFICATION_QUERYID
200	2800	0

1 row selected.

When that query causes a notification, the notification includes the query ID.

15.7.6.4 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 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. Run the queries to register. (Do not run DML or DDL operations.)
4. Close the registration, using the DBMS_CQ_NOTIFICATION.REG_END function.

[Example 15–5](#) adds a query to an existing registration whose registration ID is 21.

Example 15–5 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
    -- Run query to be registered
    SELECT DEPARTMENT_ID
      FROM HR.DEPARTMENTS; -- register this query
  CLOSE v_cursor;
  -- Close registration
  DBMS_CQ_NOTIFICATION.REG_END;
END;
/

```

15.7.7 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, to avoid replicating a notification message for multiple recipients.

15.7.8 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 these:

- 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 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, this statement displays all registrations and registered objects for the current user:

```
SELECT REGID, TABLE_NAME FROM USER_CHANGE_NOTIFICATION_REGS;
```

- Runtime 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 run the procedure. The name of the trace file usually has this 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:

```
*****
Runtime 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 runtime errors occurred during the execution of the notification handler, create a very simple version of the notification handler to verify that you are 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 nfcnt (cnt NUMBER);
INSERT INTO nfcnt (cnt) VALUES(0);
COMMIT;
CREATE OR REPLACE PROCEDURE chnf_callback
  (ntfnds IN CQ_NOTIFICATION$_DESCRIPTOR)
IS
BEGIN
  UPDATE nfcnt 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.

15.7.9 Deleting Registrations

To delete a registration, call the procedure `DBMS_CQ_NOTIFICATION.DEREGISTER` with the registration ID as the parameter. For example, this 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.

15.7.10 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 and 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 [Section 15.7.10.1](#).
2. Register the queries on the HR.EMPLOYEES and HR.DEPARTMENTS tables for QRCN, as instructed in [Section 15.7.10.2](#).

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](#)

15.7.10.1 Creating a PL/SQL Notification Handler

Create a server-side stored PL/SQL procedure to process notifications as follows:

1. Connect to the database AS SYSDBA.
2. Grant the required privileges to HR:

```
GRANT EXECUTE ON DBMS_CQ_NOTIFICATION TO HR;
GRANT CHANGE NOTIFICATION TO HR;
```

3. Enable the JOB_QUEUE_PROCESSES parameter to receive notifications:

```
ALTER SYSTEM SET "JOB_QUEUE_PROCESSES"=4;
```

4. Connect to the database as a non-SYS user (such as HR).

5. Create database tables to hold records of notification events received:

```
-- Create table to record notification events.
DROP TABLE nfevents;
CREATE TABLE nfevents (
    regid      NUMBER,
    event_type NUMBER
);

-- Create table to record notification queries:
DROP TABLE nfqueries;
CREATE TABLE nfqueries (
    qid      NUMBER,
    qop      NUMBER
);

-- Create table to record changes to registered tables:
DROP TABLE nftablechanges;
CREATE TABLE nftablechanges (
    qid      NUMBER,
    table_name      VARCHAR2(100),
    table_operation NUMBER
);

-- Create table to record ROWIDs of changed rows:
DROP TABLE nfrowchanges;
CREATE TABLE nfrowchanges (
    qid      NUMBER,
    table_name VARCHAR2(100),
    row_id    VARCHAR2(2000)
```

);

6. Create the procedure HR.chnf_callback, as shown in [Example 15–6](#).

Example 15–6 Creating Server-Side PL/SQL Notification Handler

```

CREATE OR REPLACE PROCEDURE chnf_callback (
  ntnfnds 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 := ntnfnds.registration_id;
  event_type := ntnfnds.event_type;

  INSERT INTO nfevents (regid, event_type)
  VALUES (chnf_callback.regid, chnf_callback.event_type);

  numqueries :=0;

  IF (event_type = DBMS_CQ_NOTIFICATION.EVENT_QUERYCHANGE) THEN
    numqueries := ntnfnds.query_desc_array.count;

    FOR i IN 1..numqueries LOOP  -- loop over queries
      qid := ntnfnds.query_desc_array(i).queryid;
      qop := ntnfnds.query_desc_array(i).queryop;

      INSERT INTO nfqueries (qid, qop)
      VALUES(chnf_callback.qid, chnf_callback.qop);

      numtables := 0;
      numtables := ntnfnds.query_desc_array(i).table_desc_array.count;

      FOR j IN 1..numtables LOOP  -- loop over tables
        tbname :=
          ntnfnds.query_desc_array(i).table_desc_array(j).table_name;
        operation_type :=
          ntnfnds.query_desc_array(i).table_desc_array(j).Opflags;

        INSERT INTO nftablechanges (qid, table_name, table_operation)
        VALUES (
          chnf_callback.qid,
          tbname,
          operation_type
        );

        IF (bitand(operation_type, DBMS_CQ_NOTIFICATION.ALL_ROWS) = 0) THEN
          numrows := ntnfnds.query_desc_array(i).table_desc_array(j).numrows;
        ELSE
          numrows :=0;  -- ROWID info not available
        END IF;
      END LOOP;
    END LOOP;
  END IF;
END;

```

```

        END IF;

        -- Body of loop does not run when numrows is zero.
        FOR k IN 1..numrows LOOP  -- loop over rows
            Row_id :=
            ntfnds.query_desc_array(i).table_desc_array(j).row_desc_array(k).row_id;

            INSERT INTO nfrowchanges (qid, table_name, row_id)
            VALUES (chnf_callback.qid, tbname, chnf_callback.Row_id);

        END LOOP;  -- loop over rows
        END LOOP;  -- loop over tables
        END LOOP;  -- loop over queries
    END IF;
    COMMIT;
END;
/

```

15.7.10.2 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 15-7](#).

Example 15-7 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;

```

```

DBMS_CQ_NOTIFICATION.reg_end;
END;
/

```

View the newly created registration:

```

SELECT queryid, regid, TO_CHAR(querytext)
FROM user_cg_notification_queries;

```

Result is similar to:

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

Run this transaction, which changes the result of the query with QUERYID 22:

```

UPDATE DEPARTMENTS
SET DEPARTMENT_NAME = 'FINANCE'
WHERE department_name = 'IT';

```

The notification procedure chnf_callback (which you created in [Example 15–6](#)) runs.

Query the table in which notification events are recorded:

```
SELECT * FROM nfevents;
```

Result is similar to:

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;
```

Result is similar to:

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;
```

Result is similar to:

REGID	TABLE_NAME	ROWID
61	HR.DEPARTMENTS	AAANKdAABAALinAAF

15.8 Using OCI to Create CQN Registrations

This section describes using OCI to create CQN registrations. When you use OCI, the notification handler is a client-side C callback procedure.

Topics

- [Using OCI for Query Result Set Notifications](#)
- [Using OCI to Register a Continuous Query Notification](#)
- [Using OCI Subscription Handle Attributes for Continuous Query Notification](#)
- [Using OCI_ATTR_CQ_QUERYID Attribute](#)
- [Using OCI Continuous Query Notification Descriptors](#)
- [Demonstrating Continuous Query Notification in an OCI Sample Program](#)

See Also: *Oracle Call Interface Programmer's Guide* for more information about publish-subscribe notification in OCI

15.8.1 Using OCI for Query Result Set Notifications

To record QOS (quality of service flags) specific to continuous query (CQ) notifications, set the attribute `OCI_ATTR_SUBSCR_CQ_QOSFLAGS` on the subscription handle `OCI_HTYPE_SUBSCR`. To request that the registration is at query granularity, as opposed to object granularity, set the `OCI_SUBSCR_CQ_QOS_QUERY` flag bit on the attribute `OCI_ATTR_SUBSCR_CQ_QOSFLAGS`.

The pseudocolumn `CQ_NOTIFICATION_QUERY_ID` can be optionally specified to retrieve the query ID of a registered query. This does not automatically convert the granularity to query level. The value of the pseudocolumn on return is set to the unique query ID assigned to the query. The query ID pseudocolumn can be omitted for OCI-based registrations, in which case the query ID is returned as a `READ` attribute of the statement handle. (This attribute is called `OCI_ATTR_CQ_QUERYID`).

During notifications, the client-specified callback is invoked and the top-level notification descriptor is passed as an argument.

Information about the query IDs of the changed queries is conveyed through a special descriptor type called `OCI_DTYPE_CQDES`. A collection (`OCIColl`) of query descriptors is embedded inside the top-level notification descriptor. Each descriptor is of type `OCI_DTYPE_CQDES`. The query descriptor has the following attributes:

- `OCI_ATTR_CQDES_OPERATION` - can be one of `OCI_EVENT_QUERYCHANGE` or `OCI_EVENT_DEREG`.
- `OCI_ATTR_CQDES_QUERYID` - query ID of the changed query.
- `OCI_ATTR_CQDES_TABLE_CHANGES` - array of table descriptors describing DML operations on tables that led to the query result set change. Each table descriptor is of the type `OCI_DTYPE_TABLE_CHDES`.

See Also: [Section 15.8.5.1, "OCI_DTYPE_CHDES"](#)

15.8.2 Using OCI to Register a Continuous Query Notification

The calling session must have the `CHANGE NOTIFICATION` system privilege and `SELECT` privileges on all objects that it attempts to register. A registration is a persistent entity that is recorded in the database, and is visible to all instances of Oracle RAC. If the

registration was at query granularity, transactions that cause the query result set to change and commit in any instance of Oracle RAC generate notification.

If the registration was at object granularity, transactions that modify registered objects in any instance of Oracle RAC generate notification.

Queries involving materialized views or nonmaterialized views are *not* supported.

The registration interface employs a callback to respond to changes in underlying objects of a query and uses a namespace extension (DBCHANGE) to AQ.

The steps in writing the registration are:

1. Create the environment in `OCI_EVENTS` and `OCI_OBJECT` mode.
2. Set the subscription handle attribute `OCI_ATTR_SUBSCR_NAMESPACE` to namespace `OCI_SUBSCR_NAMESPACE_DBCHANGE`.
3. Set the subscription handle attribute `OCI_ATTR_SUBSCR_CALLBACK` to store the OCI callback associated with the query handle. The callback has the following prototype:

```
void notification_callback (void *ctx, OCISubscription *subscrhp,
                           void *payload, ub4 paylen, void *desc, ub4 mode);
```

The parameters are described in "Notification Callback in OCI" in *Oracle Call Interface Programmer's Guide*.

4. Optionally associate a client-specific context using `OCI_ATTR_SUBSCR_CTX` attribute.
5. Set the `OCI_ATTR_SUBSCR_TIMEOUT` attribute to specify a `ub4` timeout interval in seconds. If it is not set, there is no timeout.
6. Set the `OCI_ATTR_SUBSCR_QOSFLAGS` attribute, the QOS (quality of service) levels, with the following values:
 - The `OCI_SUBSCR_QOS_PURGE_ON_NTFN` flag allows the registration to be purged on the first notification.
 - The `OCI_SUBSCR_QOS_RELIABLE` flag allows notifications to be persistent. You can use surviving instances of Oracle RAC to send and retrieve continuous query notification messages, even after a node failure, because invalidations associated with this registration are queued persistently into the database. If `FALSE`, then invalidations are enqueued into a fast in-memory queue. This option describes the persistence of notifications and not the persistence of registrations. Registrations are automatically persistent by default.
7. Call `OCISubscriptionRegister()` to create a new registration in the DBCHANGE namespace.
8. Associate multiple query statements with the subscription handle by setting the attribute `OCI_ATTR_CHNF_REGHANDLE` of the statement handle, `OCI_HTYPE_STMT`. The registration is completed when the query is executed.

See Also: *Oracle Call Interface Programmer's Guide* for more information about `OCI_ATTR_CHNF_REGHANDLE`

9. Optionally unregister a subscription. The client can call the `OCISubscriptionRegister()` function with the subscription handle as a parameter.

A binding of a statement handle to a subscription handle is valid only for only the first execution of a query. If the application must use the same OCI statement handle for subsequent executions, it must repopulate the registration handle attribute of the

statement handle. A binding of a subscription handle to a statement handle is permitted only when the statement is a query (determined at execute time). If a DML statement is executed as part of the execution, then an exception is issued.

15.8.3 Using OCI Subscription Handle Attributes for Continuous Query Notification

The subscription handle attributes for continuous query notification can be divided into generic attributes (common to all subscriptions) and namespace-specific attributes (particular to continuous query notification).

The `WRITE` attributes on the statement handle can be modified only before the registration is created.

Generic Attributes - Common to All Subscriptions

`OCI_ATTR_SUBSCR_NAMESPACE (WRITE)` - Set this attribute to `OCI_SUBSCR_NAMESPACE_DBCHANGE` for subscription handles.

`OCI_ATTR_SUBSCR_CALLBACK (WRITE)` - Use this attribute to store the callback associated with the subscription handle. The callback is executed when a notification is received.

When a new continuous query notification message becomes available, the callback is invoked in the listener thread with `desc` pointing to a descriptor of type `OCI_DTYPE_CHDES` that contains detailed information about the invalidation.

`OCI_ATTR_SUBSCR_QOSFLAGS` - This attribute is a generic flag with the following values:

<code>#define OCI_SUBSCR_QOS_RELIABLE</code>	<code>0x01</code>	<code>/* reliable */</code>
<code>#define OCI_SUBSCR_QOS_PURGE_ON_NTFN</code>	<code>0x10</code>	<code>/* purge on first ntn */</code>

- `OCI_SUBSCR_QOS_RELIABLE` - Set this bit to allow notifications to be persistent. Therefore, you can use surviving instances of an Oracle RAC cluster to send and retrieve invalidation messages, even after a node failure, because invalidations associated with this registration ID are queued persistently into the database. If this bit is `FALSE`, then invalidations are enqueued in to a fast in-memory queue. This option describes the persistence of notifications and not the persistence of registrations. Registrations are automatically persistent by default.
- `OCI_SUBSCR_QOS_PURGE_ON_NTFN` - Set this bit to allow the registration to be purged on the first notification.

A parallel example is presented in *Oracle Call Interface Programmer's Guide* in publish-subscribe registration functions in OCI.

`OCI_ATTR_SUBSCR_CQ_QOSFLAGS` - This attribute describes the continuous query notification-specific QOS flags (mode is `WRITE`, data type is `ub4`), which are:

- `0x1 OCI_SUBSCR_CQ_QOS_QUERY` - Set this flag to indicate that query-level granularity is required. Generate notification only if the query result set changes. By default, this level of QOS has no false positives.
- `0x2 OCI_SUBSCR_CQ_QOS_BEST EFFORT` - Set this flag to indicate that best effort filtering is acceptable. It can be used by caching applications. The database can use heuristics based on cost of evaluation and avoid full pruning in some cases.

`OCI_ATTR_SUBSCR_TIMEOUT` - Use this attribute to specify a `ub4` timeout value defined in seconds. If the timeout value is 0 or not specified, then the registration is active until explicitly unregistered.

Namespace- Specific or Feature-Specific Attributes

The following attributes are namespace-specific or feature-specific to the continuous query notification feature.

`OCI_ATTR_CHNF_TABLENAMES` (data type is `(OCIColl *)`) - These attributes are provided to retrieve the list of table names that were registered. These attributes are available from the subscription handle, after the query is executed.

`OCI_ATTR_CHNF_ROWIDS` - A Boolean attribute (default FALSE). If TRUE, then the continuous query notification message includes row-level details such as operation type and `ROWID`.

`OCI_ATTR_CHNF_OPERATIONS` - Use this `ub4` flag to selectively filter notifications based on operation type. This option is ignored if the registration is of query-level granularity. Flags stored are:

- `OCI_OPCODE_ALL` - All operations
- `OCI_OPCODE_INSERT` - Insert operations on the table
- `OCI_OPCODE_UPDATE` - Update operations on the table
- `OCI_OPCODE_DELETE` - Delete operations on the table

`OCI_ATTR_CHNF_CHANGELAG` - The client can use this `ub4` value to specify the number of transactions by which the client is willing to lag behind. The client can also use this option as a throttling mechanism for continuous query notification messages. When you choose this option, `ROWID`-level granularity of information is unavailable in the notifications, even if `OCI_ATTR_CHNF_ROWIDS` was TRUE. This option is ignored if the registration is of query-level granularity.

After the `OCISubscriptionRegister()` call is invoked, none of the preceding attributes (generic, name-specific, or feature-specific) can be modified on the registration already created. Any attempt to modify those attributes is not reflected on the registration already created, but it does take effect on newly created registrations that use the same registration handle.

See Also: *Oracle Call Interface Programmer's Guide* for more information about continuous query notification descriptor attributes

Notifications can be spaced out by using the grouping NTFN option. The relevant generic notification attributes are:

`OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE`
`OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE`
`OCI_ATTR_SUBSCR_NTFN_GROUPING_START_TIME`
`OCI_ATTR_SUBSCR_NTFN_GROUPING_REPEAT_COUNT`

See Also: *Oracle Call Interface Programmer's Guide* for more details about these attributes in publish-subscribe register directly to the database

15.8.4 Using `OCI_ATTR_CQ_QUERYID` Attribute

The attribute `OCI_ATTR_CQ_QUERYID` on the statement handle, `OCI_HTYPE_STMT`, obtains the query ID of a registered query after registration is made by the call to `OCISqlExecute()`.

See Also: *Oracle Call Interface Programmer's Guide* for more information about `OCI_ATTR_CQ_QUERYID`

15.8.5 Using OCI Continuous Query Notification Descriptors

The continuous query notification descriptor is passed into the desc parameter of the notification callback specified by the application. The following attributes are specific to continuous query notification. The OCI type constant of the continuous query notification descriptor is OCI_DTYPE_CHDES.

The notification callback receives the top-level notification descriptor, OCI_DTYPE_CHDES, as an argument. This descriptor in turn includes either a collection of OCI_DTYPE_CQDES or OCI_DTYPE_TABLE_CHDES descriptors based on whether the event type was OCI_EVENT_QUERYCHANGE or OCI_EVENT_OBJCHANGE. An array of table continuous query descriptors is embedded inside the continuous query descriptor for notifications of type OCI_EVENT_QUERYCHANGE. If ROWID level granularity of information was requested, each OCI_DTYPE_TABLE_CHDES contains an array of row-level continuous query descriptors (OCI_DTYPE_ROW_CHDES) corresponding to each modified ROWID.

15.8.5.1 OCI_DTYPE_CHDES

This is the top-level continuous query notification descriptor type.

OCI_ATTR_CHDES_DBNAME (oratext *) - Name of the database (source of the continuous query notification)

OCI_ATTR_CHDES_XID (RAW(8)) - Message ID of the message

OCI_ATTR_CHDES_NFTYPE - Flags describing the notification type:

- 0x0 OCI_EVENT_NONE - No further information about the continuous query notification
- 0x1 OCI_EVENT_STARTUP - Instance startup
- 0x2 OCI_EVENT_SHUTDOWN - Instance shutdown
- 0x3 OCI_EVENT_SHUTDOWN_ANY - Any instance shutdown - Oracle Real Application Clusters (Oracle RAC)
- 0x5 OCI_EVENT_DEREG - Unregistered or timed out
- 0x6 OCI_EVENT_OBJCHANGE - Object change notification
- 0x7 OCI_EVENT_QUERYCHANGE - Query change notification

OCI_ATTR_CHDES_TABLE_CHANGES - A collection type describing operations on tables of data type (OCIColl *). This attribute is present only if the OCI_ATTR_CHDES_NFTYPE attribute was of type OCI_EVENT_OBJCHANGE; otherwise, it is NULL. Each element of the collection is a table of continuous query descriptors of type OCI_DTYPE_TABLE_CHDES.

OCI_ATTR_CHDES_QUERIES - A collection type describing the queries that were invalidated. Each member of the collection is of type OCI_DTYPE_CQDES. This attribute is present only if the attribute OCI_ATTR_CHDES_NFTYPE was of type OCI_EVENT_QUERYCHANGE; otherwise, it is NULL.

15.8.5.1.1 OCI_DTYPE_CQDES This notification descriptor describes a query that was invalidated, usually in response to the commit of a DML or a DDL transaction. It has the following attributes:

- OCI_ATTR_CQDES_OPERATION (ub4, READ) - Operation that occurred on the query. It can be one of these values:
 - OCI_EVENT_QUERYCHANGE - Query result set change
 - OCI_EVENT_DEREG - Query unregistered

- OCI_ATTR_CQDES_TABLE_CHANGES (OCIColl *, READ) - A collection of table continuous query descriptors describing DML or DDL operations on tables that caused the query result set change. Each element of the collection is of type OCI_DTYPE_TABLE_CHDES.
- OCI_ATTR_CQDES_QUERYID (ub8, READ) - Query ID of the query that was invalidated.

15.8.5.1.2 OCI_DTYPE_TABLE_CHDES This notification descriptor conveys information about changes to a table involved in a registered query. It has the following attributes:

- OCI_ATTR_CHDES_TABLE_NAME (oratext *) - Schema annotated table name.
- OCI_ATTR_CHDES_TABLE_OPFLAGS (ub4) - Flag field describing the operations on the table. Each of the following flag fields is in a separate bit position in the attribute:
 - 0x1 OCI_OPCODE_ALLROWS - The table is completely invalidated.
 - 0x2 OCI_OPCODE_INSERT - Insert operations on the table.
 - 0x4 OCI_OPCODE_UPDATE - Update operations on the table.
 - 0x8 OCI_OPCODE_DELETE - Delete operations on the table.
 - 0x10 OCI_OPCODE_ALTER - Table altered (schema change). This includes DDL statements and internal operations that cause row migration.
 - 0x20 OCI_OPCODE_DROP - Table dropped.
- OCI_ATTR_CHDES_TABLE_ROW_CHANGES - This is an embedded collection describing the changes to the rows within the table. Each element of the collection is a row continuous query descriptor of type OCI_DTYPE_ROW_CHDES that has the following attributes:
 - OCI_ATTR_CHDES_ROW_ROWID (OraText *) - String representation of a ROWID.
 - OCI_ATTR_CHDES_ROW_OPFLAGS - Reflects the operation type: INSERT, UPDATE, DELETE, or OTHER.

See Also: *Oracle Call Interface Programmer's Guide* for more information about continuous query notification descriptor attributes

15.8.6 Demonstrating Continuous Query Notification in an OCI Sample Program

[Example 15–8](#) is a simple OCI program, `demoquery.c`. See the comments in the listing. The calling session must have the CHANGE NOTIFICATION system privilege and SELECT privileges on all objects that it attempts to register.

Example 15–8 Program Listing That Demonstrates Continuous Query Notification

```
/* Copyright (c) 2010, Oracle. All rights reserved. */

#ifndef S_ORACLE
#include <oratypes.h>
#endif

*****  

*This is a DEMO program. To test, compile the file to generate the executable  

*demoquery. Then demoquery can be invoked from a command prompt.  

*It will have the following output:  

  Initializing OCI Process  

  Registering query : select last_name, employees.department_id, department_name  

           from employees, departments
```

```
where employee_id = 200
      and employees.department_id = departments.department_id

Query Id 23
Waiting for Notifications

*Then from another session, log in as HR/HR and perform the following
* DML transactions. It will cause two notifications to be generated.

update departments set department_name ='Global Admin' where department_id=10;
commit;
update departments set department_name ='Administration' where department_id=10;
commit;

*The demoquery program will now show the following output corresponding
*to the notifications received.
```

```
Query 23 is changed
Table changed is HR.DEPARTMENTS table_op 4
Row changed is AAAMBoAABAAKX2AAA row_op 4
Query 23 is changed
Table changed is HR.DEPARTMENTS table_op 4
Row changed is AAAMBoAABAAKX2AAA row_op 4
```

*The demo program waits for exactly 10 notifications to be received before logging off and unregistering the subscription.

```
***** /  
/*-----  
 PRIVATE TYPES AND CONSTANTS  
-----*/  
  
/*-----  
 STATIC FUNCTION DECLARATIONS  
-----*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <oci.h>  
  
#define MAXSTRLENGTH 1024  
#define bit(a,b) ((a)&(b))  
  
static int notifications_processed = 0;  
static OCISubscription *subhandle1 = (OCISubscription *)0;  
static OCISubscription *subhandle2 = (OCISubscription *)0;  
static void checker(/*_ OCIError *errhp, sword status _*/);  
static void registerQuery(/*_ OCISvcCtx *svchp, OCIError *errhp, OCISql *stmt  
                           OCIEnv *envhp _*/);  
static void myCallback (/*_ dvoid *ctx, OCISubscription *subscrhp,  
                           dvoid *payload, ub4 *payl, dvoid *descriptor,  
                           ub4 mode _*/);  
static int NotificationDriver(/*_ int argc, char *argv[] _*/);  
static sword status;  
static boolean logged_on = FALSE;  
static void processRowChanges(OCIEnv *envhp, OCIError *errhp, OCISql *stmthp,  
                           OCIColl *row changes);
```

```

static void processTableChanges(OCIEnv *envhp, OCIError *errhp,
                               OCISStmt *stmthp, OCIColl *table_changes);
static void processQueryChanges(OCIEnv *envhp, OCIError *errhp, OCISStmt *stmthp,
                               OCIColl *query_changes);
static int nonractests2/*_ int argc, char *argv[] */;

int main(int argc, char **argv)
{
    NotificationDriver(argc, argv);
    return 0;
}

int NotificationDriver(argc, argv)
int argc;
char *argv[];
{
    OCIEnv *envhp;
    OCISvcCtx *svchp, *svchp2;
    OCIError *errhp, *errhp2;
    OCISession *authp, *authp2;
    OCISStmt *stmthp, *stmthp2;
    OCIDuration dur, dur2;
    int i;
    dvoid *tmp;
    OCISession *usrhp;
    OCIServer *srvhp;

    printf("Initializing OCI Process\n");
/* Initialize the environment. The environment must be initialized
   with OCI_EVENTS and OCI_OBJECT to create a continuous query notification
   registration and receive notifications.
*/
    OCIEnvCreate( (OCIEnv **) &envhp, OCI_EVENTS|OCI_OBJECT, (dvoid *)0,
                  (dvoid * (*) (dvoid *, size_t)) 0,
                  (dvoid * (*) (dvoid *, dvoid *, size_t))0,
                  (void (*) (dvoid *, dvoid *)) 0,
                  (size_t) 0, (dvoid **) 0 );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                    (size_t) 0, (dvoid **) 0 );
    /* server contexts */
    OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                  (size_t) 0, (dvoid **) 0 );
    OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                  (size_t) 0, (dvoid **) 0 );
    checker(errhp, OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0,
                                   (ub4) OCI_DEFAULT));
    /* set attribute server context in the service context */
    OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) srvhp,
                (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp );

    /* allocate a user context handle */
    OCIHandleAlloc((dvoid *)envhp, (dvoid **) &usrhp, (ub4) OCI_HTYPE_SESSION,
                  (size_t) 0, (dvoid **) 0 );

    OCIAttrSet((dvoid *)usrhp, (ub4) OCI_HTYPE_SESSION,
               (dvoid *)((text *)"HR"), (ub4)strlen((char *)"HR"),

```

```

    OCI_ATTR_USERNAME, errhp);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
            (dvoid *)((text *)"HR"), (ub4)strlen((char *)"HR"),
            OCI_ATTR_PASSWORD, errhp);
checker(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
                                OCI_DEFAULT));
/* Allocate a statement handle */
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                (ub4) OCI_HTYPE_STMT, 52, (dvoid **) &tmp);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp, (ub4)0,
            OCI_ATTR_SESSION, errhp);

registerQuery(svchp, errhp, stmthp, envhp);
printf("Waiting for Notifications\n");
while (notifications_processed !=10)
{
    sleep(1);
}
printf ("Going to unregister HR\n");
fflush(stdout);
/* Unregister HR */
checker(errhp,
        OCISubscriptionUnRegister(svchp, subhandle1, errhp, OCI_DEFAULT));
checker(errhp, OCISessionEnd(svchp, errhp, usrhp, (ub4) 0));
printf("HR Logged off.\n");

if (subhandle1)
    OCIHandleFree((dvoid *)subhandle1, OCI_HTYPE_SUBSCRIPTION);
if (stmthp)
    OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT);
if (srvhp)
    OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER);
if (svchp)
    OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX);
if (authp)
    OCIHandleFree((dvoid *) usrhp, (ub4) OCI_HTYPE_SESSION);
if (errhp)
    OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR);
if (envhp)
    OCIHandleFree((dvoid *) envhp, (ub4) OCI_HTYPE_ENV);

return 0;
}

void checker(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    sb4 errcode = 0;
    int retval = 1;

    switch (status)
    {
    case OCI_SUCCESS:
        retval = 0;

```

```

        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;
    }
    if (retval)
    {
        exit(1);
    }
}

void processRowChanges(OCIEnv *envhp, OCIError *errhp, OCISstmt *stmthp,
                      OCIColl *row_changes)
{
    dvoid **row_descp;
    dvoid *row_desc;
    boolean exist;
    ub2 i, j;
    dvoid *elemind = (dvoid *)0;
    oratext *row_id;
    ub4 row_op;

    sb4 num_rows;
    if (!row_changes) return;
    checker(errhp, OCICollSize(envhp, errhp,
                               (CONST OCIColl *) row_changes, &num_rows));
    for (i=0; i<num_rows; i++)
    {
        checker(errhp, OCICollGetElem(envhp,
                                      errhp, (OCIColl *) row_changes,
                                      i, &exist, &row_descp, &elemind));

        row_desc = *row_descp;
        checker(errhp, OCIAttrGet (row_desc,
                                   OCI_DTYPE_ROW_CHDES, (dvoid *)&row_id,
                                   NULL, OCI_ATTR_CHDES_ROW_ROWID, errhp));
}

```

```

        checker(errhp, OCIAttrGet (row_desc,
                                   OCI_DTYPE_ROW_CHDES, (dvoid *)&row_op,
                                   NULL, OCI_ATTR_CHDES_ROW_OPFLAGS, errhp));

        printf ("Row changed is %s row_op %d\n", row_id, row_op);
        fflush(stdout);
    }
}

void processTableChanges(OCIEnv *envhp, OCIError *errhp, OCISstmt *stmthp,
                        OCIColl *table_changes)
{
    dvoid **table_descp;
    dvoid *table_desc;
    dvoid **row_descp;
    dvoid *row_desc;
    OCIColl *row_changes = (OCIColl *)0;
    boolean exist;
    ub2 i, j;
    dvoid *elemind = (dvoid *)0;
    oratext *table_name;
    ub4 table_op;

    sb4 num_tables;
    if (!table_changes) return;
    checker(errhp, OCICollSize(envhp, errhp,
                               (CONST OCIColl *) table_changes, &num_tables));
    for (i=0; i<num_tables; i++)
    {
        checker(errhp, OCICollGetElem(envhp,
                                      errhp, (OCIColl *) table_changes,
                                      i, &exist, &table_descp, &elemind));

        table_desc = *table_descp;
        checker(errhp, OCIAttrGet (table_desc,
                                   OCI_DTYPE_TABLE_CHDES, (dvoid *)&table_name,
                                   NULL, OCI_ATTR_CHDES_TABLE_NAME, errhp));
        checker(errhp, OCIAttrGet (table_desc,
                                   OCI_DTYPE_TABLE_CHDES, (dvoid *)&table_op,
                                   NULL, OCI_ATTR_CHDES_TABLE_OPFLAGS, errhp));
        checker(errhp, OCIAttrGet (table_desc,
                                   OCI_DTYPE_TABLE_CHDES, (dvoid *)&row_changes,
                                   NULL, OCI_ATTR_CHDES_TABLE_ROW_CHANGES, errhp));

        printf ("Table changed is %s table_op %d\n", table_name, table_op);
        fflush(stdout);
        if (!bit(table_op, OCI_OPCODE_ALLROWS))
            processRowChanges(envhp, errhp, stmthp, row_changes);
    }
}

void processQueryChanges(OCIEnv *envhp, OCIError *errhp, OCISstmt *stmthp,
                        OCIColl *query_changes)
{
    sb4 num_queries;
    ub8 queryid;
    OCINumber qidnum;
    ub4 queryop;
    dvoid *elemind = (dvoid *)0;
}

```

```

dvoid *query_desc;
dvoid **query_descp;
ub2 i;
boolean exist;
OCIColl *table_changes = (OCIColl *)0;

if (!query_changes) return;
checker(errhp, OCICollSize(envhp, errhp,
                           (CONST OCIColl *) query_changes, &num_queries));
for (i=0; i < num_queries; i++)
{
    checker(errhp, OCICollGetElem(envhp,
                                  errhp, (OCIColl *) query_changes,
                                  i, &exist, &query_descp, &elemind));

    query_desc = *query_descp;
    checker(errhp, OCIAttrGet (query_desc,
                               OCI_DTYPE_CQDES, (dvoid *)&queryid,
                               NULL, OCI_ATTR_CQDES_QUERYID, errhp));
    checker(errhp, OCIAttrGet (query_desc,
                               OCI_DTYPE_CQDES, (dvoid *)&queryop,
                               NULL, OCI_ATTR_CQDES_OPERATION, errhp));
    printf(" Query %d is changed\n", queryid);
    if (queryop == OCI_EVENT_DEREG)
        printf("Query Deregistered\n");
    checker(errhp, OCIAttrGet (query_desc,
                               OCI_DTYPE_CQDES, (dvoid *)&table_changes,
                               NULL, OCI_ATTR_CQDES_TABLE_CHANGES, errhp));
    processTableChanges(envhp, errhp, stmthp, table_changes);

}

}

void myCallback (ctx, subscrhp, payload, payl, descriptor, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *payload;
ub4 *payl;
dvoid *descriptor;
ub4 mode;
{
    OCIColl *table_changes = (OCIColl *)0;
    OCIColl *row_changes = (OCIColl *)0;
    dvoid *change_descriptor = descriptor;
    ub4 notify_type;
    ub2 i, j;
    OCIEnv *envhp;
    OCIError *errhp;
    OCIColl *query_changes = (OCIColl *)0;
    OCIServer *srvhp;
    OCISvcCtx *svchp;
    OCISession *usrhp;
    dvoid    *tmp;
    OCISStmt *stmthp;

    (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 **) &svchp, OCI_HTYPE_SERVER,
        (size_t) 0, (dvoid **) 0);

(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
        (size_t) 0, (dvoid **) 0);

OCIAttrGet (change_descriptor, OCI_DTYPE_CHDES, (dvoid *) &notify_type,
        NULL, OCI_ATTR_CHDES_NFTYPE, errhp);
fflush(stdout);
if (notify_type == OCI_EVENT_SHUTDOWN ||
    notify_type == OCI_EVENT_SHUTDOWN_ANY)
{
    printf("SHUTDOWN NOTIFICATION RECEIVED\n");
    fflush(stdout);
    notifications_processed++;
    return;
}
if (notify_type == OCI_EVENT_STARTUP)
{
    printf("STARTUP NOTIFICATION RECEIVED\n");
    fflush(stdout);
    notifications_processed++;
    return;
}

notifications_processed++;
checker(errhp, OCIServerAttach( svchp, errhp, (text *) 0, (sb4) 0,
        (ub4) OCI_DEFAULT));

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
        52, (dvoid **) &tmp);
/* set attribute server context in the service context */
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) svchp,
        (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
        (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
        (dvoid *)"HR", (ub4)strlen("HR"), OCI_ATTR_USERNAME, errhp);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
        (dvoid *)"HR", (ub4)strlen("HR"),
        OCI_ATTR_PASSWORD, errhp);

checker(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
        OCI_DEFAULT));

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
        (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* Allocate a statement handle */
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
        (ub4) OCI_HTYPE_STMT, 52, (dvoid **) &tmp);

if (notify_type == OCI_EVENT_OBJCHANGE)
{

```

```

        checker(errhp, OCIAttrGet (change_descriptor,
                                   OCI_DTYPE_CHDES, &table_changes, NULL,
                                   OCI_ATTR_CHDES_TABLE_CHANGES, errhp));
        processTableChanges(envhp, errhp, stmthp, table_changes);
    }
    else if (notify_type == OCI_EVENT_QUERYCHANGE)
    {
        checker(errhp, OCIAttrGet (change_descriptor,
                                   OCI_DTYPE_CHDES, &query_changes, NULL,
                                   OCI_ATTR_CHDES_QUERIES, errhp));
        processQueryChanges(envhp, errhp, stmthp, query_changes);
    }
    checker(errhp, OCISessionEnd(svchp, errhp, ushrp, OCI_DEFAULT));
    checker(errhp, OCIServerDetach(srvhp, errhp, OCI_DEFAULT));
    if (stmthp)
        OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT);
    if (errhp)
        OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR);
    if (srvhp)
        OCIHandleFree((dvoid *)srvhp, OCI_HTYPE_SERVER);
    if (svchp)
        OCIHandleFree((dvoid *)svchp, OCI_HTYPE_SVCCTX);
    if (usrhp)
        OCIHandleFree((dvoid *)usrhp, OCI_HTYPE_SESSION);
    if (envhp)
        OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
    }

void registerQuery(svchp, errhp, stmthp, envhp)
OCISvcCtx *svchp;
OCIError *errhp;
OCISStmt *stmthp;
OCIEnv *envhp;
{
    OCISubscription *subscrhp;
    ub4 namespace = OCI_SUBSCR_NAMESPACE_DBCHANGE;
    ub4 timeout = 60;
    OCIDefine *defnp1 = (OCIDefine *)0;
    OCIDefine *defnp2 = (OCIDefine *)0;
    OCIDefine *defnp3 = (OCIDefine *)0;
    OCIDefine *defnp4 = (OCIDefine *)0;
    OCIDefine *defnp5 = (OCIDefine *)0;
    int mgr_id =0;
    text query_text1[] = "select last_name, employees.department_id, department_name \
from employees,departments where employee_id = 200 and employees.department_id =\
departments.department_id";

    ub4 num_prefetch_rows = 0;
    ub4 num_reg_tables;
    OCIColl *table_names;
    ub2 i;
    boolean rowids = TRUE;
    ub4 qosflags = OCI_SUBSCR_CQ_QOS_QUERY  ;
    int empno=0;
    OCINumber qidnum;
    ub8 qid;
    char outstr[MAXSTRLENGTH], dname[MAXSTRLENGTH];
    int q3out;
}

```

```

fflush(stdout);
/* allocate subscription handle */
OCIHandleAlloc ((dvoid *) envhp, (dvoid **) &subscrhp,
                 OCI_HTYPE_SUBSCRIPTION, (size_t) 0, (dvoid **) 0);

/* set the namespace to DBCHANGE */
checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
                           (dvoid *) &namespace, sizeof(ub4),
                           OCI_ATTR_SUBSCR_NAMESPACE, errhp));

/* Associate a notification callback with the subscription */
checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
                           (void *)myCallback, 0, OCI_ATTR_SUBSCR_CALLBACK, errhp));
/* Allow extraction of rowid information */
checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
                           (dvoid *)&rowids, sizeof(ub4),
                           OCI_ATTR_CHNF_ROWIDS, errhp));

checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
                           (dvoid *)&qosflags, sizeof(ub4),
                           OCI_ATTR_SUBSCR_CQ_QOSFLAGS, errhp));

/* Create a new registration in the DBCHANGE namespace */
checker(errhp,
        OCISubscriptionRegister(svchp, &subscrhp, 1, errhp, OCI_DEFAULT));

/* Multiple queries can now be associated with the subscription */

subhandle1 = subscrhp;

printf("Registering query : %s\n", (const signed char *)query_text1);
/* Prepare the statement */
checker(errhp, OCISqlPrepare (stmthp, errhp, query_text1,
                             (ub4)strlen((const signed char *)query_text1), OCI_V7_SYNTAX,
                             OCI_DEFAULT));

checker(errhp,
        OCIDefineByPos(stmthp, &defnp1,
                      errhp, 1, (dvoid *)outstr, MAXSTRLENGTH * sizeof(char),
                      SQLT_STR, (dvoid *)0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT));
checker(errhp,
        OCIDefineByPos(stmthp, &defnp2,
                      errhp, 2, (dvoid *)&empno, sizeof(empno),
                      SQLT_INT, (dvoid *)0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT));
checker(errhp,
        OCIDefineByPos(stmthp, &defnp3,
                      errhp, 3, (dvoid *)&cname, sizeof(cname),
                      SQLT_STR, (dvoid *)0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT));

/* Associate the statement with the subscription handle */
OCIAttrSet (stmthp, OCI_HTYPE_STMT, subscrhp, 0,
            OCI_ATTR_CHNF_REGHANDLE, errhp);

/* Execute the statement, the execution performs object registration */
checker(errhp, OCISqlExecute (svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL ,
                             OCI_DEFAULT));
fflush(stdout);

```

```

        OCIAttrGet(stmthp, OCI_HTYPE_STMT, &qid, (ub4 *)0,
                    OCI_ATTR_CQ_QUERYID, errhp);
        printf("Query Id %d\n", qid);

        /* commit */
        checker(errhp, OCITransCommit(svchp, errhp, (ub4) 0));

    }

    static void cleanup(envhp, svchp, srvhp, errhp, usrhp)
    OCIEnv *envhp;
    OCISvcCtx *svchp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISession *usrhp;
    {
        /* detach from the server */
        checker(errhp, OCISessionEnd(svchp, errhp, usrhp, OCI_DEFAULT));
        checker(errhp, OCIServerDetach(srvhp, errhp, (ub4)OCI_DEFAULT));

        if (usrhp)
            (void) OCIHandleFree((dvoid *) usrhp, (ub4) OCI_HTYPE_SESSION);
        if (svchp)
            (void) OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX);
        if (srvhp)
            (void) OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER);
        if (errhp)
            (void) OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR);
        if (envhp)
            (void) OCIHandleFree((dvoid *) envhp, (ub4) OCI_HTYPE_ENV);
    }
}

```

15.9 Querying CQN Registrations

To see top-level information about all registrations, including their QOS options, query the static data dictionary view `*_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, use this query:

```
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`

15.10 Interpreting Notifications

When a transaction commits, the database determines whether registered objects were modified in the transaction. If so, it runs 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](#)

15.10.1 Interpreting a CQ_NOTIFICATION\$_DESCRIPTOR Object

When a CQN registration generates a notification, the 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 running this statement:

```
DESC CQ_NOTIFICATION$_DESCRIPTOR
```

Table 15–4 summarizes the attributes of CQ_NOTIFICATION\$_DESCRIPTOR.

Table 15–4 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	<p>The database event that triggers a notification. For example, the attribute can contain these constants, which correspond to different database events:</p> <ul style="list-style-type: none"> ▪ EVENT_NONE ▪ EVENT_STARTUP (Instance startup) ▪ EVENT_SHUTDOWN (Instance shutdown - last instance shutdown for Oracle RAC) ▪ EVENT_SHUTDOWN_ANY (Any instance shutdown for 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	<p>This field is present only for OCN registrations. For QRCN registrations, it is NULL.</p> <p>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 15–5.</p> <p>Otherwise: NULL.</p>

Table 15–4 (Cont.) Attributes of CQ_NOTIFICATION\$_DESCRIPTOR

Attribute	Description
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 15–6 . Otherwise: NULL.

15.10.2 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 running this statement:

```
DESC CQ_NOTIFICATION$_TABLE
```

[Table 15–5](#) summarizes the attributes of CQ_NOTIFICATION\$_TABLE.

Table 15–5 Attributes of CQ_NOTIFICATION\$_TABLE

Attribute	Specifies . . .
OPFLAGS	The type of operation performed on the modified table. For example, the attribute can contain these constants, which correspond to different database operations: <ul style="list-style-type: none"> ■ ALL_ROWS signifies that either the entire table is modified, as in a DELETE *, or row-level granularity of information is not requested or unavailable 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 Table 15–7 describes. If ALL_ROWS was set in the opflags, then the ROW_DESC_ARRAY member is NULL.

15.10.3 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 running this statement:

```
DESC CQ_NOTIFICATION$_QUERY
```

[Table 15–6](#) summarizes the attributes of CQ_NOTIFICATION\$_QUERY.

Table 15–6 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 15–5 .

15.10.4 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 ROWIDs for the changed rows. If ALL_ROWS was set in the OPFLAGS field of the CQ_NOTIFICATION\$_TABLE object, then ROWID information is unavailable.

[Table 15–7](#) summarizes the attributes of CQ_NOTIFICATION\$_ROW.

Table 15–7 Attributes of CQ_NOTIFICATION\$_ROW

Attribute	Specifies . . .
OPFLAGS	The type of operation performed on the modified table. See the description of OPFLAGS in Table 15–5 .
ROW_ID	The ROWID of the changed row.

Part IV

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:

- [Chapter 16, "Using Oracle Flashback Technology"](#)
- [Chapter 17, "Choosing a Programming Environment"](#)
- [Chapter 18, "Developing Applications with Multiple Programming Languages"](#)
- [Chapter 19, "Developing Applications with Oracle XA"](#)
- [Chapter 20, "Developing Applications with the Publish-Subscribe Model"](#)
- [Chapter 21, "Using the Oracle ODBC Driver"](#)
- [Chapter 22, "Using the Identity Code Package"](#)
- [Chapter 23, "Understanding Schema Object Dependency"](#)
- [Chapter 24, "Using Edition-Based Redefinition"](#)
- [Chapter 25, "Using Transaction Guard"](#)
- [Chapter 26, "Ensuring Application Continuity"](#)

See Also: *Oracle Database Performance Tuning Guide* and *Oracle Database SQL Tuning Guide* for performance issues to consider when developing applications

Using Oracle Flashback Technology

This chapter explains how to use Oracle Flashback Technology in database applications.

Topics:

- [Overview of Oracle Flashback Technology](#)
- [Configuring Your Database for Oracle Flashback Technology](#)
- [Using Oracle Flashback Query \(SELECT AS OF\)](#)
- [Using Oracle Flashback Version Query](#)
- [Using Oracle Flashback Transaction Query](#)
- [Using Oracle Flashback Transaction Query with Oracle Flashback Version Query](#)
- [Using DBMS_FLASHBACK Package](#)
- [Using Flashback Transaction](#)
- [Using Flashback Data Archive](#)
- [General Guidelines for Oracle Flashback Technology](#)
- [Performance Guidelines for Oracle Flashback Technology](#)
- [Multitenant Container Database Restrictions for Oracle Flashback Technology](#)

16.1 Overview of Oracle Flashback Technology

Oracle Flashback Technology is a group of Oracle Database features 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:

- 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

Oracle 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 runs 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. It is retained for the time specified by `undo_retention`, or up to the tuned undo retention in the presence of Automatic Undo Management (AUM). By using flashback features, you can use undo data to query past data or recover from logical damage. Besides using it in flashback features, Oracle Database uses undo data to perform these actions:

- Roll back active transactions
- Recover terminated transactions by using database or process recovery
- Provide read consistency for SQL queries

Note: After executing a `CREATE TABLE` statement, wait at least 15 seconds to commit any transactions, to ensure that Oracle Flashback features (especially Oracle Flashback Version Query) reflect those transactions.

Topics:

- [Application Development Features](#)
- [Database Administration Features](#)

For additional general information about flashback features, see *Oracle Database Concepts*

16.1.1 Application Development Features

In application development, you can use these flashback features to report historical data or undo erroneous changes. (You can also use these features interactively as a database user or administrator.)

Oracle Flashback Query

Use this feature to retrieve data for an earlier time that you specify with the `AS OF` clause of the `SELECT` statement. For more information, see [Section 16.3, "Using Oracle Flashback Query \(SELECT AS OF\)."](#)

Oracle 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 an Oracle Flashback Version Query, use the `VERSIONS BETWEEN` clause of the `SELECT` statement. For more information, see [Section 16.4, "Using Oracle Flashback Version Query."](#)

Oracle 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. To perform an Oracle Flashback Transaction Query, select from the static data dictionary view `FLASHBACK_TRANSACTION_QUERY`. For more information, see [Section 16.5, "Using Oracle Flashback Transaction Query."](#)

Typically, you use Oracle Flashback Transaction Query with an Oracle Flashback Version Query that provides the transaction IDs for the rows of interest (see

[Section 16.6, "Using Oracle Flashback Transaction Query with Oracle Flashback Version Query."](#)

DBMS_FLASHBACK Package

Use this feature to set the internal Oracle Database clock to an earlier time 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](#)). For more information, see [Section 16.7, "Using DBMS_FLASHBACK Package."](#)

Flashback Transaction

Use Flashback Transaction to roll back a transaction and its dependent transactions while the database remains online. This recovery operation uses undo data to create and run the corresponding compensating transactions that return the affected data to its original state. (Flashback Transaction is part of DBMS_FLASHBACK package.) For more information, see [Section 16.7, "Using DBMS_FLASHBACK Package."](#)

Flashback Data Archive

Use Flashback Data Archive to automatically track and archive historical versions of changes to tables enabled for flashback archive, ensuring SQL-level access to the versions of database objects without getting a snapshot-too-old error. For more information, see [Section 16.9, "Using Flashback Data Archive."](#)

16.1.2 Database Administration Features

These flashback features are primarily for data recovery. Typically, you use these features only as a database administrator.

This chapter focuses on the [Section 16.1.1, "Application Development Features"](#). For more information about the database administration features, see *Oracle Database Administrator's Guide* and the *Oracle Database Backup and Recovery User's Guide*.

Oracle 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.

Oracle Flashback Drop

Use this feature to recover a dropped table. This feature reverses the effects of a `DROP TABLE` statement.

Oracle Flashback Database

Use this feature to quickly return the database to an earlier point in time, by using the recovery area. This is fast, because you do not have to restore database backups.

16.2 Configuring Your Database for Oracle Flashback Technology

Before you can use flashback features in your application, you or your database administrator must perform the configuration tasks described in these topics:

- [Configuring Your Database for Automatic Undo Management](#)
- [Configuring Your Database for Oracle Flashback Transaction Query](#)
- [Configuring Your Database for Flashback Transaction](#)
- [Enabling Oracle Flashback Operations on Specific LOB Columns](#)

- [Granting Necessary Privileges](#)

16.2.1 Configuring Your Database for Automatic Undo Management

To configure your database for Automatic Undo Management (AUM), you or your database administrator must:

- 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 these 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 and 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. For more information about V\$UNDOSTAT, see *Oracle Database Reference*.

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

16.2.2 Configuring Your Database for Oracle Flashback Transaction Query

To configure your database for the Oracle Flashback Transaction Query feature, you or your database administrator must:

- Ensure that Oracle Database is running with version 10.0 compatibility.
- Enable supplemental logging:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

16.2.3 Configuring Your Database for Flashback Transaction

To configure your database for the Flashback Transaction feature, you or your database administrator must:

- 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, enable minimal and primary key supplemental logging:
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
- If you want to track foreign key dependencies, enable foreign key supplemental logging:
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (FOREIGN KEY) COLUMNS;

Note: If you have very many foreign key constraints, enabling foreign key supplemental logging might not be worth the performance penalty.

16.2.4 Enabling Oracle 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

16.2.5 Granting Necessary Privileges

You or your database administrator must grant privileges to users, roles, or applications that must use these flashback features. For information about the `GRANT` statement, see *Oracle Database SQL Language Reference*.

For Oracle Flashback Query and Oracle Flashback Version Query

To allow access to specific objects during queries, grant `FLASHBACK` and either `READ` or `SELECT` privileges on those objects.

To allow queries on all tables, grant the `FLASHBACK ANY TABLE` privilege.

For Oracle Flashback Transaction Query

Grant the `SELECT ANY TRANSACTION` privilege.

To allow execution of undo SQL code retrieved by an Oracle 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 Archive

To allow a specific user to enable Flashback Data Archive on tables, using a specific Flashback Data Archive, grant the `FLASHBACK ARCHIVE` object privilege on that Flashback Data Archive to that user. To grant the `FLASHBACK ARCHIVE` object privilege,

you must either be logged on as SYSDBA or have FLASHBACK ARCHIVE ADMINISTER system privilege.

To allow execution of these statements, grant the FLASHBACK ARCHIVE ADMINISTER system privilege:

- CREATE FLASHBACK ARCHIVE
- ALTER FLASHBACK ARCHIVE
- DROP FLASHBACK ARCHIVE

To grant the FLASHBACK ARCHIVE ADMINISTER system privilege, you must be logged on as SYSDBA.

To create a default Flashback Data Archive, using either the CREATE FLASHBACK ARCHIVE or ALTER FLASHBACK ARCHIVE statement, you must be logged on as SYSDBA.

To disable Flashback Data Archive for a table that has been enabled for Flashback Data Archive, you must either be logged on as SYSDBA or have the FLASHBACK ARCHIVE ADMINISTER system privilege.

16.3 Using Oracle Flashback Query (SELECT AS OF)

To use Oracle Flashback Query, use a SELECT statement with an AS OF clause. Oracle Flashback Query retrieves data as it existed at an earlier time. The query explicitly references a past time through a time stamp or System Change Number (SCN). It returns committed data that was current at that point in time.

Uses of Oracle 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 an earlier time.
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.
- Selecting data that was valid at a particular time or at any time within a user-defined valid time period.
For example, you can find employees with valid employee information as of a particular timestamp or between a specified start and end time in the specified valid time period. (For more information, see [Section 1.9.4, "Temporal Validity Support"](#).)
- Simplifying application design by removing the need to store some kinds of temporal data.
Oracle 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 Oracle Flashback Query](#)

For more information about the `SELECT AS OF` statement, see *Oracle Database SQL Language Reference*.

16.3.1 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 Oracle 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 16-1](#) retrieves the state of the record for Chung at 9:30AM, April 4, 2004:

Example 16-1 Retrieving a Lost Row with Oracle 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 16-2](#) restores Chung's information to the `employees` table:

Example 16-2 Restoring a Lost Row After Oracle 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'
);
```

16.3.2 Guidelines for Oracle Flashback Query

- You can specify or omit the `AS OF` clause for each table and specify different times for different tables.

Note: If a table is a Flashback Data Archive and you specify a time for it that is earlier than its creation time, the query returns zero rows for that table, rather than causing an error. (For information about Flashback Data Archives, see [Section 16.9, "Using Flashback Data Archive"](#).)

- You can use the `AS OF` clause in queries to perform data definition language (DDL) operations (such as creating and truncating tables) or data manipulation language (DML) statements (such as `INSERT` and `DELETE`) in the same session as Oracle Flashback Query.
- To use the result of Oracle 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 Oracle Flashback Query in your application, use an SCN instead of a time stamp. See [Section 16.10, "General Guidelines for Oracle Flashback Technology"](#).
- 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 Oracle Flashback Query with a CREATE TABLE AS SELECT or INSERT INTO TABLE SELECT statement. For example, this 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)
  MINUS SELECT * FROM employees;
```

SYSTIMESTAMP refers to the time zone of the database host environment.

- You can use the AS OF clause in queries to check for data that was valid at a particular time. See [Section 1.9.4, "Temporal Validity Support"](#).

16.4 Using Oracle Flashback Version Query

Use Oracle Flashback Version Query to retrieve the different versions of specific rows that existed during a given time interval. A row version is created whenever a COMMIT statement is executed.

Note: After executing a CREATE TABLE statement, wait at least 15 seconds to commit any transactions, to ensure that Oracle Flashback Version Query reflects those transactions.

Specify Oracle Flashback Version Query using the VERSIONS BETWEEN clause of the SELECT statement. The syntax is either:

```
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*).

or:

```
VERSIONS PERIOD FOR user_valid_time [ BETWEEN TIMESTAMP start AND end ]
```

where *user_valid_time* refers to the user-specified valid time period, as explained in [Section 1.9.4, "Temporal Validity Support"](#).

Oracle 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, which can reveal when and how a particular change (perhaps erroneous) occurred to your database.

Table 16–1 describes the pseudocolumns of metadata about the row version. The `VERSIONS_*` pseudocolumns have values only for transaction-time Flashback Version Queries (that is, queries with the clause `BETWEEN TIMESTAMP start AND end`).

Table 16–1 Oracle Flashback Version Query Row Data Pseudocolumns

Pseudocolumn Name	Description
<code>VERSIONS_STARTSCN</code>	Starting System Change Number (SCN) or <code>TIMESTAMP</code> 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 Oracle Flashback Table or Oracle Flashback Query.
<code>VERSIONS_STARTTIME</code>	If this pseudocolumn is <code>NULL</code> , then the row version was created before <code>start</code> .
<code>VERSIONS_ENDSCN</code>	SCN or <code>TIMESTAMP</code> when the row version expired.
<code>VERSIONS_ENDTIME</code>	If this pseudocolumn is <code>NULL</code> , then either the row version was current at the time of the query or the row corresponds to a <code>DELETE</code> operation.
<code>VERSIONS_XID</code>	Identifier of the transaction that created the row version.
<code>VERSIONS_OPERATION</code>	Operation performed by the transaction: <code>I</code> for insertion, <code>D</code> for deletion, or <code>U</code> for update. The version is that of the row that was inserted, deleted, or updated; that is, the row after an <code>INSERT</code> operation, the row before a <code>DELETE</code> operation, or the row affected by an <code>UPDATE</code> operation. For user updates of an index key, Oracle Flashback Version Query might treat an <code>UPDATE</code> operation as two operations, <code>DELETE</code> plus <code>INSERT</code> , represented as two version rows with a <code>D</code> followed by an <code>I</code> <code>VERSIONS_OPERATION</code> .

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* \leq t < VERSIONS_END*`. For example, this output indicates that the salary was 10243 from September 9, 2002, included, to November 25, 2003, excluded.

<code>VERSIONS_START_TIME</code>	<code>VERSIONS_END_TIME</code>	<code>SALARY</code>
-----	-----	-----
09-SEP-2003	25-NOV-2003	10243

Here is a typical use of Oracle Flashback Version Query:

```
SELECT versions_startscn, versions_starttime,
       versions_endscn, versions_endtime,
       versions_xid, versions_operation,
       last_name, salary
  FROM employees
 VERSIONS BETWEEN TIMESTAMP
    TO_TIMESTAMP('2008-12-18 14:00:00', 'YYYY-MM-DD HH24:MI:SS')
  AND TO_TIMESTAMP('2008-12-18 17:00:00', 'YYYY-MM-DD HH24:MI:SS')
 WHERE first_name = 'John';
```

You can use `VERSIONS_XID` with Oracle 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 [Section 16.5, "Using Oracle Flashback Transaction Query"](#).

Flashback Version Query allows index-only access only with IOTs (index-organized tables), but index fast full scan is not allowed.

See Also: *Oracle Database SQL Language Reference* for information about Oracle Flashback Version Query pseudocolumns and the syntax of the `VERSIONS` clause

16.5 Using Oracle Flashback Transaction Query

Use Oracle Flashback Transaction Query to retrieve metadata and historical data for a given transaction or for all transactions in a given time interval. Oracle Flashback Transaction Query queries the static data dictionary view `FLASHBACK_TRANSACTION_QUERY`, whose columns are described in *Oracle Database Reference*.

The column `UNDO_SQL` shows the SQL code that is the logical opposite of the DML operation performed by the transaction. You can usually use this code to reverse the logical steps taken during the transaction. However, there are cases where the `SQL_UNDO` code is not the exact opposite of the original transaction. For example, a `SQL_UNDO` `INSERT` operation might not insert a row back in a table at the same `ROWID` from which it was deleted.

This 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 that shows the logical opposite of the operation:

```
SELECT xid, operation, start_scn, commit_scn, logon_user, undo_sql
  FROM flashback_transaction_query
 WHERE xid = HEXTORAW('000200030000002D');
```

This statement uses Oracle 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')
);
```

Note: If you query `FLASHBACK_TRANSACTION_QUERY` without specifying `XID` in the `WHERE` clause, the query scans many unrelated rows, degrading performance.

See Also:

- *Oracle Database Backup and Recovery User's Guide*. for information about how a database administrator can use Flashback Table to restore an entire table, rather than individual rows
- *Oracle Database Administrator's Guide* for information about how a database administrator can use Flashback Table to restore an entire table, rather than individual rows

16.6 Using Oracle Flashback Transaction Query with Oracle Flashback Version Query

In this example, a database administrator does this:

```

DROP TABLE emp;
CREATE TABLE emp (
    empno    NUMBER PRIMARY KEY,
    empname VARCHAR2(16),
    salary   NUMBER
);
INSERT INTO emp (empno, empname, salary) VALUES (111, 'Mike', 555);
COMMIT;

DROP TABLE dept;
CREATE TABLE dept (
    deptno   NUMBER,
    deptname VARCHAR2(32)
);
INSERT INTO dept (deptno, deptname) 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 (deptno, deptname) 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 (empno, empname, salary) 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 must diagnose the problem. The database administrator issues this query to retrieve versions of the rows in the `emp` table that correspond to `empno` 111. The query uses Oracle Flashback Version Query pseudocolumns:

```

SELECT versions_xid XID, versions_startscn START_SCN,
       versions_endscn END_SCN, versions_operation OPERATION,
       empname, salary
  FROM emp
 WHERE empno = 111
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE;

```

Results are similar to:

XID	START_SCN	END_SCN	OPERATION	EMPNAME	SALARY
09001100B2200000	10093466		I	Tom	927
030002002B210000	10093459		D	Mike	555
0800120096200000	10093375	10093459	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 030002002B210000 as the erroneous transaction and uses Oracle Flashback Transaction Query to audit all changes made by this transaction:

```
SELECT xid, start_scn, commit_scn, operation, logon_user, undo_sql
FROM flashback_transaction_query
WHERE xid = HEXTORAW('000200030000002D');
```

Results are similar to:

XID	START_SCN	COMMIT_SCN	OPERATION	LOGON_USER
UNDO_SQL				
030002002B210000	10093452	10093459	DELETE	HR
	insert into "HR"."EMP"("EMPNO", "EMPNAME", "SALARY") values ('111', 'Mike', '655');			
030002002B210000	10093452	10093459	INSERT	HR
	delete from "HR"."DEPT" where ROWID = 'AAATjuAAEAAAJsAAB';			
030002002B210000	10093452	10093459	UPDATE	HR
	update "HR"."EMP" set "SALARY" = '555' where ROWID = 'AAATjsAAEAAAJsAAB';			
030002002B210000	10093452	10093459	BEGIN	HR
4 rows selected.				

To make the result of the next query easier to read, the database administrator uses these SQL*Plus commands:

```
COLUMN operation FORMAT A9
COLUMN table_name FORMAT A10
COLUMN table_owner FORMAT A11
```

To see the details of the erroneous transaction and all subsequent transactions, the database administrator performs this 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');
```

Results are similar to:

XID	START_SCN	COMMIT_SCN	OPERATION	TABLE_NAME	TABLE_OWNER
02000E0074200000	10093435	10093446	INSERT	DEPT	HR
030002002B210000	10093452	10093459	DELETE	EMP	HR
030002002B210000	10093452	10093459	INSERT	DEPT	HR
030002002B210000	10093452	10093459	UPDATE	EMP	HR
0800120096200000	10093374	10093375	INSERT	EMP	HR
09001100B2200000	10093462	10093466	UPDATE	EMP	HR
09001100B2200000	10093462	10093466	UPDATE	EMP	HR

```
09001100B2200000 10093462 10093466 INSERT EMP HR
8 rows selected.
```

Note: Because the preceding query does not specify the XID in the WHERE clause, it scans many unrelated rows, degrading performance.

16.7 Using DBMS_FLASHBACK Package

The DBMS_FLASHBACK package provides the same functionality as Oracle Flashback Query, but Oracle Flashback Query is sometimes more convenient.

The DBMS_FLASHBACK package acts as a time machine: you can turn back the clock, perform normal queries as if you were at that earlier time, 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 earlier times.

You must have the EXECUTE privilege on the DBMS_FLASHBACK package.

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 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.

To use a cursor to store the results of queries, open the cursor before invoking DBMS_FLASHBACK.DISABLE. After storing the results and invoking DBMS_FLASHBACK.DISABLE, you can:

- 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 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

16.8 Using Flashback Transaction

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 run the **compensating transactions** that return the affected data to its original state.

The transactions being rolled back are subject to these restrictions:

- They cannot have performed DDL operations that changed the logical structure of database tables.
- They cannot use Large Object (LOB) Data Types:
 - BFILE
 - BLOB
 - CLOB
 - NCLOB
- They cannot use features that LogMiner does not support.

The features that LogMiner supports depends on the value of the COMPATIBLE initialization parameter for the database that is rolling back the transaction. The default value is the release number of the most recent major release.

Flashback Transaction inherits SQL data type support from LogMiner. Therefore, if LogMiner fails due to an unsupported SQL data type in a the transaction, Flashback Transaction fails too.

Some data types, though supported by LogMiner, do not generate undo information as part of operations that modify columns of such types. Therefore, Flashback Transaction does not support tables containing these data types. These include tables with BLOB, CLOB and XML type.

See Also:

- *Oracle Data Guard Concepts and Administration* for information about data type and DDL support on a logical standby database
- *Oracle Database SQL Language Reference* for information about LOB data types
- *Oracle Database Utilities* for information about LogMiner
- *Oracle Database Administrator's Guide* for information about the COMPATIBLE initialization parameter

Topics:

- [Dependent Transactions](#)
- [TRANSACTION_BACKOUT Parameters](#)
- [TRANSACTION_BACKOUT Reports](#)

16.8.1 Dependent Transactions

In the context of Flashback Transaction, transaction 2 can depend on transaction 1 in any of these ways:

- **Write-after-write dependency**

Transaction 1 changes a row of a table, and later transaction 2 changes the same row.

- **Primary key dependency**

A table has a primary key constraint on column c. In a row of the table, column c has the value v. Transaction 1 deletes that row, and later transaction 2 inserts a row into the same table, assigning the value v to column c.

- **Foreign key dependency**

In table b, column b1 has a foreign key constraint on column a1 of table a. Transaction 1 changes a value in a1, and later transaction 2 changes a value in b1.

16.8.2 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 16-2](#)

For the syntax of the TRANSACTION_BACKOUT procedure and detailed parameter descriptions, see *Oracle Database PL/SQL Packages and Types Reference*.

Table 16-2 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 transaction causes an error and appears in *_FLASHBACK_TXN_REPORT.
NOCASCADE_FORCE	Backs out specified transactions, ignoring dependent transactions. Server runs 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.

16.8.3 TRANSACTION_BACKOUT Reports

To see the reports that TRANSACTION_BACKOUT generates, query the static data dictionary views `*_FLASHBACK_TXN_STATE` and `*_FLASHBACK_TXN_REPORT`.

16.8.3.1 *_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 for 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`

16.8.3.2 *_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_REPORT`

16.9 Using Flashback Data Archive

A Flashback Data Archive provides the ability to track and store transactional changes to a table over its lifetime. 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. If you are logged on as `SYSDBA`, 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 specified when the Flashback Data Archive was created.

By default, Flashback Data Archive is not enabled for any tables. You can enable Flashback Data Archive for a table if all of these conditions are true:

- You have the `FLASHBACK ARCHIVE` object privilege on the Flashback Data Archive to use for that table.
- The table is not nested, temporary, remote, or external.
- The table contains neither `LONG` nor nested columns.
- The table does not use any of these Flashback Data Archive reserved words as column names:
 - `STARTSCN`
 - `ENDSCN`
 - `RID`
 - `XID`
 - `OP`

— OPERATION

You cannot enable Flashback Data Archive on a nested table, temporary table, external table, materialized view, Advanced Query (AQ) table, or non-table object.

After Flashback Data Archive is enabled for a table, you can disable it only if you either have the FLASHBACK ARCHIVE ADMINISTER system privilege or you are logged on as SYSDBA.

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.

Effective with Oracle Database 12c Release 1 (12.1.0.1), Flashback Data Archive is enhanced to include the following:

- **User context tracking.** The metadata information for tracking transactions can include (if the feature is enabled) the user context, which makes it easier to determine which user made which changes to a table.

To set the user context level (determining how much user context is to be saved), use the `DBMS_FLASHBACK_ARCHIVE.SET_CONTEXT_LEVEL` procedure. To access the context information, use the `DBMS_FLASHBACK_ARCHIVE.GET_SYS_CONTEXT` function. (The `DBMS_FLASHBACK_ARCHIVE` package is described in *Oracle Database PL/SQL Packages and Types Reference*.)

- **Database hardening.** This feature enables you to associate a set of tables together in an "application", and then enable Flashback Data Archive on all those tables with a single command. Also, database hardening enables you to lock all the tables with a single command, preventing any DML on those tables until they are subsequently unlocked. Database hardening is designed to make it easier to use Flashback Data Archive to track and protect the security-sensitive tables for an application.

To register an application for database hardening, use the `DBMS_FLASHBACK_ARCHIVE.REGISTER_APPLICATION` procedure, which is described in *Oracle Database PL/SQL Packages and Types Reference*.

- **Import and export of history** using `DBMS_FLASHBACK_ARCHIVE` procedures to create a temporary history table, and then later importing that table into the designated history table after loading that table with the desired history data. The temporary history table can be loaded using a variety of methods, including Data Pump. Support is also included for importing user-generated history. If you have been maintaining history using some other mechanism, such as triggers, you can import that history into Flashback Data Archive. (The `DBMS_FLASHBACK_ARCHIVE` package is described in *Oracle Database PL/SQL Packages and Types Reference*.)

Topics:

- [Creating a Flashback Data Archive](#)
- [Altering a Flashback Data Archive](#)
- [Dropping a Flashback Data Archive](#)
- [Specifying the Default Flashback Data Archive](#)
- [Enabling and Disabling Flashback Data Archive](#)
- [DDL Statements on Tables Enabled for Flashback Data Archive](#)
- [Viewing Flashback Data Archive Data](#)
- [Flashback Data Archive Scenarios](#)

16.9.1 Creating a Flashback Data Archive

Create a Flashback Data Archive with the `CREATE FLASHBACK ARCHIVE` statement, specifying:

- 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, error ORA-55621 occurs.
- Retention time (number of days that Flashback Data Archive data for the table is guaranteed to be stored)
- (Optional) Whether to optimize the storage of data in the history tables maintained in the Flashback Data Archive, using [NO] `OPTIMIZE DATA`.

The default is `NO OPTIMIZE DATA`.

If you are logged on as `SYSDBA`, you can also specify that 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 [Section 16.9.4, "Specifying the Default Flashback Data Archive"](#)).

Oracle recommends that all users who must use Flashback Data Archive have unlimited quota on the Flashback Data Archive tablespace; however, if this is not the case, you must grant sufficient quota on that tablespace to those users.

Examples

- Create a default Flashback Data Archive named `f1a1` that uses up to 10 G of tablespace `tbs1`, whose data are retained for one year:

```
CREATE FLASHBACK ARCHIVE DEFAULT f1a1 TABLESPACE tbs1
  QUOTA 10G RETENTION 1 YEAR;
```

- Create a Flashback Data Archive named `f1a2` that uses tablespace `tbs2`, whose data are retained for two years:

```
CREATE FLASHBACK ARCHIVE f1a2 TABLESPACE tbs2 RETENTION 2 YEAR;
```

For more information about the `CREATE FLASHBACK ARCHIVE` statement, see *Oracle Database SQL Language Reference*.

16.9.2 Altering a Flashback Data Archive

With the `ALTER FLASHBACK ARCHIVE` statement, you can:

- 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.

If you are logged on as `SYSDBA`, you can also use the `ALTER FLASHBACK ARCHIVE` statement to make a specific file the default Flashback Data Archive for the system.

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*.

16.9.3 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*.

16.9.4 Specifying the Default Flashback Data Archive

By default, the system has no default Flashback Data Archive. If you are logged on as SYSDBA, you can specify default Flashback Data Archive in either of these 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

16.9.5 Enabling and Disabling Flashback Data Archive

By default, Flashback Data Archive is disabled for any table. You can enable Flashback Data Archive for a table if you have the FLASHBACK ARCHIVE object privilege on the Flashback Data Archive to use for that table.

To enable Flashback Data Archive 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 are stored. The default is the default Flashback Data Archive for the system. If you specify a nonexistent Flashback Data Archive, an error occurs.

If you enable Flashback Data Archive for a table, but AUM is disabled, error ORA-55614 occurs when you try to modify the table.

If a table has Flashback Data Archive enabled, and you try to enable it again with a different Flashback Data Archive, an error occurs.

After Flashback Data Archive is enabled for a table, you can disable it only if you either have the FLASHBACK ARCHIVE ADMINISTER system privilege or you are logged on as SYSDBA. To disable Flashback Data Archive 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.)

After enabling Flashback Data Archive on a table, Oracle recommends waiting at least 20 seconds before inserting data into the table, and waiting up to 5 minutes before using Flashback Query on the table.

See Also: *Oracle Database SQL Language Reference* for more information about the FLASHBACK ARCHIVE clause of the CREATE TABLE statement, including restrictions on its use

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 `f1a1`:

```
CREATE TABLE employee (EMPNO NUMBER(4) NOT NULL, ENAME VARCHAR2(10),
  JOB VARCHAR2(9), MGR NUMBER(4)) FLASHBACK ARCHIVE f1a1;
```

- Enable Flashback Data Archive for the table `employee` and store the historical data in the default Flashback Data Archive:

```
ALTER TABLE employee FLASHBACK ARCHIVE;
```

- Enable Flashback Data Archive for the table `employee` and store the historical data in the Flashback Data Archive `f1a1`:

```
ALTER TABLE employee FLASHBACK ARCHIVE f1a1;
```

- Disable Flashback Data Archive for the table `employee`:

```
ALTER TABLE employee NO FLASHBACK ARCHIVE;
```

16.9.6 DDL Statements on Tables Enabled for Flashback Data Archive

Flashback Data Archive supports only these DDL statements:

- ALTER TABLE statement that does any of the following:
 - Adds, drops, renames, or modifies a column
 - Adds, drops, or renames a constraint
 - Drops or truncates a partition or subpartition operation
- TRUNCATE TABLE statement
- RENAME statement that renames a table

Flashback Data Archive does not support DDL statements that move, split, merge, or coalesce partitions or subpartitions, move tables, or convert `LONG` columns to `LOB` columns.

For example, the following DDL statements cause error ORA-55610 when used on a table enabled for Flashback Data Archive:

- ALTER TABLE statement that includes an UPGRADE TABLE clause, with or without an INCLUDING DATA clause
- ALTER TABLE statement that moves or exchanges a partition or subpartition operation
- DROP TABLE statement

If you must use unsupported DDL statements on a table enabled for Flashback Data Archive, use the `DBMS_FLASHBACK_ARCHIVE.DISASSOCIATE_FBA` procedure to disassociate the base table from its Flashback Data Archive. To reassociate the Flashback Data Archive with the base table afterward, use the `DBMS_FLASHBACK_ARCHIVE.REASSOCIATE_FBA` procedure. Also, to drop a table enabled for Flashback Data

Archive, you must first disable Flashback Data Archive on the table by using the `ALTER TABLE ... NO FLASHBACK ARCHIVE` clause.

See Also:

- *Oracle Database SQL Language Reference* for information about the `ALTER TABLE` statement
- *Oracle Database SQL Language Reference* for information about the `TRUNCATE TABLE` statement
- *Oracle Database SQL Language Reference* for information about the `RENAME` statement
- *Oracle Database SQL Language Reference* for information about the `DROP TABLE` statement
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_FLASHBACK_ARCHIVE` package

16.9.7 Viewing Flashback Data Archive Data

[Table 16-3](#) lists and briefly describes the static data dictionary views that you can query for information about Flashback Data Archive files.

Table 16-3 Static Data Dictionary Views for Flashback Data Archive Files

View	Description
<code>*_FLASHBACK_ARCHIVE</code>	Displays information about Flashback Data Archive files.
<code>*_FLASHBACK_ARCHIVE_TS</code>	Displays tablespaces of Flashback Data Archive files.
<code>*_FLASHBACK_ARCHIVE_TABLES</code>	Displays information about tables that are enabled for Data Flashback Archive files.

See Also:

- *Oracle Database Reference* for detailed information about `*_FLASHBACK_ARCHIVE`
- *Oracle Database Reference* for detailed information about `*_FLASHBACK_ARCHIVE_TS`
- *Oracle Database Reference* for detailed information about `*_FLASHBACK_ARCHIVE_TABLES`

16.9.8 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](#)

16.9.8.1 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.)

16.9.8.2 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 `f1a1` that uses up to 10 G of tablespace `tbs1`, whose data are retained for five years (you must be logged on as `SYSDBA`):

```
CREATE FLASHBACK ARCHIVE DEFAULT f1a1 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 this 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 this 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;
```

16.9.8.3 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 `f1a2` that uses up to 20 G of tablespace `tbs1`, whose data are retained for five years (you must be logged on as `SYSDBA`):

```
CREATE FLASHBACK ARCHIVE DEFAULT f1a2 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 this query:

```
SELECT * FROM investments AS OF
  TIMESTAMP TO_TIMESTAMP ('2006-12-31 16:00:00', 'YYYY-MM-DD HH24:MI:SS')
  WHERE name = 'LISA';
```

16.9.8.4 Scenario: Using Flashback Data Archive for Auditing

A medical insurance company must audit a medical clinic. The medical insurance company has its claims in the table `Billings`, and creates a default Flashback Data Archive named `f1a4` that uses up to 100 G of tablespace `tbs1`, whose data are retained for 10 years:

```
CREATE FLASHBACK ARCHIVE DEFAULT f1a4 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 this query:

```
SELECT date_billed, amount_billed, patient_name, claim_Id,
  test_costs, diagnosis
  FROM Billings
  AS OF TIMESTAMP
  TO_TIMESTAMP('2007-05-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS');
```

16.9.8.5 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.

At first, Bob thinks he cannot return the `employee` table to its prior state without going to the backups. Then he remembers that the `employee` table has Flashback Data Archive enabled.

First, he verifies that no other transaction modified the `employee` table after his: The commit time stamp from the transaction query corresponds to Bob's transaction, two days ago.

Next, Bob uses these statements to return the `employee` table to the way it was before his erroneous change:

```
DELETE EMPLOYEE WHERE MANAGER = 'LISA JOHNSON';
INSERT INTO EMPLOYEE
  SELECT * FROM EMPLOYEE
  AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '2' DAY)
  WHERE MANAGER = 'LISA JOHNSON';
```

Bob then reexecutes the update that Lisa had requested.

16.10 General Guidelines for Oracle 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 Oracle Flashback Query, Oracle Flashback Version Query, or Oracle Flashback Transaction Query for SQL code that you write, for convenience. An Oracle 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 time stamp or SCN argument. For example, add or subtract an INTERVAL value to the value of the SYSTIMESTAMP function.
- Use Oracle Flashback Query, Oracle Flashback Version Query, and Oracle Flashback Transaction Query locally or remotely. An example of a remote Oracle Flashback Query is:

```
SELECT * FROM employees@some_remote_host AS OF
    TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE);
```

- To ensure database consistency, 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, error ORA-01466 occurs. DDL operations that alter the storage attributes of a table (such as PCTFREE, INITTRANS, and MAXTRANS) do not invalidate undo data.
- To query past data at a precise time, use an SCN. If you use a time stamp, 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 time stamps at a granularity of 3 seconds.

For example, suppose that the SCN values 1000 and 1005 are mapped to the time stamps 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; an Oracle Flashback Query for 8:46 AM is mapped to SCN 1005. Therefore, 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 immediately before the DDL operation, causing error ORA-01466.

- You cannot retrieve past data from a dynamic performance (V\$) view. A query on such a view returns current data.
- You can perform queries on past data in static data dictionary views, such as *_TABLES.
- You can enable optimization of data storage for history tables maintained by Flashback Data Archive by specifying OPTIMIZE DATA when creating or altering a Flashback Data Archive.

OPTIMIZE DATA optimizes the storage of data in history tables by using any of these features:

- Advanced Row Compression
- Advanced LOB Compression
- Advanced LOB Deduplication
- Segment-level compression tiering
- Row-level compression tiering

The default is not to optimize the storage of data in history tables.

16.11 Performance Guidelines for Oracle Flashback Technology

- Use the DBMS_STATS package to generate statistics for all tables involved in an Oracle Flashback Query. Keep the statistics current. Oracle 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 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 operations are CPU-bound.

Oracle recommends that you have enough buffer cache, so that the versions query for the archiver finds the undo data in the buffer cache. Buffer cache access is significantly faster than disk access.

- If very large transactions (such as affecting more than 1 million rows) are performed on tracked tables, set the large pool size high enough (at least 1 GB) for Parallel Query not to have to allocate new chunks out of the SGA. (For information about setting the large pool size, see the chapter about tuning the shared pool and the large pool in *Oracle Database Performance Tuning Guide*.)
- For Oracle Flashback Version Query, use index structures. Oracle Database keeps undo data for index changes and data changes. Performance of index lookup-based Oracle Flashback Version Query is an order of magnitude faster than the full table scans that are otherwise needed.
- In an Oracle 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).
- An Oracle Flashback Query against a materialized view does not take advantage of query rewrite optimization.

16.12 Multitenant Container Database Restrictions for Oracle Flashback Technology

These Oracle Flashback Technology features are unavailable for a multitenant container database (CDB):

- For Oracle Database 12c Release 1 (12.1.0.1), Flashback Data Archive (FDA) is not supported in a CDB.

For Oracle Database 12c Release 1 (12.1.0.2), this restriction is removed.

- Flashback Transaction Query is not supported in a CDB.
- Flashback Transaction Backout is not supported in a CDB.

Choosing a Programming Environment

To choose a programming environment for a development project, read:

- The topics in this chapter and the documents to which they refer.
- The platform-specific documents that explain which compilers and development tools your platforms support.

Sometimes the choice of programming environment is obvious, for example:

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

If no programming language provides all the features you need, you can use multiple programming languages, because:

- Every programming language in this chapter can invoke PL/SQL and Java stored subprograms. (Stored subprograms include triggers and ADT methods.)
- PL/SQL, Java, SQL, and Oracle Call Interface (OCI) can invoke external C subprograms.
- External C subprograms can access Oracle Database using SQL, OCI, or Pro*C (but not C++).

For more information about multilanguage programming, see [Chapter 18, "Developing Applications with Multiple Programming Languages."](#)

Topics:

- [Overview of Application Architecture](#)
- [Overview of the Program Interface](#)
- [Overview of PL/SQL](#)
- [Overview of Oracle Database Java Support](#)
- [Choosing PL/SQL or Java](#)
- [Overview of Precompilers](#)
- [Overview of OCI and OCCI](#)
- [Choosing a Precompiler or OCI](#)
- [Overview of Oracle Data Provider for .NET \(ODP.NET\)](#)
- [Overview of OraOLEDB](#)

17.1 Overview of Application Architecture

In this topic, **application architecture** refers to the computing environment in which a database application connects to an Oracle Database.

Topics:

- [Client/Server Architecture](#)
- [Server-Side Programming](#)
- [Two-Tier and Three-Tier Architecture](#)

See Also: *Oracle Database Concepts* for more information about application architecture

17.1.1 Client/Server Architecture

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.

See Also: *Oracle Database Concepts* for more information about client/server architecture

17.1.2 Server-Side Programming

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 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.

See Also: *Oracle Database Concepts* for more information about server-side programming

17.1.3 Two-Tier and Three-Tier Architecture

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.

See Also: *Oracle Database Concepts* for more information about multitier architecture

17.2 Overview of the Program Interface

The **program interface** is the software layer between a database application and Oracle Database. The program interface:

- Provides a security barrier, preventing destructive access to the SGA by client user processes
- Acts as a communication mechanism, formatting information requests, passing data, and trapping and returning errors
- Converts and translates data, particularly between different types of computers or to external user program data types

The Oracle code acts as a server, performing database tasks on behalf of an application (a client), such as fetching rows from data blocks. The program interface consists of several parts, provided by both Oracle Database software and operating system-specific software.

See Also: *Oracle Database Concepts* for more information about the program interface

Topics:

- [User Interface](#)
- [Stateful and Stateless User Interfaces](#)

17.2.1 User Interface

The **user interface** is what your application displays to end users. It depends on the technology behind the application and 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 traditional user interface can also be provided in a web browser through HTML and Java.

17.2.2 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 not be entered again. When the application can 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, enabling you to construct a wizard-like interface that remembers user choices through several different steps. You can use cookies to store small items of information about the client system, and retrieve them when the user returns to a website. You can use servlets to keep a database session open and store variables between requests from the same client.

17.3 Overview of PL/SQL

PL/SQL, the Oracle procedural extension of SQL, is a completely portable, high-performance transaction-processing language. PL/SQL lets you manipulate data with SQL statements; control program flow with conditional selection and loops; declare constants and variables; define subprograms; define types, subtypes, and ADTs and declare variables of those types; and trap runtime errors.

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

You can use PL/SQL to develop stored procedures that can be invoked by a web client.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about the advantages, main features, and architecture of PL/SQL

17.4 Overview of Oracle Database Java Support

This section provides an overview of Oracle 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 enable data-intensive Java code to run within the database.

Topics:

- [Overview of Oracle JVM](#)
- [Overview of Oracle JDBC](#)
- [Overview of Oracle SQLJ](#)
- [Comparing Oracle JDBC and Oracle SQLJ](#)
- [Overview of Oracle JPublisher](#)
- [Overview of Java Stored Subprograms](#)
- [Overview of Oracle Database Web Services](#)

See Also:

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

17.4.1 Overview of Oracle JVM

Oracle JVM, the Java Virtual Machine provided with the Oracle Database, is compliant with the J2SE version 1.5.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 these benefits:

- Java applications have the same session isolation and data integrity as SQL operations.
- You need not 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).

Effective with Oracle Database 12c Release 1 (12.1.0.1), Oracle JVM provides complete support for the latest Java Standard Edition. Compatibility with latest Java standards increases application portability and enables direct execution of client-side Java classes in the database.

See Also:

- *Oracle Database Concepts* for additional general information about Oracle JVM
- *Oracle Database Java Developer's Guide* for information about support for the latest Java Standard Edition

17.4.2 Overview of Oracle JDBC

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

Oracle Database includes these extensions to the JDBC 1.22 standard:

- Support for Oracle data types
- 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 highlights include datasources, JTA, and distributed transactions.

Oracle Database supports these features from the JDBC 3.0 standard:

- Support for JDK 1.5.
- Toggling between local and global transactions.
- Transaction savepoints.
- Reuse of prepared statements by connection pools.

Note: JDBC code and SQLJ code interoperate. For more information, see [Section 17.4.4](#).

Topics:

- [Oracle JDBC Drivers](#)
- [Sample JDBC 2.0 Program](#)
- [Sample Pre-2.0 JDBC Program](#)

See Also: *Oracle Database Concepts* for additional general information about Java support in Oracle Database

17.4.2.1 Oracle JDBC Drivers

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 Part 3 of the SQL standard, "Call-Level Interface."

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.

Topics:

- [JDBC Thin Driver](#)
- [JDBC OCI Driver](#)
- [JDBC Server-Side Internal Driver](#)

See Also:

- *Oracle Database Concepts* for additional general information about JDBC drivers
- *Oracle Database JDBC Developer's Guide* for more information about JDBC

17.4.2.1.1 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 run only in a browser that supports sockets.

17.4.2.1.2 JDBC OCI Driver The JDBC OCI driver is a Type 2 JDBC driver. It makes calls to OCI written in C to interact with Oracle Database, thus using native and Java methods.

The OCI driver provides 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 Oracle8i or later including Oracle Net, OCI libraries, CORE libraries, and all other dependent files. The OCI driver usually runs 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.

17.4.2.1.3 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. 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.

17.4.2.2 Sample JDBC 2.0 Program

This 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();
```

17.4.2.3 Sample Pre-2.0 JDBC Program

This source code registers an Oracle JDBC thin driver, connects to the database, creates a Statement object, runs 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", "password");

        // 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, database information, row prefetching, and execution batching.

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

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

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

17.4.3 Overview of Oracle SQLJ

Note: In this guide, **SQLJ** refers to Oracle SQLJ and its extensions.

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 and static SQL. Support for dynamic SQL is an Oracle extension to the SQLJ standard.

Oracle Database provides a translator and a runtime driver to support SQLJ. The SQLJ translator is 100% pure Java and is portable to any JVM that is compliant with JDK version 6 or higher.

The Oracle SQLJ translator performs these tasks:

- Translates SQLJ source to Java code with calls to the SQLJ runtime 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 run in the Oracle JVM. SQLJ is integrated with JDeveloper. Source-level debugging support for SQLJ is available in JDeveloper.

This 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: *Oracle Database SQLJ Developer's Guide* for more examples and details about Oracle SQLJ syntax

Topics:

- [Benefits of SQLJ](#)
- [SQLJ Stored Subprograms in the Server](#)

See Also: *Oracle Database Concepts* for additional general information about SQLJ

17.4.3.1 Benefits of SQLJ

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

In particular, Oracle SQLJ does this:

- 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 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 Database type extensions.

17.4.3.2 SQLJ Stored Subprograms in the Server

SQLJ applications can be stored and executed in the server by using these 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.

17.4.4 Comparing Oracle JDBC and Oracle SQLJ

JDBC code and SQLJ code interoperate, enabling 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.

Some differences between JDBC and SQLJ are:

- 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.
- 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 provides type-checking on calls. JDBC passes values to and from SQL without compile-time type checking.
- SQLJ programs enable 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 four examples show, on successive lines, how to call a stored procedure or a stored function using either JDBC escape syntax or Oracle JDBC syntax:

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

The SQLJ equivalent is:

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

These 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 Database ADTs and collections.
- PL/SQL and Java stored subprograms can be used interchangeably.

17.4.5 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 ADTs, 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 database object type instances to and from the database transparently.

See Also:

- *Oracle Database Concepts* for additional general information about Oracle JPublisher
- *Oracle Database JPublisher User's Guide* for complete information about Oracle JPublisher

17.4.6 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 using an execution model similar to that of PL/SQL.

See Also:

- *Oracle Database Concepts* for additional general information about Java stored subprograms
- *Oracle Database Java Developer's Guide* for complete information about Java stored subprograms

17.4.7 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, and which enable applications to interact 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 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. Also, 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

See Also: *Oracle Database Concepts* for additional general information about Oracle Database as a web service provider

17.5 Choosing PL/SQL or Java

PL/SQL and Java interoperate in the server. You can run a PL/SQL package from Java or PL/SQL can be invoked from Java, so that either one can be invoked from distributed CORBA and Enterprise Java Beans clients.

[Table 17-1](#) shows PL/SQL packages and their Java equivalents.

Table 17-1 PL/SQL Packages and Their Java Equivalents

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 <code>oracle.aurora.rdbms.OracleDBMSOutputStream</code> or Java stored subprogram <code>DBMS_JAVA.SET_STREAMS</code> .
DBMS_PIPE	Call with SQLJ or JDBC.
DBMS_SESSION	Use JDBC to run an <code>ALTER SESSION</code> statement.
DBMS_SNAPSHOT	Call with SQLJ or JDBC.
DBMS_SQL	Use JDBC.
DBMS_TRANSACTION	Use JDBC to run an <code>ALTER SESSION</code> statement.
DBMS.Utility	Call with SQLJ or JDBC.
UTL_FILE	Grant the <code>JAVAUSERPRIV</code> privilege and then use Java I/O entry points.

Topics:

- [Similarities of PL/SQL and Java](#)
- [PL/SQL Advantages Over Java](#)
- [Java Advantages Over PL/SQL](#)

17.5.1 Similarities of PL/SQL and Java

Both PL/SQL and Java provide packages and libraries.

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

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

17.5.2 PL/SQL Advantages Over Java

As an extension of SQL, PL/SQL supports all SQL data types, data encapsulation, information hiding, overloading, and exception-handling. Therefore:

- SQL data types are easier to use in PL/SQL than in Java.
- SQL operations are faster with PL/SQL than with Java, especially when a large amount of data is involved, when mostly database access is done, or when bulk operations are used.

Some advanced PL/SQL capabilities are unavailable for Java in Oracle9*i* (for example, autonomous transactions and the `dblink` facility for remote databases).

Code development is usually faster in PL/SQL than in Java; however, this really depends upon the development tool or development environment you are using.

PL/SQL is preferred when your data logic is SQL intensive. That is, the data processing or data validation requirements of your application are high.

Also, there is a large user base with Oracle-supplied packages and third party libraries that can draw upon for development.

17.5.3 Java Advantages Over PL/SQL

Java is used for open distributed applications, and many Java-based development tools are available throughout the industry. Java has native mechanisms that are unavailable in PL/SQL. For example, Java has built-in security mechanisms, an automatic Garbage Collector, type safety mechanisms, byte-code verifier, and Java 2 security. Also, Java provides built-in rapid development features, such as, built-in automatic bounds checking on arrays, built-in network access classes, and APIs that contain many useful and ready-to-use classes. Java has a vast set of class libraries, tools, and third-party class libraries that can be reused in the database. Java has a richer type system than PL/SQL. Java can use CORBA (which can have many different computer languages in its clients) and Enterprise Java Beans. PL/SQL packages can be invoked from CORBA or Enterprise Java Beans clients. You can run XML tools, the Internet File System, or JavaMail from Java.

If your application must interact with ERP systems, RMI servers, Java/J2EE, and web services, Java is preferred because none of these things can be accomplished with PL/SQL. Java is also preferred if you must develop part of your application in the middle-tier because your business logic is complex or compute intensive with little to moderate direct SQL access, you are implementing a middle-tier-driven presentation logic, your application requires transparent Java persistence, or your application requires container-managed infrastructure services. Thus, when needing to partition your application between the database tier and middle tier, migrate that part of your application as needed to the middle tier and use Java/J2EE.

17.6 Overview of Precompilers

Client/server programs are typically written using **precompilers**, which are programming tools that let you embed SQL statements in high-level programs written in languages such as C, C++, or COBOL. Because the client application hosts the SQL statements, it is called a **host program**, and the language in which it is written is called the **host language**.

A precompiler accepts the host program as input, translates the embedded SQL statements into standard database runtime library calls, and generates a source program that you can compile, link, and run in the usual way.

Topics:

- [Overview of the Pro*C/C++ Precompiler](#)
- [Overview of the Pro*COBOL Precompiler](#)

See Also: *Oracle Database Concepts* for additional general information about Oracle precompilers

17.6.1 Overview of the Pro*C/C++ Precompiler

For the Pro*C/C++ precompiler, the host language is either C or C++. Some features of the Pro*C/C++ precompiler are:

- 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, and at runtime.
- 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 run 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 data types and C language data types.
- 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 enables 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 ADTs and named collection types in your database to structures and headers that you include in your source.

- Three kinds of collection types: associative arrays, nested tables and VARRAY, are supported with a set of SQL statements that give you 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 run 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 optimize the performance of Pro*C/C++ application. The connection pool option is not enabled by default.

See Also: *Pro*C/C++ Precompiler Programmer's Guide* for complete information about the Pro*C/C++ precompiler

Example 17-1 is a code fragment from a C source program that queries the table employees in the schema hr.

Example 17-1 Pro*C/C++ Application

```
...
#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 inline inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can compile, link, and run 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.

17.6.2 Overview of the Pro*COBOL Precompiler

For the Pro*COBOL precompiler, the host language is COBOL. Some features of the Pro*COBOL precompiler are:

- 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, and at runtime.
- You can conditionally precompile sections of your code so that they can run 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 enables users to input variable values and statement syntax.

See Also: *Pro*COBOL Programmer's Guide* for complete information about the Pro*COBOL precompiler

Example 17–2 is a code fragment from a COBOL source program that queries the table employees in the schema hr.

Example 17–2 Pro*COBOL Application

```
...
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 inline inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can compile, link, and run 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.

17.7 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 Database objects
- Access to external databases
- Ability to develop applications that service an increasing number of users and requests without additional hardware investments

OCI lets you manipulate data and schemas in a database using a host programming language, such as C. OCCI is an object-oriented interface suitable for use with C++. These APIs provide a library of standard database access and retrieval functions in the form of a dynamic runtime library that can be linked in an application at runtime. You need not 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 Database 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](#)

17.7.1 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 Database 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 arrays.
- Ability to associate a commit request with a statement execution 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 code when a call is still running or cannot complete.

17.7.2 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 Database types
External subprogram	To write C callbacks from PL/SQL (OCI only)

17.7.3 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 unavailable 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.

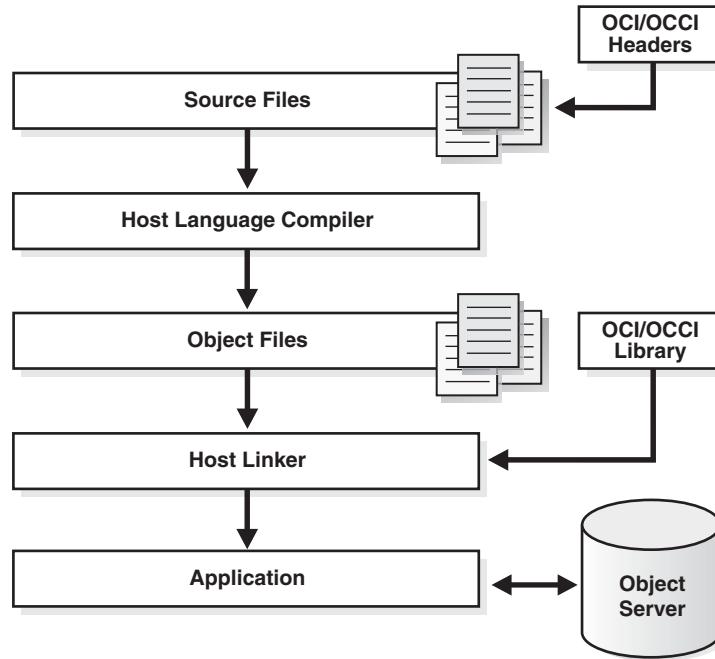
Alternatively, you can use PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written

in SQL alone. OCI and OCCI also provide facilities for accessing and manipulating objects in Oracle Database.

17.7.4 Building an OCI or OCCI Application

As Figure 17-1 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 17-1 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.

17.8 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 enables 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 enables 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 data types.
- OCI calls can be embedded in a Pro*C/C++ application.

17.9 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 enables native providers such as ODP.NET to expose specific features and data types specific to Oracle Database.

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

This 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=password;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*.

17.10 Overview of OraOLEDB

Oracle Provider for OLE DB (OraOLEDB) is an OLE DB data provider that offers high performance and efficient access to Oracle data by OLE DB consumers. In general, this developer's guide assumes that you are using OraOLEDB through OLE DB or ADO.

See Also: *Oracle Provider for OLE DB Developer's Guide for Microsoft Windows*

Developing Applications with 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](#)
- [What Is an External Procedure?](#)
- [Overview of Call Specification for External Procedures](#)
- [Loading External Procedures](#)
- [Publishing External Procedures](#)
- [Publishing Java Class Methods](#)
- [Publishing External C Procedures](#)
- [Locations of Call Specifications](#)
- [Passing Parameters to External C Procedures with Call Specifications](#)
- [Running External Procedures with CALL Statements](#)
- [Handling Errors and Exceptions in Multilanguage Programs](#)
- [Using Service Routines with External C Procedures](#)
- [Doing Callbacks with External C Procedures](#)

18.1 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++, through the Oracle C++ Call Interface (OCCI), as described in the *Oracle C++ 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 Provider for OLE DB, as described in *Oracle Provider for OLE DB Developer's Guide for Microsoft Windows*.
- .NET, through Oracle Data Provider for .NET, as described in *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*
- Java, through the JDBC and SQLJ client-side application programming interfaces (APIs). See *Oracle Database JDBC Developer's Guide* and *Oracle Database SQLJ Developer's Guide*.
- Java in the database, as described in *Oracle Database Java Developer's Guide*. This includes the use of Java stored procedures (Java methods published to SQL and stored in the database), as described in a chapter in *Oracle Database Java Developer's Guide*.

The JPublisher utility is also available for generating Java classes to represent database entities, such as SQL objects and PL/SQL packages, in a Java client program; publishing from SQL, PL/SQL, and server-side Java to web services; and enabling the invocation of external web services from inside the database. See *Oracle Database JPublisher User'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 must 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.
- For both portability and security, you might select Java.
- For familiarity with Microsoft programming languages, you might select .NET.

Most significantly for performance, only PL/SQL and Java methods run within the address space of the server. C/C++ and .NET 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 Oracle Provider for OLE DB and subsequently OCI, which is implemented in C.

Taking all these factors into account suggests that there might be situations in which you might need to implement your application in multiple languages. For example, because Java runs within the address space of the server, 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 if C can call your procedure, then SQL or PL/SQL can use it. Therefore, if you have a candidate C++ procedure, use a C++ `extern "C"` statement in that procedure to make it callable by C.

Therefore, 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.

18.2 What Is an External Procedure?

An **external procedure** is a procedure stored in a dynamic link library (DLL). 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 runtime, 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. 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 problems on the client side do not adversely affect the database
- Move computation-bound programs from client to server where they run 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

Note: The external library (DLL file) must be statically linked. In other words, it must not reference external symbols from other external libraries (DLL files). Oracle Database does not resolve such symbols, so they can cause your external procedure to fail.

See Also: *Oracle Database Security Guide* for information about securing external procedures

18.3 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. (Java class methods are not external procedures, but they still use call specifications.)

Note: To support legacy applications, call specifications also enable you to publish with the AS EXTERNAL clause. For application development, however, using the AS LANGUAGE clause is recommended.

In general, call specifications enable:

- Dispatching the appropriate C or Java target procedure
- Data type conversions

- Parameter mode mappings
- Automatic memory allocation and cleanup
- Purity constraints to be specified, where necessary, for package 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 existing program as an external procedure, load, publish, and then call it.

18.4 Loading External Procedures

To make your external C procedures or Java methods available to PL/SQL, you must first load them.

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 this 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 [Section 18.5.2.4, "CALLING STANDARD."](#)

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 these:

- A multithreaded extproc agent
- Oracle Database in shared 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 follow these steps:

1. [Define the C Procedures](#)
2. [Set Up the Environment](#)
3. [Identify the DLL](#)
4. [Publish the External Procedures](#)

18.4.1 Define the C Procedures

Define the C procedures using one of these prototypes:

- Kernighan & Ritchie style prototypes; for example:

```
void C_findRoot(x)
  float x;
  ...
```

- ISO/ANSI prototypes other than numeric data types that are less than full width (such as `float`, `short`, `char`); for example:

```
void C_findRoot(double x)
  ...
```

- Other data types that do not change size under default argument promotions.

This example changes size under default argument promotions:

```
void C_findRoot(float x)
  ...
```

18.4.2 Set Up the Environment

When you use the default configuration for external procedures, Oracle Database spawns extproc directly. You need not 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 systems and at ORACLE_HOME\hs\admin on Windows), using this 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 these 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 (semicolon-separated on Windows systems) 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 (semicolon-separated on Windows systems) 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.

Set the ENFORCE_CREDENTIAL environment variable, which enforces the usage of credentials when spawning an extproc process. The ENFORCE_CREDENTIAL value can be TRUE or FALSE (the default). For a discussion of ENFORCE_CREDENTIAL and the expected behaviors of an extproc process based on possible authentication and impersonation scenarios, see the information about securing external procedures in *Oracle Database Security Guide*.

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.

Note: To use credentials for extproc, you cannot use Oracle Listener to spawn the extproc agent.

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 is to 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 is to run in dedicated mode, additional configuration of the agent process is not necessary.

If the agent is to 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 configuration, use the agent control utility, `agtctl`. For example, start `extproc` using this command:

```
agtctl startup extproc agent_sid
```

where `agent_sid` is the system identifier that this `extproc` agent services. 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 in Oracle Call Interface Programmer's Guide*.

Note:

- If you use a multithreaded `extproc` agent, the library you call must be thread-safe—to avoid errors such as a damaged 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 F-1 in *Oracle Call Interface Programmer's Guide* illustrates the architecture of the multithreaded `extproc` agent.

18.4.3 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 this syntax:

```
CREATE LIBRARY [schema_name.]library_name
  {IS | AS} 'file_path'
  [AGENT 'agent_link'];
```

Note: The `ANY` privileges are very powerful and must not be granted lightly. For more information, see:

- *Oracle Database Security Guide* for information about managing system privileges, including `ANY`
 - *Oracle Database Security Guide* for guidelines for securing user accounts and privileges
-

Oracle recommends that you specify the path to the DLL using a directory object, rather than only the DLL name. In this example, you create alias library `c_utils`, which represents `DLL utils.so`:

```
CREATE LIBRARY C_utils AS 'utils.so' IN DLL_DIRECTORY;
```

where `DLL_DIRECTORY` is a directory object that refers to `'/DLLs'`.

As an alternative, you can specify the full path to the DLL, as in this example:

```
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 this example, the agent specified by the name `agent_link` is used to run any external procedure in the library `C_Utils`:

```
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';
```

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.

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 run 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 (semicolon-separated on Windows systems) list of DLL names qualified with the complete path. For example:

```
EXTPROC_DLLS=/private1/home/john Doe/dll/myDll.so:/private1/home/john Doe/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 variables 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 about the EXTPROC feature.

Note: In extproc.ora on a Windows system, specify the path using a drive letter and using a double backslash (\\) for each backslash in the path. (The first backslash in each double backslash serves as an escape character.)

18.4.4 Publish the External Procedures

You find or write an external C procedure, and add it to the DLL. When the procedure is in the DLL, you publish it using the call specification mechanism described in [Section 18.5, "Publishing External Procedures."](#)

18.5 Publishing External Procedures

Oracle Database can use only 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}

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*] |

```
[ NAME c_string_literal_name ] LIBRARY library_name }
[ AGENT IN ( argument [, argument]... ) ]
[ 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:

- [Section 18.5.1, "AS LANGUAGE Clause for Java Class Methods"](#)
- [Section 18.5.2, "AS LANGUAGE Clause for External C Procedures"](#)

18.5.1 AS LANGUAGE Clause for Java Class Methods

The AS LANGUAGE clause is the interface between PL/SQL and a Java class method.

18.5.2 AS LANGUAGE Clause for External C Procedures

These subclauses tell PL/SQL where to locate the external C procedure, how to call it, and what to pass to it:

- [LIBRARY](#)
- [NAME](#)
- [LANGUAGE](#)
- [CALLING STANDARD](#)
- [WITH CONTEXT](#)
- [PARAMETERS](#)
- ["AGENT IN"](#)

Of the preceding subclauses, only LIBRARY is required.

18.5.2.1 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 quotation marks, then it becomes case-sensitive. (By default, the name is stored in upper case.) You must have EXECUTE privileges on the alias library.

18.5.2.2 NAME

Specifies the external C procedure to be called. If you enclose the procedure name in double quotation marks, 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.

18.5.2.3 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.

18.5.2.4 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.

18.5.2.5 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.

18.5.2.6 PARAMETERS

Specifies the positions and data types 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).

18.5.2.7 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 runtime through `AGENT IN` allows greater flexibility.

When the agent name is specified this way, it overrides any agent name declared in the alias library. If no agent name is specified, the default is the `extproc` agent on the same instance as the calling program.

18.6 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 multiple procedures.

The `NAME`-clause string uniquely identifies the Java method. The PL/SQL function or procedure and Java must have corresponding 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 only selected public static methods can be explicitly published to SQL. However, all methods can be invoked from other Java classes residing in the database, provided they have proper authorization.

Suppose you want to publish this 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);
    }
}
```

This 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';
```

18.7 Publishing External C Procedures

In this 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. */
```

18.8 Locations of Call Specifications

For both Java class methods and external C procedures, call specifications can be specified in any of these locations:

- Standalone PL/SQL procedures
- PL/SQL Package Specifications
- PL/SQL Package Bodies
- ADT Specifications
- ADT Bodies

Topics:

- [Example: Locating a Call Specification in a PL/SQL Package](#)
- [Example: Locating a Call Specification in a PL/SQL Package Body](#)
- [Example: Locating a Call Specification in an ADT Specification](#)
- [Example: Locating a Call Specification in an ADT Body](#)
- [Example: Java with AUTHID](#)

- [Example: C with Optional AUTHID](#)
- [Example: Mixing Call Specifications in a Package](#)

Note: In these examples, the AUTHID and SQL_NAME_RESOLVE clauses might 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

18.8.1 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;
```

18.8.2 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;
```

18.8.3 Example: Locating a Call Specification in an ADT Specification

Note: For examples in this topic to work, you must set up this data structure (which requires that you have the privilege CREATE ANY LIBRARY):

```
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)
);

```

18.8.4 Example: Locating a Call Specification in an ADT 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;

```

18.8.5 Example: Java with AUTHID

Here is an example of 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)';

```

18.8.6 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);

```

18.8.7 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;

```

18.9 Passing Parameters to External C Procedures with Call Specifications

Call specifications allow a mapping between PL/SQL and C data types. See [Section 18.9.1](#) for data type mappings.

Passing parameters to an external C procedure is complicated by several circumstances:

- The available set of PL/SQL data types does not correspond one-to-one with the set of C data types.
- 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.

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 Data Types](#)
- [External Data Type Mappings](#)
- [Passing Parameters BY VALUE or BY REFERENCE](#)
- [Declaring Formal Parameters](#)
- [Overriding Default Data Type Mapping](#)
- [Specifying Properties](#)

18.9.1 Specifying Data Types

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 data types for the parameters. PL/SQL data types map to default external data types, as shown in [Table 18–1](#).

Note: The PL/SQL data types BINARY_INTEGER and PLS_INTEGER are identical. For simplicity, this guide uses "PLS_INTEGER" to mean both BINARY_INTEGER and PLS_INTEGER.

Table 18–1 Parameter Data Type Mappings

PL/SQL Data Type	Supported External Types	Default External Type
BINARY_INTEGER	[UNSIGNED] CHAR	INT
BOOLEAN	[UNSIGNED] SHORT	
PLS_INTEGER	[UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	
NATURAL ¹	[UNSIGNED] CHAR	UNSIGNED INT
NATURALN ¹	[UNSIGNED] SHORT	
POSITIVE ¹	[UNSIGNED] INT	
POSITIVEN ¹	[UNSIGNED] LONG	
SIGNTYPE ¹	SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	
FLOAT	FLOAT	FLOAT
REAL		
DOUBLE PRECISION	DOUBLE	DOUBLE

Table 18–1 (Cont.) Parameter Data Type Mappings

PL/SQL Data Type	Supported External Types	Default External Type
CHAR	STRING	STRING
CHARACTER	OCISTRING	
LONG		
NCHAR		
NVARCHAR2		
ROWID		
VARCHAR		
VARCHAR2		
LONG RAW	RAW	RAW
RAW	OCIRAW	
BFILE	OCILOBLOCATOR	OCILOBLOCATOR
BLOB		
CLOB		
NCLOB		
NUMBER	OCINUMBER	OCINUMBER
DEC ¹		
DECIMAL ¹		
INT ¹		
INTEGER ¹		
NUMERIC ¹		
SMALLINT ¹		
DATE	OCIDATE	OCIDATE
TIMESTAMP	OCIDateTime	OCIDateTime
TIMESTAMP WITH TIME ZONE		
TIMESTAMP WITH LOCAL TIME ZONE		
INTERVAL DAY TO SECOND	OCIInterval	OCIInterval
INTERVAL YEAR TO MONTH		
composite object types: ADTs	dvoid	dvoid
composite object types: collections (associative arrays, varrays, nested tables)	OCICOLL	OCICOLL

¹ This PL/SQL type compiles only if you use AS EXTERNAL in your call spec.

18.9.2 External Data Type Mappings

Each external data type maps to a C data type, and the data type conversions are performed implicitly. To avoid errors when declaring C prototype parameters, see [Table 18–2](#), which shows the C data type to specify for a given external data type and PL/SQL parameter mode. For example, if the external data type of an OUT parameter is STRING, then specify the data type char * in your C prototype.

Table 18–2 External Data Type Mappings

External Data Type 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 *

Table 18–2 (Cont.) External Data Type Mappings

External Data Type Corresponding to PL/SQL Type	If Mode is IN or RETURN, Specify in C Prototype...	If Mode is IN by Reference or RETURN by Reference, Specify in C Prototype...	If Mode is IN OUT or OUT, Specify in C Prototype...
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
CHAR	char	char *	char *
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	OCIStruct *	OCIStruct *	OCIStruct *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *
OCIDATE	OCIDate *	OCIDate *	OCIDate *
OCICOLL	OCIColl * or OCIArrray * or OCITable *	OCIColl ** or OCIArrray ** or OCITable **	OCIColl ** or OCIArrray ** or OCITable **
OCITYPE	OCIType *	OCIType *	OCIType *
TDO	OCIType *	OCIType *	OCIType *
ADT (final types)	dvoid*	dvoid*	dvoid*

Table 18-2 (Cont.) External Data Type Mappings

External Data			
Type Corresponding to PL/SQL 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...
ADT (nonfinal types)	dvoid*	dvoid*	dvoid**

Composite data 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 data type, but must use the data types generated by Oracle Database's **Object Type Translator** (OTT). The optional TDO argument for `INDICATOR`, and for composite objects, in general, has the C data type, `OCIType` *.

`OCICOLL` for `REF` and collection arguments is optional and exists only for completeness. You cannot map a `REF` or collection type onto any other data type, or any other data type onto a `REF` or collection type.

18.9.3 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 data types that are, by default, passed by value. This is true for `IN`, and `RETURN` arguments of these 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.

18.9.4 Declaring Formal Parameters

Generally, the PL/SQL procedure that publishes an external procedure declares a list of formal parameters, as this example shows:

Note: You might need to set up this data structure for examples in this topic 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 data type (which maps to the default external data type). 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:

- Nondefault external data types
- 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.

18.9.5 Overriding Default Data Type Mapping

In some cases, you can use the `PARAMETERS` clause to override the default data type mappings. For example, you can remap the PL/SQL data type `BOOLEAN` from external data type `INT` to external data type `CHAR`.

18.9.6 Specifying Properties

You can also use the `PARAMETERS` clause to pass more information about PL/SQL formal parameters and function results to an external procedure. Do this by specifying one or more of these properties:

```
INDICATOR [{STRUCT | TDO}]
LENGTH
DURATION
MAXLEN
CHARSETID
CHARSETFORM
SELF
```

[Table 18–3](#) shows the allowed and the default external data types, PL/SQL data types, and PL/SQL parameter modes allowed for a given property. `MAXLEN` (used to specify data returned from C back to PL/SQL) cannot be applied to an `IN` parameter.

Table 18-3 Properties and Data Types

Property	Allowed External Types (C)	Default External Type (C)	Allowed PL/SQL Types	Allowed PL/SQL Modes	Default PL/SQL Passing Method
INDICATOR	SHORT	SHORT	all scalars	IN	BY VALUE
				IN OUT	BY REFERENCE
				OUT	BY REFERENCE
				RETURN	BY REFERENCE
LENGTH	[UNSIGNED] SHORT	INT	CHAR	IN	BY VALUE
	[UNSIGNED] INT		LONG RAW	IN OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	OUT	BY REFERENCE
			VARCHAR2	RETURN	BY REFERENCE
MAXLEN	[UNSIGNED] SHORT	INT	CHAR	IN OUT	BY REFERENCE
	[UNSIGNED] INT		LONG RAW	OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	RETURN	BY REFERENCE
			VARCHAR2		
CHARSETID	UNSIGNED SHORT	UNSIGNED INT	CHAR	IN	BY VALUE
CHARSETFORM	UNSIGNED INT		CLOB	IN OUT	BY REFERENCE
	UNSIGNED LONG		VARCHAR2	OUT	BY REFERENCE
				RETURN	BY REFERENCE

In this 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( int x, short x_ind, char *y, int *y_len, int *y maxlen,
short *retind );
```

The additional parameters in the C prototype correspond to the INDICATOR (for x), LENGTH (of y), and MAXLEN (of y), and 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](#)

18.9.6.1 INDICATOR

An `INDICATOR` is a parameter whose value indicates whether 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 determine if a parameter or function result is `NULL`. Also, an external procedure might need to signal the server that a returned value is `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 in [Section 18.9.6, "Specifying Properties."](#)

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,

18.9.6.2 LENGTH and MAXLEN

In PL/SQL, there is no standard way to indicate the length of a `RAW` or string parameter. However, you might 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.

`MAXLEN` does not apply to `IN` parameters. For `OUT`, `IN OUT`, and `RETURN` parameters, `MAXLEN` is passed by reference and is read-only.

18.9.6.3 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

18.9.6.4 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.

18.9.6.5 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 you want to create a Person object, consisting of a person's name and date of birth, and then create a table of this object type. You eventually want to determine the age of each Person object in this table.

1. 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
);
/
```

2. Declare the body of the member function 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;
/

```

(Typically, the member function is implemented in PL/SQL, but in this example it is an external procedure.)

The `calcAge_func` member function takes no arguments and 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.

3. Create and populate the matching table.

```

CREATE TABLE Person_tab OF Person1_typ;

INSERT INTO Person_tab
VALUES ('BOB', TO_DATE('14-MAY-85'));

INSERT INTO Person_tab
VALUES ('JOHN', TO_DATE('22-DEC-71'));

```

4. 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_
-----        -----
BOB           14-MAY-85    0
JOHN          22-DEC-71    0

```

This is sample C code that implements the external member function and the Object-Type-Translator (OTT)-generated struct definitions:

```

#include <oci.h>

struct PERSON
{
    OCIStruct *NAME;
    OCIDate B_DATE;
};

typedef struct PERSON PERSON;

struct PERSON_ind
{
    OCIIInd _atomic;
    OCIIInd NAME;
    OCIIInd 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;
OCIIInd         *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);
}

```

18.9.6.6 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);
```

18.9.6.7 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 this 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.

18.9.6.8 Interlanguage Parameter Mode Mappings

PL/SQL supports the IN, IN OUT, and OUT parameter modes, and the RETURN clause for procedures returning values.

18.10 Running 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 [Section 18.10.2, "CALL Statement Syntax."](#)

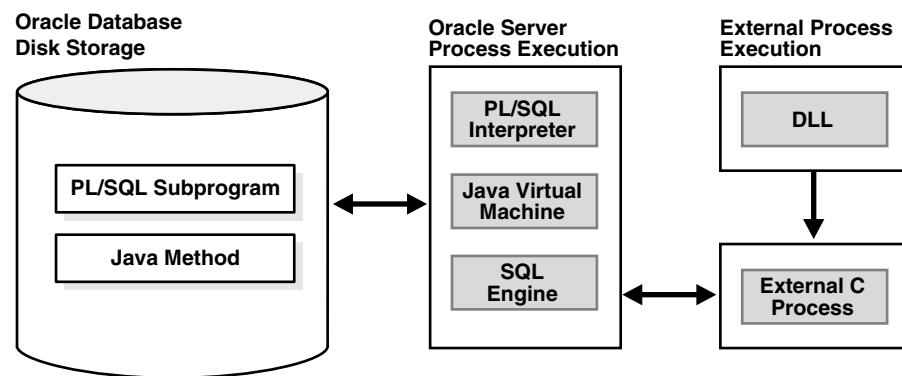
Such calls, which you code in the same manner as a call to a regular PL/SQL procedure, can appear in:

- Anonymous blocks

- Standalone and package procedures
- Methods of an object type
- Database triggers
- SQL statements (calls to package functions only).

Any PL/SQL block or procedure running 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 18–1](#) shows how Oracle Database and external procedures interact.

Figure 18–1 Oracle Database and External Procedures



Topics:

- [Preconditions for External Procedures](#)
- [CALL Statement Syntax](#)
- [Calling Java Class Methods](#)
- [Calling External C Procedures](#)

18.10.1 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](#)

18.10.1.1 Privileges of External Procedures

When external procedures are called through `CALL` specifications, they run with definer's privileges, rather than invoker privileges.

A program running with invoker privileges is not bound to a particular schema. It runs at the calling site and accesses database items (such as tables and views) with the caller's visibility and permissions. However, a program running with definer's privileges is bound to the schema in which it is defined. It runs at the defining site, in

the definer's schema, and accesses database items with the definer's visibility and permissions.

18.10.1.2 Managing Permissions

To call an external procedure, a user must have the EXECUTE privilege on its call specification. To grant this privilege, use the SQL statement GRANT (described in *Oracle Database SQL Language Reference*). For example, this statement allows the user `johndoe` to call the external procedure whose call specification is `plsToJ_demoExternal_proc`:

```
GRANT EXECUTE ON plsToJ_demoExternal_proc TO johndoe;
```

Grant the EXECUTE privilege on a call specification only to users who must call the procedure.

18.10.1.3 Creating Synonyms for External Procedures

For convenience, you or your DBA can create synonyms for external procedures using the CREATE PUBLIC SYNONYM statement. In this 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 johndoe.RecursiveFactorial;
```

18.10.2 CALL Statement Syntax

Call the external procedure through the SQL CALL statement. You can run 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 running 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 you 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 are 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.

18.10.3 Calling Java Class Methods

To call the `J_calcFactorial` class method published in [Section 18.6, "Publishing Java Class Methods"](#):

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
```

18.10.4 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, your DLL might not 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 stopped. 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, you call PL/SQL function `plsCallsCdivisor_func`, which you published in [Section 18.7, "Publishing External C Procedures"](#), 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;
BEGIN
  CALL plsCallsCdivisor_func(a, b);
  IF g IN (2,4,8) THEN ...
END;
```

18.11 Handling Errors and Exceptions in Multilanguage Programs

The PL/SQL compiler raises compile-time exceptions if an AS EXTERNAL call specification is found in a TYPE or PACKAGE specification.

C programs can raise exceptions through the OCIExtproc functions.

18.12 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](#)

18.12.1 OCIExtProcAllocCallMemory

The OCIExtProcAllocCallMemory 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:

```
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 this 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);
}

#ifndef 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;
    OCISqlStatement  *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 *) "password", (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));

/* ----- subprogram called here-----*/
printf ("calling concat...\n");
concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l);

return 0;
}

#endif

```

18.12.2 OCIExtProcRaiseExc

The OCIExtProcRaiseExc 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 OCIExtProcRaiseExc(
    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 this example shows, if the divisor is zero, `C_divide` uses `OCIExtProcRaiseExc` 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 Database error 1476. */
    if (OCIExtProcRaiseExc(ctx, (int)1476) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /* Incorrect parameters were passed. */
      assert(0);
    }
  }
  *result = (float)dividend / (float)divisor;
}

```

18.12.3 OCIExtProcRaiseExcWithMsg

The `OCIExtProcRaiseExcWithMsg` service routine raises a user-defined exception and returns a user-defined error message. The C prototype for this function follows:

```

int OCIExtProcRaiseExcWithMsg(
  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, you published external procedure `plsTo_divide_proc`. In this 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 Database 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;
}
```

18.13 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 C Procedures](#)
- [Example: Calling an External C Procedure](#)
- [Global Variables in External C Procedures](#)
- [Static Variables in External C Procedures](#)
- [Restrictions on External C Procedures](#)

18.13.1 OCIExtProcGetEnv

The `OCIExtProcGetEnv` 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 must 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 [Section 18.13.5](#).

Note: Callbacks are not necessarily a same-session phenomenon; you might run an SQL statement in a different session through OCIlogon.

An external C procedure running on Oracle Database can call a service routine to obtain OCI environment and service handles. With the OCI, you can use callbacks to run 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 this 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 need not include `oci.h`; instead, include `ociextp.h`.

18.13.2 Object Support for OCI Callbacks

To run object-related callbacks from your external procedures, the OCI environment in the `extproc` agent is fully initialized in object mode. You retrieve handles to this environment with the `OCIExtProcGetEnv` procedure.

The object runtime environment lets you use static and dynamic object support provided by OCI. To use 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 run callbacks, or call `OCIExtProc` service routines. After every external procedure call, the callback mechanism is cleaned up and all OCI handles are freed.

18.13.3 Restrictions on Callbacks

With callbacks, this SQL statements and OCI subprograms are not supported:

- Transaction control statements such as `COMMIT`
- Data definition statements such as `CREATE`
- These 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`
- These OCI subprograms:

```
OCIEnvInit
OCIInitialize
OCIPasswordChange
OCIServerAttach
OCIServerDetach
OCISessionBegin
OCISessionEnd
OCISvcCtxToLda
OCITransCommit
OCITransDetach
OCITransRollback
OCITransStart
```

Also, with OCI subprogram `OCCHandleAlloc`, these handle types are not supported:

```
OCI_HTYPE_SERVER
OCI_HTYPE_SESSION
OCI_HTYPE_SVCCTX
OCI_HTYPE_TRANS
```

18.13.4 Debugging External C 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 data type. For example, to pass an `OUT` parameter of type `REAL`, you must specify `float *`. Specifying `float`, `double *`, or any other C data type 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.

18.13.5 Example: Calling an External C 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` account, which must have `CREATE LIBRARY` privileges.

18.13.6 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. Therefore, in external procedures, all functions in a DLL share the value of the global variable. Global variables are also used to store data that is intended to persist beyond the lifetime of a function. However, Oracle discourages the use of global variables for two reasons:

- Threading

In the nonthreaded configuration of the agent process, one function is active at a time. For the multithreaded `extproc` agent, multiple functions can be active at the same time, and they might try to access the global variable concurrently, with unsuccessful results.

- DLL caching

Suppose that function `func1` tries to pass data to function `func2` by storing the data in a global variable. After `func1` completes, the DLL cache might be unloaded, causing all global variables to lose their values. Then, when `func2` runs, the DLL is reloaded, and all global variables are initialized to 0.

18.13.7 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 in [Section 18.13.6, "Global Variables in External C Procedures,"](#) 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 runtime 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 data types 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.

18.13.8 Restrictions on External C Procedures

These 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.
- External procedure callouts combined with distributed transactions is not 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.

Developing Applications with Oracle XA

Note: Avoid using XA if possible, for these reasons:

- XA can degrade performance.
- XA can cause in-doubt transactions.
- XA might be unable to take advantage of Oracle Database 12c Release 1 (12.1.0.1) features that enhance the ability of applications to continue after recoverable outages.

It might be possible to avoid using XA even when that seems avoidable (for example, if Oracle and non-Oracle resources must be used in the same transaction).

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 multiple databases.

Topics:

- [X/Open Distributed Transaction Processing \(DTP\)](#)
- [Oracle XA Library Subprograms](#)
- [Developing and Installing XA Applications](#)
- [Troubleshooting XA Applications](#)
- [Oracle XA Issues and Restrictions](#)

See Also:

- *Distributed TP: The XA Specification*, for an overview of XA, including basic architecture. Access at <https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=11144>.
- *Oracle Call Interface Programmer's Guide* for background and reference information about the Oracle XA library
- The Oracle Database platform-specific documentation for information about library linking filenames
- README for changes, bugs, and restrictions in the Oracle XA library for your platform

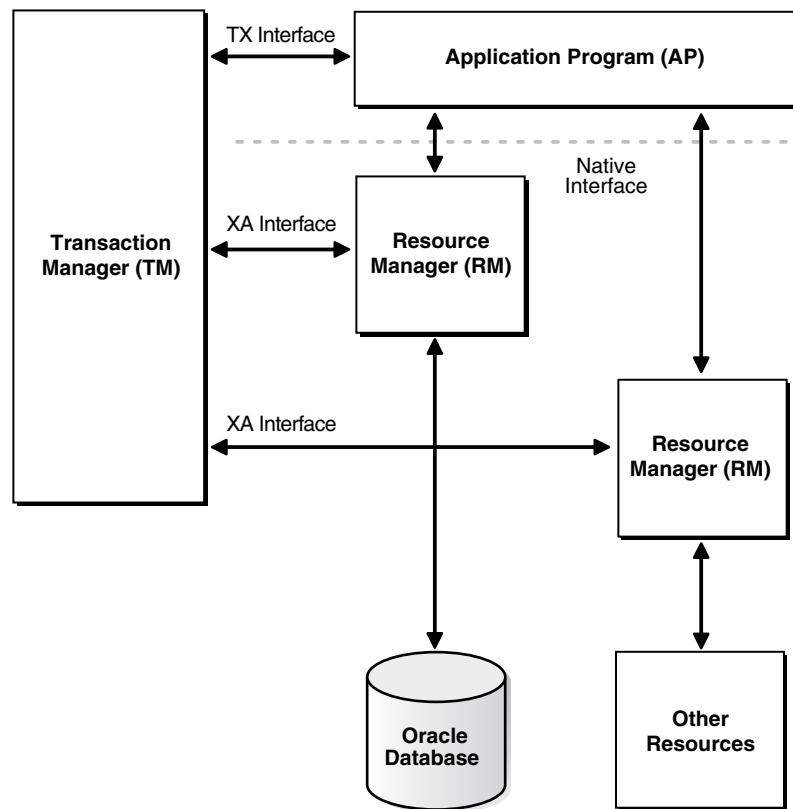
19.1 X/Open Distributed Transaction Processing (DTP)

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 19–1 illustrates a possible X/Open DTP model.

Figure 19–1 Possible DTP Model



Topics:

- [DTP Terminology](#)
- [Required Public Information](#)

19.1.1 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 comprise a global transaction. For 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), such as:

- Oracle Tuxedo
- IBM Transarc Encina
- IBM CICS

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 subprograms, which are described in ["Oracle XA Library Subprograms" on page 19-5](#), 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 Oracle Call Interface (OCI) program. The AP operates on the RM 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 subprograms 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 runs 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.

19.1.2 Required Public Information

As a resource manager, Oracle Database must publish the information described in [Table 19–1](#).

Table 19–1 Required XA Features Published by Oracle Database

XA Feature	Oracle Database Details
<code>xa_switch_t</code> structures	The Oracle Database <code>xa_switch_t</code> structure name is <code>xaosw</code> for static registration and <code>xaoswd</code> for dynamic registration. These structures contain entry points and other information for the resource manager.
<code>xa_switch_t</code> resource manager	The Oracle Database resource manager name within the <code>xa_switch_t</code> structure is <code>Oracle_XA</code> .
Close string	The close string used by <code>xa_close</code> is ignored and can be null.
Open string	For the description of the format of the open string that <code>xa_open</code> uses, see Section 19.3.3, "Defining the <code>xa_open</code> String."
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 <code>sqllib</code> , then link with <code>\$ORACLE_HOME/rdbms/lib/xaons1.o</code> or <code>\$ORACLE_HOME/rdbms/lib32/xaons1.o</code> (for 32 bit application on 64 bit platforms).
Requirements	None. The functionality to support XA is part of both Standard Edition and Enterprise Edition.

19.2 Oracle XA Library Subprograms

The Oracle XA library subprograms 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 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:

- Starting a transaction and associating it with an ID
- Rolling back a transaction
- Preparing and committing a transaction

Topics:

- [Oracle XA Library Subprograms](#)
- [Oracle XA Interface Extensions](#)

19.2.1 Oracle XA Library Subprograms

XA Library subprograms are described in [Table 19–2](#).

Table 19–2 XA Library Subprograms

XA Subprogram	Description
xa_open	Connects to the RM.
xa_close	Disconnects from the RM.
xa_start	Starts a 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.
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 need not worry about the subprograms in [Table 19–2](#) except to understand the role played by the `xa_open` string.

19.2.2 Oracle XA Interface Extensions

Oracle Database's XA interface includes some additional functions, which are described in [Table 19–3](#).

Table 19–3 Oracle XA Interface Extensions

Function	Description
<code>OCISvcCtx *xaoSvcCtx(text *dbname)</code>	Returns the OCI service handle for a given XA connection. The <code>dbname</code> parameter must be the same as the <code>DB</code> parameter passed in the <code>xa_open</code> string. OCI applications can use this routing instead of the <code>sql11d2</code> calls to obtain the connection handle. Hence, OCI applications need not link with the <code>sql11lib</code> library. The service handle can be converted to the Version 7 OCI logon data area (LDA) by using <code>OCISvcCtxToLda</code> [Version 8 OCI]. Client applications must remember to convert the Version 7 LDA to a service handle by using <code>OCILdaToSvcCtx</code> after completing the OCI calls.
<code>OCIEnv *xaoEnv(text *dbname)</code>	Returns the OCI environment handle for a given XA connection. The <code>dbname</code> parameter must be the same as the <code>DB</code> parameter passed in the <code>xa_open</code> string.
<code>int xaosterr(OCISvcCtx *SvcCtx, sb4 error)</code>	Converts an Oracle Database error code to an XA error code (applicable only to dynamic registration). The first parameter is the service handle used to run 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 <code>xa_start</code> failed. The function returns <code>XA_OK</code> if the error was not generated by the XA module or a valid XA error if the error was generated by the XA module.

19.3 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](#)

19.3.1 DBA or System Administrator Responsibilities

The responsibilities of the DBA or system administrator are:

1. Define the open string, with help from the application developer. For details, see [Section 19.3.3, "Defining the `xa_open` String."](#)
2. Ensure that the static data dictionary view `DBA_PENDING_TRANSACTIONS` exists and grant the `READ` or `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 must have. Ensure 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 is to write the XA trace file.

See Also: [Section 19.3.3, "Defining the `xa_open` String,"](#) for information about 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.

19.3.2 Application Developer Responsibilities

The responsibilities of the application developer are:

1. Define the open string with help from the DBA or system administrator, as explained in [Section 19.3.3, "Defining the `xa_open` String."](#).
2. Develop the applications.

Observe special restrictions on transaction-oriented SQL statements for precompilers.

See Also: [Section 19.3, "Developing and Installing XA Applications"](#)

3. Link the application according to TPM vendor instructions.

19.3.3 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](#)

19.3.3.1 Syntax of the `xa_open` String

You can define an open string with the syntax shown in [Example 19–1](#).

Example 19–1 `xa_open` String

```
ORACLE_XA{+required_fields...} [+optional_fields...]
```

These strings shows sample parameter settings:

```
ORACLE_XA+DB=MANAGERS+SqlNet=SID1+ACC=P/username/password
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+DB=PAYROLL+SqlNet=SID2+ACC=P/username/password
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+SqlNet=SID3+ACC=P/username/password
  +SesTM=10+LogDir=/usr/local/xalog
```

These topics describe valid parameters for the `required_fields` and `optional_fields` placeholders:

- [Required Fields for the `xa_open` String](#)
- [Optional Fields for the `xa_open` String](#)

Note:

- You can enter the required fields and optional fields in any order when constructing the open string.
- All field names are case insensitive. Whether their values are case-sensitive depends on the platform.
- There is no way to use the plus character (+) as part of the actual information string.

19.3.3.2 Required Fields for the `xa_open` String

The `required_fields` placeholder in [Example 19–1](#) refers to any of the name-value pairs described in [Table 19–4](#).

Table 19–4 Required Fields of `xa_open` string

Syntax Element	Description
<code>Acc=P//</code>	Specifies that no explicit user or password information is provided and that the operating system authentication form is used. For more information see <i>Oracle Database Administrator's Guide</i> .
<code>Acc=P/user/password</code>	Specifies the user name and password for a valid Oracle Database account. As described in Section 19.3.1, "DBA or System Administrator Responsibilities," ensure that HR has the READ or SELECT privilege on the <code>DBA_PENDING_TRANSACTIONS</code> table.
<code>SesTm=session_time_limit</code>	<p>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 terminates the transaction. For example, <code>SesTM=15</code> indicates that the session idle time limit is 15 seconds.</p> <p>For example, if the TPM uses remote subprogram calls between the client and the servers, then <code>SesTM</code> applies to the time between the completion of one RPC and the initiation of the next RPC, or the <code>tx_commit</code>, or the <code>tx_rollback</code>.</p> <p>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 <code>SesTM=0</code>, then the <code>SesTM</code> setting is not effective after the parent process is terminated.</p>

19.3.3.3 Optional Fields for the `xa_open` String

The `optional_fields` placeholder in [Example 19–1](#) refers to any of the name-value pairs described in [Table 19–5](#).

Table 19–5 Optional Fields in the `xa_open` String

Syntax Element	Description
<code>NoLocal= true false</code>	Specifies whether local transactions are allowed. The default value is <code>false</code> . If the application must disallow local transactions, then set the value to <code>true</code> .

Table 19–5 (Cont.) Optional Fields in the *xa_open* String

Syntax Element	Description
<code>DB=db_name</code>	<p>Specifies the name used by Oracle Database precompilers to identify the database. For example, <code>DB=payroll</code> specifies that the database name is <code>payroll</code> and that the application server program uses that name in <code>AT</code> clauses.</p> <p>Application programs that use only the default database for the Oracle Database precompiler (that is, they do not use the <code>AT</code> clause in their SQL statements) must omit the <code>DB=db_name</code> clause in the open string. Applications that use explicitly named databases must indicate that database name in their <code>DB=db_name</code> field. Oracle Database Version 7 OCI programs must call the <code>sql11d2</code> function to obtain the correct context for logon data area (<code>Lda_Def</code>), which is the equivalent of an OCI service context. Version 8 and higher OCI programs must call the <code>xaoSvcCtx</code> function to get the <code>OCISvcCtx</code> service context.</p> <p>The <code>db_name</code> 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 run SQL statements. The SID is set from either the environment variable <code>ORACLE_SID</code> 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 <code>db_name</code>.</p>
<code>LogDir=log_dir</code>	<p>Specifies the path name on the local system where the Oracle XA library error and tracing information is to be logged. The default is <code>\$ORACLE_HOME/rdbms/log</code> if <code>ORACLE_HOME</code> is set; otherwise, it specifies the current directory. For example, <code>LogDir=/xa_trace</code> indicates that the logging information is located under the <code>/xa_trace</code> directory. Ensure that the directory exists and the application server can write to it.</p>
<code>Objects= true false</code>	<p>Specifies whether the application is initialized in object mode. The default value is <code>false</code>. If the application must use certain API calls that require object mode, such as <code>OCIRawAssignBytes</code>, then set the value to <code>true</code>.</p>
<code>MaxCur=maximum_#_of_open_cursors</code>	<p>Specifies the number of cursors to be allocated when the database is opened. It serves the same purpose as the precompiler option <code>maxopencursors</code>. For example, <code>MaxCur=5</code> indicates that the precompiler tries to keep five open cursors cached. This parameter overrides the precompiler option <code>maxopencursors</code> that you might have specified in your source code or at compile time.</p>
<code>SqlNet=db_link</code>	<p>Specifies the Oracle Net database link to use to log on to the system. This string must be an entry in <code>tnsnames.ora</code>. For example, the string <code>SqlNet=inst1_disp</code> might connect to a shared server at instance 1 if so defined in <code>tnsnames.ora</code>.</p> <p>You can use the <code>SqlNet</code> parameter to specify the <code>ORACLE_SID</code> in cases where you cannot control the server environment variable. You must also use it when the server must access multiple Oracle Database instances. To use the Oracle Net string without actually accessing a remote database, use the Pipe driver. For example, specify <code>SqlNet=localsid1</code>, where <code>localsid1</code> is an alias defined in the <code>tnsnames.ora</code> file.</p>

Table 19-5 (Cont.) Optional Fields in the `xa_open` String

Syntax Element	Description
<code>Loose_Coupling=true false</code>	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 <code>true</code> for loosely coupled branches. If branches are tightly coupled, then they share locks. Set the value to <code>false</code> for tightly coupled branches. The default value is <code>false</code> . Note: When running Oracle RAC, if transaction branches land on different Oracle RAC instances, then they are loosely coupled even if <code>Loose_Coupling=false</code> .
<code>SesWt=session_wait_limit</code>	Specifies the number of seconds Oracle Database waits for a transaction branch that is being used by another session before <code>XA_RETRY</code> is returned. The default value is 60 seconds.
<code>Threads=true false</code>	Specifies whether the application is multithreaded. The default value is <code>false</code> . If the application is multithreaded, then the setting is <code>true</code> .
<code>FAN=true false</code>	Specifies whether the application will use Fast Application Notification (FAN). The default value is <code>false</code> . If the application will use FAN, then the setting is <code>true</code> . For information about FAN, see <i>Oracle Database Administrator's Guide</i> .

19.3.4 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 these options when interfacing with precompilers:

- [Using Precompilers with the Default Database](#)
- [Using Precompilers with a Named Database](#)

The examples in this topic use the precompiler Pro*C/C++.

19.3.4.1 Using Precompilers with the Default Database

To interface to a precompiler with the default database, ensure 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.

This is an example of an open string identifying a default Pro*C/C++ connection:

```
ORACLE_XA+SqlNet=maildb+ACC=P/username/password
+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;
```

19.3.4.2 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 and 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 open strings in the transaction manager as shown in [Example 19–2](#).

Example 19–2 Sample Open String Configuration

```
ORACLE_XA+DB=MANAGERS+SqlNet=SID1+ACC=P/username/password
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+DB=PAYOUT+SqlNet=SID2+ACC=P/username/password
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+SqlNet=SID3+ACC=P/username/password
  +SesTM=10+LogDir=/usr/local/xalog
```

There is no `DB=db_name` field in the last open string in [Example 19–2](#).

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 these:

```
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 this 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.

19.3.5 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 run the function `xaoSvcCtx` to obtain the service context structure when they must access the resource manager.

In applications that must 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*

19.3.6 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 19–6](#), but not the XA Library Subprograms listed in [Table 19–2](#).

The TMs typically control the transactions through the XA interface. This interface includes the functions described in [Table 19–2](#).

Table 19–6 TX Interface Functions

TX Function	Description
<code>tx_open</code>	Logs into the resource manager(s)
<code>tx_close</code>	Logs out of the resource manager(s)
<code>tx_begin</code>	Starts a transaction
<code>tx_commit</code>	Commits a transaction
<code>tx_rollback</code>	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 19-14 use such a client/server model. A service is a logical unit of work that, for 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 runs SQL statements to update information in certain tables in the database. Also, 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 19-14, you can encode transaction control statements within either the client or the server.

To have multiple processes 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, these examples 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.

19.3.7 Examples of Precompiler Applications

These examples illustrate precompiler applications. Assume that the application server has logged onto the RMs system, in a TPM-specific manner. [Example 19-3](#) shows a transaction started by an application server.

Example 19-3 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 19-4](#) shows a transaction started by an application client.

Example 19-4 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>
}
```

19.3.8 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:

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, ensure 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 true only 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 run only once.

Table 19-7 lists the TPM functions that replace regular Oracle Database statements when migrating precompiler or OCI applications to TPM applications.

Table 19-7 TPM Replacement Statements

Regular Oracle Database Statements	TPM Functions
<code>CONNECT</code> <i>user/password</i>	<code>tx_open</code> (possibly implicit)
implicit start of transaction	<code>tx_begin</code>
SQL	Service that runs the SQL
<code>COMMIT</code>	<code>tx_commit</code>
<code>ROLLBACK</code>	<code>tx_rollback</code>
<code>disconnect</code>	<code>tx_close</code> (possibly implicit)

19.3.9 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 thread and deallocated 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](#)

19.3.9.1 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 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.

19.3.9.2 Restrictions on Threading in Oracle XA

These restrictions apply when using threads:

- Any Pro* or OCI code that runs as part of the application server process on the transaction monitor cannot be threaded unless the transaction monitor is explicitly told when each 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.

19.3.10 Using the DBMS_XA Package

PL/SQL applications can use the Oracle XA library with the `DBMS_XA` package. For information about this package, see *Oracle Database PL/SQL Packages and Types Reference*.

In [Example 19-5](#), 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. All three sessions are connected to the `HR` schema.

Example 19-5 Using the DBMS_XA Package

```
REM Session 1 starts a transaction and does some work.
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.
  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.

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

```

19.4 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](#)

19.4.1 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 these contents:

```
1032.12345.2: ORA-01017: invalid username/password; logon denied
1032.12345.2: xaolgn: XAER_INVAL; logon denied
```

Table 19–8 explains the meaning of each element.

Table 19–8 Sample Trace File Contents

String	Description
1032	The time when the information is logged.
12345	The process ID (PID).
2	Resource manager ID
xaolgn	Name of module
XAER_INVAL	Error returned as specified in the XA standard

Table 19–8 (Cont.) Sample Trace File Contents

String	Description
ORA-01017	Oracle Database information that was returned

Topics:

- [xa_open String DbgFl](#)
- [Trace File Locations](#)

19.4.1.1 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 these 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 programs. This is generally useful 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 useful 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.

19.4.1.2 Trace File Locations

The XA application determines a location for the trace file according to this 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 this 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.

19.4.2 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 these 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.

19.4.3 Using SYS Account Tables to Monitor Oracle XA Transactions

These 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, this column information applies specifically to the DBA_2PC_NEIGHBORS table:

- The DBID column is always `xa_orcl`
- The DBUSER_OWNER column is always `db_namexa.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, this 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.

19.5 Oracle XA Issues and Restrictions

Topics:

- [Using Database Links in Oracle XA Applications](#)
- [Managing Transaction Branches in Oracle XA Applications](#)
- [Using Oracle XA with Oracle Real Application Clusters \(Oracle RAC\)](#)
- [SQL-Based Oracle XA Restrictions](#)
- [Miscellaneous Restrictions](#)

19.5.1 Using Database Links in Oracle XA Applications

Oracle XA applications can access other Oracle Database instances through database links with these 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.

If 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 can use the database link connection if the user who created the connection also 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.

19.5.2 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 running any statement. Because the system does not obtain a lock before running 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 19–9 summarizes the trade-offs between tightly coupled branches and loosely coupled branches.

Table 19–9 Tightly and Loosely Coupled Transaction Branches

Attribute	Tightly Coupled Branches	Loosely Coupled Branches
Two Phase Commit	Read-only optimization [prepare for all branches, commit for last branch]	Two phases [prepare and commit for all branches]
Serialization	Database call	None

19.5.3 Using Oracle XA with Oracle Real Application Clusters (Oracle RAC)

As of Oracle Database 11g Release 1 (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.

Note: External procedure callouts combined with distributed transactions is not supported.

Topics:

- [GLOBAL_TXN_PROCESSES Initialization Parameter](#)
- [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](#)

19.5.3.1 GLOBAL_TXN_PROCESSES Initialization Parameter

The initialization parameter `GLOBAL_TXN_PROCESSES` specifies the initial number of `GTxN` background processes for each Oracle RAC instance. Its default value is 1.

Leave this parameter at its default value clusterwide if distributed transactions might span multiple Oracle RAC instances. 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`

19.5.3.2 Managing Transaction Branches on Oracle RAC

Note: This topic applies if either of the following is true:

- The initialization parameter `GLOBAL_TXN_PROCESSES` is not at its default value in the initialization file of every Oracle RAC instance.
 - The Oracle XA application resumes or joins previously detached branches of a transaction.
-

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 Oracle Database 11g Release 1 (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 Oracle Database 11g Release 1 (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—regardless of whether 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 Oracle Database 11g Release 1 (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 Oracle RAC instances remain tightly coupled; that is, they share locks and resources across Oracle RAC instances.

For example, when you use a DTP service, this 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

19.5.3.3 Managing Instance Recovery in Oracle RAC with DTP Services (10.2)

Before 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 Oracle Database 10g Release 2 (10.2), there is no such requirement to call `xa_recover` in cases where the TM has enough information about in-flight transactions.

Note: As of Oracle Database 9g Release 2 (9.2), `xa_recover` is required to wait for distributed data manipulation language (DML) statements to complete on remote sites.

Using DTP services in Oracle RAC has these 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
- *Oracle Real Application Clusters Administration and Deployment Guide* for information about services and distributed transaction processing in Oracle RAC

19.5.3.4 Global Uniqueness of XIDs in Oracle RAC

Before Oracle Database 11g Release 1 (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 run SQL even though the XID is not unique across Oracle RAC instances.

As of Oracle Database 11g Release 1 (11.1), Oracle RAC database detects the duplicate XIDs across Oracle RAC instances and prevents a branch with a duplicate XID from starting.

See Also: *Oracle Real Application Clusters Administration and Deployment Guide* for information about services and distributed transaction processing in Oracle RAC

19.5.3.5 Tight and Loose Coupling

Oracle Database transaction branches within the same global transaction can be coupled either tightly or loosely (for details, see [Section 19.5.2, "Managing Transaction Branches in Oracle XA Applications"](#)). Ordinarily, coupling type is determined by the value of the `Loose_Coupling` field of the `xa_open` string (see [Table 19-5](#)). However, if

transaction branches land on different Oracle RAC instances when running Oracle RAC, they are loosely coupled even if `Loose_Coupling=false`.

See Also: *Oracle Real Application Clusters Administration and Deployment Guide* for information about services and distributed transaction processing in Oracle RAC

19.5.4 SQL-Based Oracle XA Restrictions

This section describes restrictions concerning these SQL operations:

- [Rollbacks and Commits](#)
- [DDL Statements](#)
- [Session State](#)
- [EXEC SQL](#)

19.5.4.1 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 run `OCITransRollback`, or the Version 7 equivalent `oro1`. 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 run `OCITransCommit` or the Version 7 equivalent `ocom`. For example, use `tx_commit` or `tx_rollback` to end a global transaction.

19.5.4.2 DDL Statements

Because a data definition language (DDL) statement, such as `CREATE TABLE`, implies an implicit commit, the Oracle XA application cannot run any DDL statements.

19.5.4.3 Session State

Oracle Database does not guarantee that session state is 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 runs 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.

19.5.4.4 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`.

19.5.5 Miscellaneous Restrictions

- You cannot use both Oracle XA and a gateway in the same session.

- 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: [Section 19.5.1, "Using Database Links in Oracle XA Applications"](#)

- 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`).

20

Developing Applications with the Publish-Subscribe Model

This chapter explains how to develop applications on the publish-subscribe model.

Topics:

- [Introduction to the Publish-Subscribe Model](#)
- [Publish-Subscribe Architecture](#)
- [Publish-Subscribe Concepts](#)
- [Examples of a Publish-Subscribe Mechanism](#)

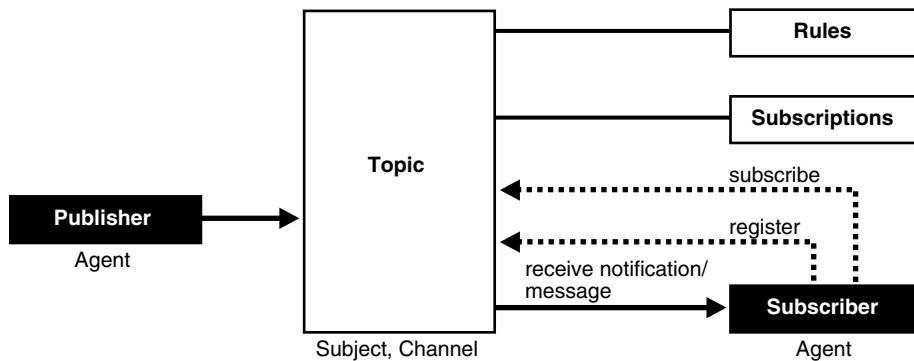
20.1 Introduction to the Publish-Subscribe Model

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 20-1](#) illustrates publish and subscribe functionality.

Figure 20–1 Oracle Publish-Subscribe Functionality

A subscriber subscribes to a queue by expressing interest in messages enqueued to that queue and by using a subject- or content-based rule as a filter. This results in a set of rule-based subscriptions associated with a given queue.

At runtime, publishers post messages to various queues. The queue (in other words, the delivery mechanisms of the underlying infrastructure) then delivers messages that match the various subscriptions to the appropriate subscribers.

20.2 Publish-Subscribe Architecture

Oracle Database includes these features to support database-enabled publish-subscribe messaging:

- [Database Events](#)
- [Oracle Advanced Queuing](#)
- [Client Notification](#)

20.2.1 Database Events

Database events support declarative definitions for publishing database events, detection, and runtime 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](#)

20.2.2 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 Database Advanced Queuing User's Guide](#)

20.2.3 Client Notification

Client notifications support asynchronous delivery of messages to interested subscribers, enabling 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*

20.3 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 might be unstructured (RAW) or it might have a well-defined structure (ADT). This allows both subject- or content-based subscriptions.

subscriber

Subscribers (agents) can 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 predefined 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 might 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 must notify all interested clients, it posts the message to all registered clients.

receiving a message

A subscriber can receive messages through any of these 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 can 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 other appropriate manner. While the messages are deferred, there is no asynchronous delivery to the end-client.

20.4 Examples of a Publish-Subscribe Mechanism

This example shows how database events, client notification, and AQ work 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. To use AQ functionality, user pubsub needs AQ_ADMINISTRATOR_ROLE privileges and EXECUTE privilege on DBMS_AQ and DBMS_AQADM.

```

Rem -----
REM create queue table for persistent multiple consumers:
Rem -----


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
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 = ''HR''' );
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). This 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 'HR' 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 HR 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  
*****/
```

```

OCI.SessionEnd ( svchp,  errhp, authp, (ub4) OCI_DEFAULT);

/* detach from server */
OCI.ServerDetach( 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 HR logs on to the database, the client is notified, and the call back function notifySnoop is invoked.

21

Using the Oracle ODBC Driver

This chapter contains the following sections:

Topics:

- [About Oracle ODBC Driver](#)
- [For All Users](#)
- [For Advanced Users](#)
- [For Programmers](#)

21.1 About Oracle ODBC Driver

What is ODBC?

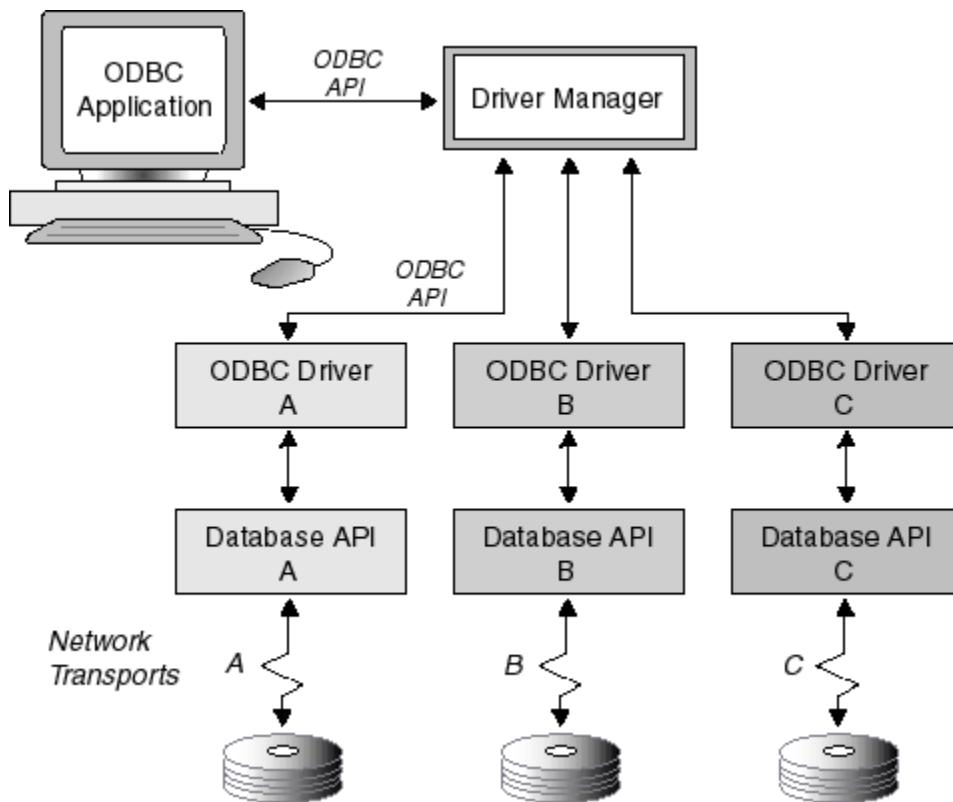
Open Database Connectivity (ODBC) provides a standard interface that allows one application to access many different data sources. The application's source code does not have to be recompiled for each data source. A database driver links the application to a specific data source. A database driver is a dynamic-link library that an application can invoke on demand to gain access to a particular data source. Therefore, the application can access any data source for which a database driver exists.

The ODBC interface defines the following:

- A library of ODBC function calls that allows an application to connect to a data source, execute structured query language (SQL) statements, and retrieve results.
- SQL syntax based on the SQL-99 specification.
- A standard set of error codes.
- A standard way to connect to and log in to a data source.
- A standard representation for data types.

[Figure 21-1](#) shows the components of the ODBC model. The model begins with an ODBC application making a call to the Driver Manager through the ODBC application program interface (API). The Driver Manager can be either the Microsoft Driver Manager or the unixODBC Driver Manager. Still using the ODBC API, the Driver Manager makes a call to the ODBC Driver. The ODBC Driver accesses the database over a network communications link using the database API. [Figure 21-1](#) shows an ODBC application accessing three separate databases.

Figure 21–1 Components of the ODBC Model



Related Topic

[What is the Oracle ODBC Driver](#)

21.2 For All Users

Topics:

- [Oracle ODBC Driver](#)
- [Configuration Tasks](#)
- [Modifying the oraodbc.ini File](#)
- [Connecting to a Data Source](#)
- [Troubleshooting](#)

21.2.1 Oracle ODBC Driver

Topics:

- [What is the Oracle ODBC Driver](#)
- [New and Changed Features](#)
- [Features Not Supported](#)
- [Files Created by the Installation](#)
- [Driver Conformance Levels](#)

- [Known Limitations](#)

21.2.1.1 What is the Oracle ODBC Driver

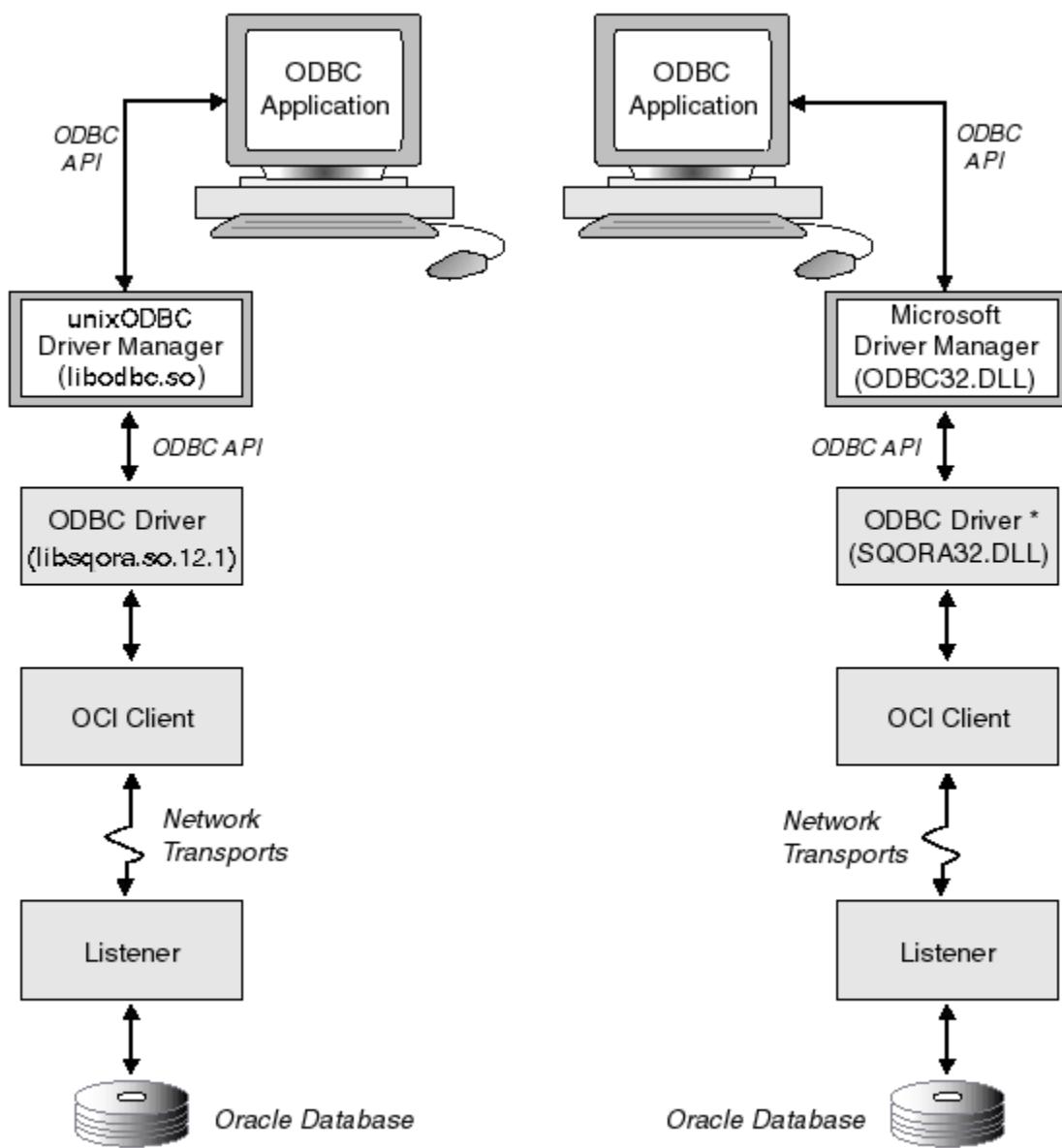
The Oracle ODBC Driver enables ODBC applications on Microsoft Windows, as well as UNIX platforms like Linux, Solaris, and IBM Advanced Interactive eXecutive (AIX) read and write access to Oracle® databases through the ODBC interface using Oracle Net Services software.

The Oracle ODBC Driver uses the Oracle Call Interface (OCI) client and server software to submit requests to and receive responses from the data source. Oracle Net Services communications protocol is used for communications between the OCI client and the Oracle server.

The Oracle ODBC Driver translates ODBC SQL syntax into syntax that can be used to access the data source. When the results are returned from the data source, the Oracle ODBC Driver translates them back to ODBC SQL syntax.

[Figure 21–2](#) shows the Oracle ODBC Driver architecture as described in the preceding paragraphs.

Figure 21–2 Oracle ODBC Driver Architecture



* The Oracle ODBC Resource data definition language (DLL) file (sqresxx.dll), where xx represents the language abbreviation, contains all pertinent language information; the default resource file used is sqresus.dll.

For more information about the OCI client and server software, refer to the OCI documentation.

Related Topics

- [Configuring the Data Source](#)
- [Connecting to a Data Source](#)
- [Driver Conformance Levels](#)
- [New and Changed Features](#)
- [Files Created by the Installation](#)

21.2.1.2 New and Changed Features

Topics:

- [New Features for Oracle ODBC Driver Release 12.1.0.1.0](#)
- [New Features for Oracle ODBC Driver Release 11.2.0.1.0](#)
- [New Features for Oracle ODBC Driver Release 11.1.0.1.0](#)
- [New Features for Oracle ODBC Driver Release 10.1.0.2.0](#)
- [Changes for Oracle ODBC Driver Release 10.1.0.2.0](#)

New Features for Oracle ODBC Driver Release 12.1.0.1.0

Features of the Oracle ODBC Driver Release 12.1.0.1.0 software for the Microsoft Windows Server 2008, Windows Server 2008 R2, Windows 7, Windows 8, Windows Server 2012, Linux X86-64 (32/64 bit), Sun Solaris SPARC64 (32,64 bit), IBM AIX 5L (32,64 bit), Sun Solaris X64 (32,64 bit), HPUX IA64 (32,64 bit), ZLinux (32,64 bit) operating systems are described as follows:

- Oracle ODBC Driver now supports 32 KB data columns with VARCHAR2, NVARCHAR2 and RAW data. See *Oracle Database PL/SQL Language Reference* and *Oracle Database SQL Language Reference* for information about creating 32 KB columns.
- New parameters in the odbc.ini file or connection level attributes:
 - `SQL_TRANSLATE_ERRORS = {T|F}` [Default is F (false)]

Any migrated third party ODBC application, which is using the SQL Translation Framework feature, expects that errors returned by the server to be in their native database format, then users can register their translation of errors with the SQL Translation Profile in Oracle Database running in SQL Translation Framework mode. After error translation is registered, then ODBC application users can enable this option, `SQLTranslateErrors = T`, to receive native errors according to their registration.

For more information on SQL Translation Framework, see *Oracle Database Migration Guide*, in particular information about SQL Translation Framework Architecture and Overview, Translation Framework installation and configuration, and migration examples.

See [Table 21-5](#) for more information.

- Oracle ODBC driver now supports executing a stored procedure, which can return implicit results without using RefCursor. This support eases any third party ODBC application, which migrated to Oracle and wants to use this same functionality that was provided by their previous vendors.

See *Oracle Database Migration Guide* for more information about implicit results support by Oracle Database.

- Extended support of `SQLColAttribute()` field identifiers to support Oracle Database auto increment feature. You can use this feature by including Oracle ODBC driver specific header file `sqora.h` in the application. See *Oracle Call Interface Programmer's Guide* for more information about auto increment:
 - `SQL_COLUMN_AUTO_INCREMENT`

Starting from Oracle Database 12c Release 1 (12.1.0.1), Oracle supports auto increment columns so the Oracle ODBC Driver has extended the same support through the existing `SQLColAttribute()` identifier `SQL_COLUMN_AUTO_INCREMENT`.

INCREMENT. This property is read-only and returns SQL_TRUE if the column is auto increment; otherwise, it returns SQL_FALSE.

- SQL_ORCLATTR_COLUMN_PROP

Starting from Oracle Database 12c Release 1 (12.1.0.1), Oracle ODBC Driver supports a new driver specific field identifier SQL_ORCLATTR_COLUMN_PROP, which returns the attributes of the column. This identifier returns SQLULEN value, which has all the column properties, shown as follows:

32	...	10	9	8	3	2	1	

| | |
 | | | -> Column is auto-increment?
 | | | -> Auto value is always generated?
 | -> If generated by default when null?

- ODBC APIs supported in Oracle Database 12c Release 1 (12.1.0.1)
 - SQLMoreResults()

Implements ODBC support for implicit results. See [Table 21-7](#) in [Section 21.4.11.2, "Implementation of ODBC API Functions,"](#) and [http://msdn.microsoft.com/en-us/library/ms714673\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms714673(v=VS.85).aspx) for more information.

New Features for Oracle ODBC Driver Release 11.2.0.1.0

Features of the Oracle ODBC Driver Release 11.2.0.1.0 software for the Microsoft Windows XP, Microsoft Windows 2003 Server, Microsoft Windows Vista, Linux X86-32 (RHEL AS 4,5), Linux X86-64 (RHEL AS 4,5) (32/64 bit), Sun Solaris SPARC64 (9,10) (32,64 bit), IBM AIX 5L 5.2 (32,64 bit), Linux IA64 (64 bit), Linux on Power (32,64 bit), Sun Solaris X64 (64 bit), Hewlett Packard Itanium (32,64 bit) operating systems are described as follows:

- Prefetching of LONG and LONG RAW data

Oracle ODBC driver is enhanced to prefetch LONG or LONG RAW data to improve performance of ODBC applications. To do this, the maximum size of LONG data (MaxLargeData) must be set in the registry on Windows (you also must add the registry key MaxLargeData in the data source name (DSN)), and set this manually in the odbc.ini file on UNIX platforms. This enhancement improves the performance of Oracle ODBC driver up to 10 times, depending on the MaxLargeData size set by the user. The default value of MaxLargeData is 0. The maximum value for MaxLargeData that you can set is 64 KB (65536 bytes).

If the value of MaxLargeData is greater than 65536, the data fetched is only 65536 bytes. If your database has LONG or LONG RAW data that is greater than 65536 bytes, then set MaxLargeData to 0 (the default value), which causes single-row fetch and fetches complete LONG data. If you pass a buffer size less than the MaxLargeData size in nonpolling mode, a data truncation error occurs if the LONG data size in the database is greater than the buffer size.

- Option for using OCIDescribeAny() for fetching metadata

When an application makes heavy calls to small packaged procedures that return REF CURSORS, a performance improvement can be made by forcing the driver to use OCIDescribeAny(). To enable this option, set the value of UseOCIDescribeAny in odbc.ini to T (True), default value is F (False), on UNIX platforms, and through the registry on Windows.

New Features for Oracle ODBC Driver Release 11.1.0.1.0

Features of the Oracle ODBC Driver Release 11.1.0.1.0 software for the Windows XP, Linux, Solaris, and IBM AIX operating systems are described as follows:

- **Disable Rule Hint (DRH Connect String)**

Added the new connection option, Disable RULE Hint that allows user to specify the option to select whether to use RULE Hint in catalog APIs. The change has been done to increase the performance of ODBC driver for catalog APIs. The default value for the option is TRUE, which means that RULE Hint is not used in catalog APIs by default.

- **Bind Number As Float (BNF Connect String)**

Added the new connection option, Bind Number As Float. By introducing Column Binding for NUMBER Column as FLOAT when column contains float data speeds up the query execution that uses bind variables as FLOAT.

- **Statement Caching**

Added support for OCI statement caching feature that provides and manages a cache of statements for each session. By implementing the support for OCI Statement Caching option, Oracle ODBC Driver performance improves when users parse the same statement multiple times in the same connection. The default value for the statement cache flag is FALSE.

New Features for Oracle ODBC Driver Release 10.1.0.2.0

Features of the Oracle ODBC Driver Release 10.1.0.2.0 software for the Windows 98, Windows 2000, Windows XP, and Windows NT X86 operating systems are described as follows:

- **Bind TIMESTAMP as DATE (BTD Connect String)**

Added the new connection option, Bind TIMESTAMP as DATE, that allows you to bind the ODBC driver SQL_TIMESTAMP data type to the Oracle DATE data type instead of to the Oracle TIMESTAMP data type (which is the default).

- **MONTHNAME (exp) Function**

Added support for the MONTHNAME (exp) function which returns the name of the month represented by the date expression. For example, 'April'.

- **DAYNAME (exp) Function**

Added support for the DAYNAME (exp) function which returns the name of the day represented by the date expression. For example, 'Tuesday'.

- **Instant Client Configuration**

Added support for the Instant Client mode configuration.

Changes for Oracle ODBC Driver Release 10.1.0.2.0

Changed or deprecated features of the Oracle ODBC Driver Release 10.1.0.2.0 include:

- **Disable Microsoft Transaction Server**

Changed the default setting for the Disable Microsoft Transaction Server (MTS) from FALSE to TRUE. By default, MTS support is disabled.

- **Floating Point Data Types**

Changed the mapping of the Oracle data types, BINARY_FLOAT and BINARY_DOUBLE, to map to the ODBC data types, SQL_REAL and SQL_DOUBLE, respectively.

- SQLGetData Extensions (GDE Connect String)
Deprecated the SQLGetData Extensions connection in this release. The functionality of this feature is always enabled.
- Force Retrieval of Longs (FRL Connect String)
Deprecated the Force Retrieval of Longs connection option in this release. The functionality of this feature is always enabled.
- Translation Options Configuration Tab
Deprecated the Translation Options tab previously found on the Oracle ODBC Driver Configuration dialog box in this release.
- Release Notes
Renamed the Release Notes file from `ODBCRelnotes.wri` to `ODBCRelnotesUS.htm`.

21.2.1.3 Features Not Supported

The Oracle ODBC Driver does not support the following ODBC 3.0 features:

- Interval data types
- `SQL_C_UBIGINT` and `SQL_C_SBIGINT` C data type identifiers
- Shared connections
- Shared environments
- The `SQL_LOGIN_TIMEOUT` attribute of `SQLSetConnectAttr`

[Table 21-1](#) describes the SQL functions that the Oracle ODBC Driver does not support.

Table 21-1 SQL Functions Not Supported by the Oracle ODBC Driver

String Functions	Numeric Functions	Time, Date, and Interval Functions
<code>BIT_LENGTH</code>	<code>ACOS</code>	<code>CURRENT_DATE</code>
<code>CHAR_LENGTH</code>	<code>ASIN</code>	<code>CURRENT_TIME</code>
<code>CHARACTER_LENGTH</code>	<code>ATAN</code>	<code>CURRENT_TIMESTAMP</code>
<code>DIFFERENCE</code>	<code>ATAN2</code>	<code>EXTRACT</code>
<code>OCTET_LENGTH</code>	<code>COT</code>	<code>TIMESTAMPDIFF</code>
<code>POSITION</code>	<code>DEGREES</code>	
	<code>RADIANS</code>	
	<code>RAND</code>	
	<code>ROUND</code>	

21.2.1.4 Files Created by the Installation

[Table 21-2](#) describes the files that are installed by the Oracle ODBC Driver kit.

Table 21-2 Files Installed by the Oracle ODBC Driver Kit

Description	File Name for Windows Installation	File Name for UNIX Installation
Oracle ODBC Database Access DLL	<code>sqora32.dll</code>	<code>libsqora.so.12.1</code>

Table 21-2 (Cont.) Files Installed by the Oracle ODBC Driver Kit

Description	File Name for Windows Installation	File Name for UNIX Installation
Oracle ODBC Driver Setup DLL	sqoras32.dll	None
Oracle ODBC Resource DLL	sqresus.dll	None
Oracle ODBC Resource DLL for Japanese	sqresja.dll	None
Oracle ODBC Driver message file	oraodbcus.msb	oraodbcus.msb
Oracle ODBC Driver message file for Japanese	oraodbcja.msb	oraodbcja.msb
Oracle ODBC Driver release notes	<i>Oracle ODBC Driver Release Notes</i>	<i>Oracle ODBC Driver Release Notes</i>
Oracle ODBC Driver Instant Client Release Notes	ODBC_IC_Readme_Win.html	ODBC_IC_Readme_Unix.html
Oracle ODBC Driver help file	sqora.htm	sqora.htm
Oracle ODBC Driver help file for Japanese	sqora.htm	sqora.htm
Oracle ODBC Driver Instant Client install script	odbc_install.exe	odbc_update_ini.sh
Oracle ODBC Driver Instant Client uninstall script	odbc_uninstall.exe	None

Microsoft Driver Manager and Administrator Files

See the Microsoft ODBC 3.52 Software Development Kit and Programmer's Reference for the list of files that are installed with Microsoft's ODBC 3.52 Components.

The Microsoft ODBC components are packages in the Microsoft Data Access Component (MDAC) kit. Oracle ODBC driver on Windows has been tested using MDAC version 2.8. This can be downloaded from

<http://www.microsoft.com/download/en/search.aspx?q=ODBC+MDAC>

unixODBC Driver Manager and Administrator Files

See the unixODBC readme and INSTALL files for the list of files that are installed with unixODBC Driver Manager.

The unixODBC Driver Manager can be downloaded from

<http://www.unixodbc.org/download.html>

21.2.1.5 Driver Conformance Levels

ODBC defines Conformance Levels for drivers in two areas:

- ODBC application programming interface (API)
- ODBC SQL-99 syntax

The Oracle ODBC Driver supports all core API functionality and a limited set of Level 1 and Level 2 functionality. See [Section 21.4.11.1, "API Conformance,"](#) for more information.

The Oracle ODBC Driver is broadly compatible with the SQL-99 Core specification which is a superset of the SQL-92 Entry Level specification. Applications must call SQLGetInfo with the appropriate information type to retrieve a list of SQL-99 supported features.

21.2.1.6 Known Limitations

The following are not supported by Oracle ODBC driver:

- ODBC ASYNC interface
- Control-C to cancel execution in an application

21.2.2 Configuration Tasks

Topics:

- [Configuring Oracle Net Services](#)
- [Configuring the Data Source](#)
- [Oracle ODBC Driver Configuration Dialog Box](#)

21.2.2.1 Configuring Oracle Net Services

Before [Configuring the Data Source](#), you must configure network database services so there is an entry for each Transparent Network Substrate (TNS) Service Name. To do this, use Oracle Net Configuration Assistant (NETCA).

Using NETCA, you can create an entry in the `tnsnames.ora` file for each TNS Service Name. NETCA is installed when you install Oracle Net Services.

21.2.2.2 Configuring the Data Source

Note: The following configuration steps are for Windows users. Unix users must use the `odbc_update_ini.sh` file to create a DSN.

After installing the Oracle ODBC Driver and [Configuring Oracle Net Services](#), and before using the Oracle ODBC Driver, you must configure the data source.

Before an application can communicate with the data source, you must provide configuration information. The configuration information informs the Oracle ODBC Driver as to which information you want to access.

The data source consists of the data that you want to access, its associated operating system, database management system, and network platform used to access the database management system. The data source for requests submitted by the Oracle ODBC Driver is an Oracle database and supports transports available under Oracle Net Services.

To configure or add an Oracle data source:

After you have installed the Oracle ODBC Driver, use the ODBC Data Source Administrator to configure or add an Oracle data source for each of your Oracle databases. The Oracle ODBC Driver uses the information you enter when you add the data source to access the data. Follow these steps:

1. From the start menu, select Programs, Administrative Tools, Data Sources (ODBC). A list of installed drivers is displayed.

2. Click **Add** in the Create New Data Source window and then select the Oracle ODBC Driver in the list of installed drivers.
3. Click **Finish**. The [Oracle ODBC Driver Configuration Dialog Box](#) is displayed. You must enter the DSN and TNS Service Name. You can provide the other information requested in the dialog box, or you can leave the fields blank and provide the information when you run the application.
4. After you have entered the data, click **OK** or click **Return**.

You can change or delete a data source at any time. The following subtopics explain how to add, change, or delete a data source.

To modify an Oracle data source:

1. From the start menu, select **Programs, Administrative Tools, Data Sources(ODBC)**.
2. In the ODBC Data Source Administrator dialog box, select the data source from the Data Sources list and click **Configure**. The Oracle ODBC Driver Configuration dialog box is displayed.
3. In the [Oracle ODBC Driver Configuration Dialog Box](#), modify the option values as necessary and click **OK**.

To delete an Oracle data source:

1. From the start menu, select **Programs, Administrative Tools, Data Sources(ODBC)**.
2. In the ODBC Data Source Administrator dialog box, select the data source you want to delete from the Data Sources list.
3. Click **Remove**, and then click **Yes** to confirm the deletion.

Related Topics

[Connecting to an Oracle Data Source](#)

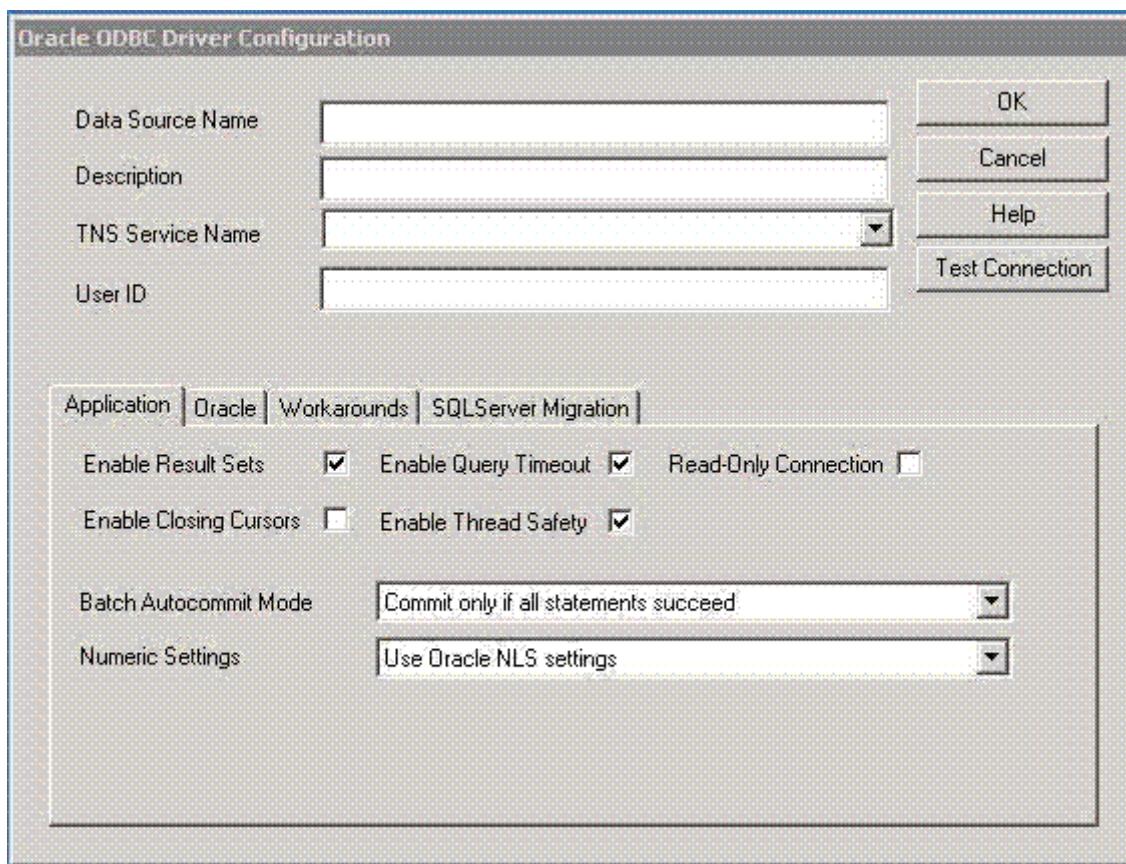
[Using the Oracle ODBC Driver for the First Time](#)

21.2.2.3 Oracle ODBC Driver Configuration Dialog Box

Note: The Oracle ODBC Driver Configuration dialog box is available only for Microsoft Windows users.

[Figure 21-3](#) is an example of the Oracle ODBC Driver Configuration dialog box.

Figure 21–3 Oracle ODBC Driver Configuration Dialog Box



The following list is an explanation of the main setup options and fields found on the Oracle ODBC Driver Configuration dialog box shown in the preceding graphic. The tabs found on the lower half of this dialog box are described in subsequent topics.

- **Data Source Name (DSN)** - The name used to identify the data source to ODBC. For example, "odbc-pc". You must enter a DSN.
- **Description** - A description or comment about the data in the data source. For example, "Hire date, salary history, and current review of all employees." The Description field is optional.
- **TNS Service Name** - The location of the Oracle database from which the ODBC driver will retrieve data. This is the same name entered in [Configuring Oracle Net Services](#) using the Oracle Net Configuration Assistant (NETCA). For more information, see the NETCA documentation and [Section 21.2.5.1, "Using the Oracle ODBC Driver for the First Time."](#) The TNS Service Name can be selected from a pull-down list of available TNS names. For example, "ODBC-PC". You must enter a TNS Service Name.
- **User ID** - The user name of the account on the server used to access the data. For example, "scott". The User ID field is optional.

You must enter the DSN and the TNS Service Name. You can provide the other information requested in the dialog box or you can leave the fields blank and provide the information when you run the application.

In addition to the main setup options previously described, there is a Test Connection button available. The Test Connection button verifies whether the ODBC environment

is configured properly by connecting to the database specified by the DSN definition. When you press the Test Connection button, you are prompted for the username and password.

For an explanation of the Options tabs found on the lower half of the Oracle ODBC Driver Configuration dialog box, click any of these links:

[Application Options](#)

[Oracle Options](#)

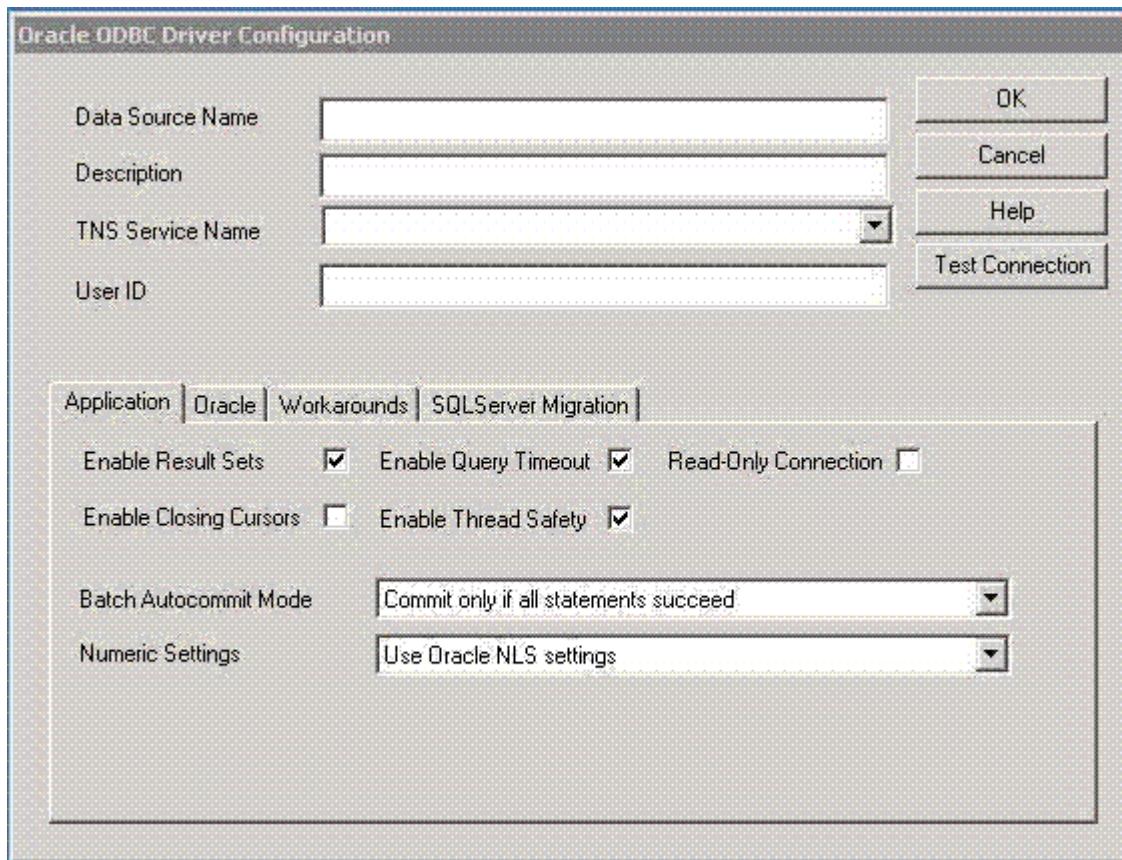
[Workarounds Options](#)

[SQL Server Migration Options](#)

Application Options

[Figure 21-4](#) is an example of the Application Options tab found on the Oracle ODBC Driver Configuration dialog box.

Figure 21-4 The Application Options Tab of the Oracle ODBC Driver Configuration Dialog Box



The following list is an explanation of the fields found on the Application Options tab shown in the preceding graphic:

- **Enable Result Sets** - Enables the processing of Oracle Result Sets. If Result Sets are not required for your application, Result Set support can be disabled. There is a small performance penalty for procedures called from packages not containing Result Sets. Result Sets are enabled by default.
- **Enable Query Timeout** - Enables query timeout for SQL queries. By default, the Oracle ODBC Driver supports the SQL_ATTR_QUERY_TIMEOUT attribute for the

SQLSetStmtAttr function. If this box is not checked, the Oracle ODBC Driver responds with a "not capable" message. Query Timeout is enabled by default.

- **Read-Only Connection** - Check this box to force read-only access. The default is write access.
- **Enable Closing Cursors** - Enables closing cursors. By default, closing cursors is disabled (the field is empty), meaning a call to close a cursor does not force the closing of OCI cursors when this behavior is not desired because it can cause an unnecessary performance hit. Enable closing cursors when you want to force the closing of OCI cursors upon a call to close a cursor.

Note: There is an impact on performance each time a cursor is closed.

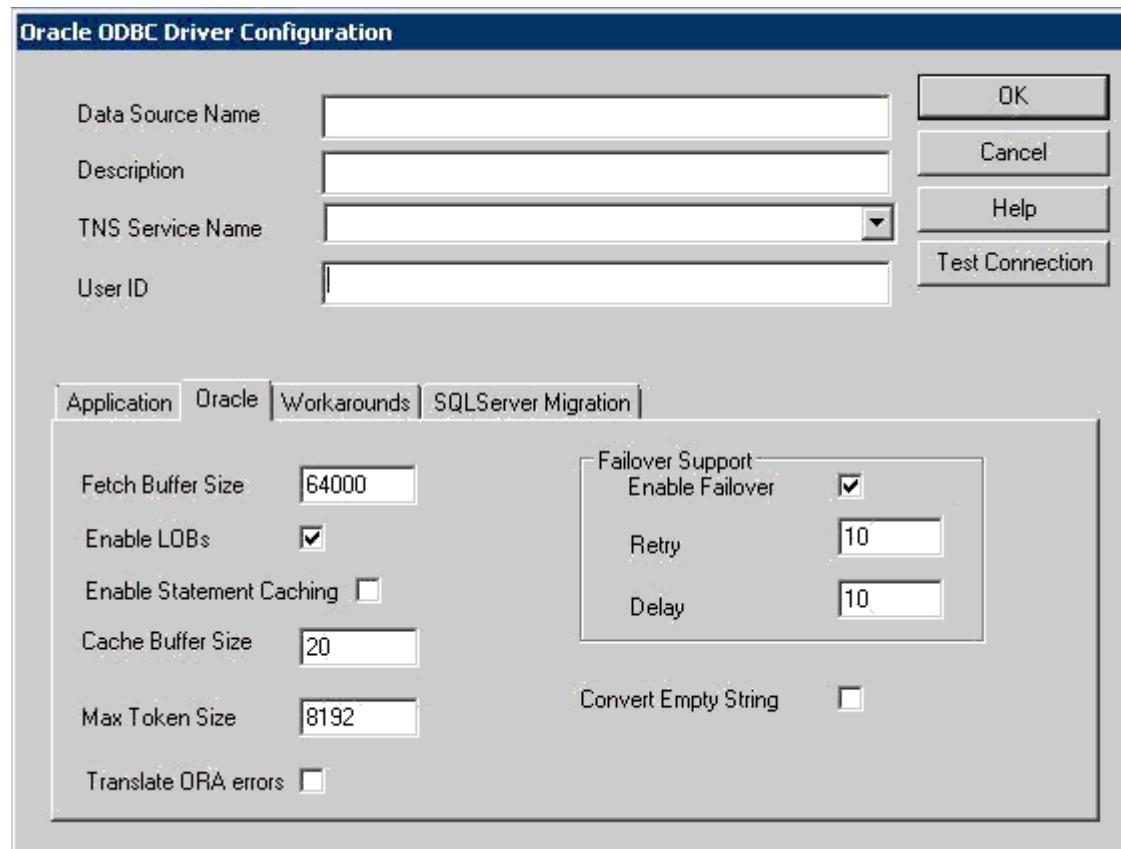
- **Enable Thread Safety** - Thread safety can be disabled for a data source. If thread safety is not required, disabling this option eliminates the overhead of using thread safety. By default, thread safety is enabled.
- **Batch Autocommit Mode** - By default, commit is executed only if all statements succeed.
- **Numeric Settings** - Allows you to choose the numeric settings that determine the decimal and group separator characters when receiving and returning numeric data that is bound as strings. This option allows you to choose Oracle NLS settings (the default setting), Microsoft default regional settings (to provide a way to mirror the Oracle OLE DB driver's behavior for greater interoperability), or US numeric settings (which are necessary when using MS Access or DAO (Database Access Objects) in non-US environments).

The main configuration setup options are described in the [Oracle ODBC Driver Configuration Dialog Box](#) topic.

Oracle Options

[Figure 21-5](#) is an example of the Oracle Options tab found on the Oracle ODBC Driver Configuration dialog box.

Figure 21–5 The Oracle Options Tab of the Oracle ODBC Driver Configuration Dialog Box



The following list is an explanation of the fields found on the Oracle Options tab shown in the preceding graphic:

- **Fetch Buffer Size** - The amount of memory used to determine how many rows of data the ODBC Driver prefetches at a time from an Oracle database regardless of the number of rows the application program requests in a single query. However, the number of prefetched rows depends on the width and number of columns specified in a single query. Applications that typically fetch fewer than 20 rows of data at a time improve their response time, particularly over slow network connections or to heavily loaded servers. Setting Fetch Buffer Size too high can make response time worse or consume large amounts of memory.

Note: When LONG and LOB data types are present, the number of rows prefetched by the ODBC Driver is not determined by the Fetch Buffer Size. The inclusion of the LONG and LOB data types minimizes the performance improvement and could result in excessive memory use. The ODBC Driver disregards Fetch Buffer Size and prefetches a set number of rows only in the presence of the LONG and LOB data types.

- **Enable LOBs** - Enables the writing of Oracle LOBs. If writing Oracle LOBs is not required for your application, LOB support can be disabled. There is a small performance penalty for insert and update statements when LOBs are enabled. LOB writing is enabled by default but disabled for Oracle databases that do not support the LOB data type.

- **Enable Statement Caching** - Enables statement caching feature, which increases the performance of parsing the query, in case the user has to parse the same text of query and related parameters multiple times. The default is disabled.
- **Cache Buffer Size** - The statement cache has a maximum size (number of statements) that can be modified by an attribute on the service context, OCI_ATTR_STMTCACHESIZE. The default cache buffer size is 20 that are used only if statement caching option is enabled. Setting cache buffer size to 0 disables statement caching feature.
- **Max Token Size** - Sets the token size to the nearest multiple of 1 KB (1024 bytes) beginning at 4 KB (4096 bytes). The default size is 8 KB (8192 bytes). The maximum value that can be set is 128 KB (131068 bytes).
- **Translate ORA errors** - Any migrated third party ODBC application, which is using the SQL Translation Framework feature, expects that errors returned by the server to be in their native database format, then users can enable this option to receive native errors based on the error translation registered with SQL Translation Profile.
- **Convert Empty String** - Any third party ODBC application that is migrated to Oracle Database requires handling empty string data (Oracle Database does not handle empty string data in table columns), then they can enable this option so that the application can insert empty string data or retrieve empty string data.

Note: This feature is not implemented for Oracle Database 12c Release 1 (12.1.0.1).

The Failover area of the Oracle Options tab contains the following fields:

- **Enable Failover** - Enables Oracle Fail Safe and Oracle Parallel Server failover retry. This option is an enhancement to the failover capabilities of Oracle Fail Safe and Oracle Parallel Server. Enable this option to configure additional failover retries. The default is enabled.
- **Retry** - The number of times the connection failover is attempted. The default is 10 attempts.
- **Delay** - The number of seconds to delay between failover attempts. The default is 10 seconds.

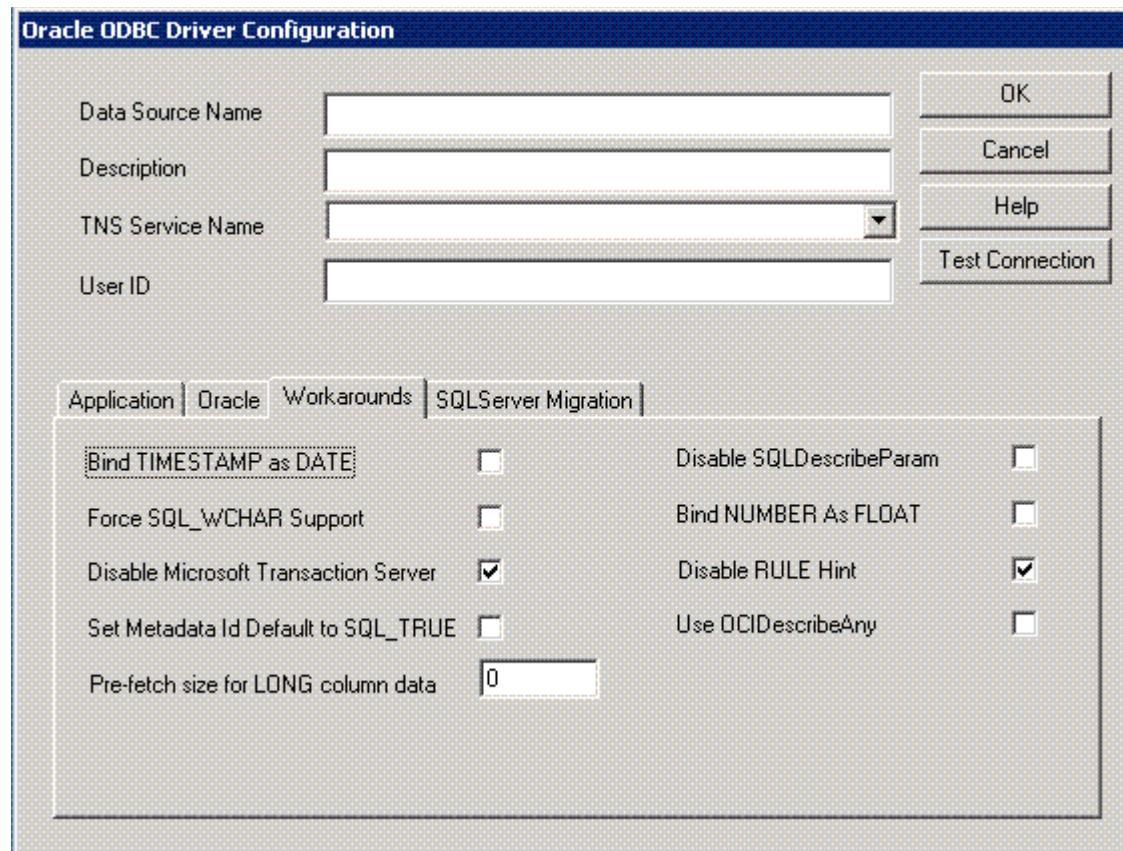
Note: See the Oracle Fail Safe and Oracle Parallel Server documentation on how to set up and use both of these products.

The main configuration setup options are described in the [Oracle ODBC Driver Configuration Dialog Box](#) topic.

Workarounds Options

[Figure 21-6](#) is an example of the Workarounds Options tab found on the Oracle ODBC Driver Configuration dialog box.

Figure 21–6 The Workarounds Options Tab of the Oracle ODBC Driver Configuration Dialog Box



The following list is an explanation of the fields found on the Workarounds Options tab shown in the preceding graphic:

- **Bind TIMESTAMP as DATE** - Check this box to force the Oracle ODBC Driver to bind `SQL_TIMESTAMP` parameters as the Oracle `DATE` type instead of as the Oracle `TIMESTAMP` type (the default). For more information, see [Section 21.3.4, "Implementation of Data Types \(Advanced\)"](#).
- **Force SQL_WCHAR Support** - Check this box to enable `SQLDescribeCol`, `SQLColumns`, and `SQLProcedureColumns` to unconditionally return the data type of `SQL_WCHAR` for `SQL_CHAR` columns; `SQL_WVARCHAR` for `SQL_VARCHAR` columns; and `SQL_WLONGVARCHAR` for `SQL_LONGVARCHAR` columns. This feature enables Unicode support in applications that rely on the results of these ODBC calls (for example, ADO). This support is disabled by default.
- **Disable Microsoft Transaction Server** - Clear the check in this box to enable Microsoft Transaction Server (MTS) support. By default, MTS support is disabled.
- **Set Metadata Id Default to SQL_TRUE** - Check this box to change the default value of the `SQL_ATTR_METADATA_ID` connection and statement attribute at connection time to `SQL_TRUE`. Under normal circumstances, `SQL_ATTR_METADATA_ID` would default to `SQL_FALSE`. ODBC calls made by the application to specifically change the value of the attribute after connection time are unaffected by this option and complete their functions as expected. By default, this option is off. The [Implementation of ODBC API Functions](#) topic provides additional information about the `SQL_ATTR_METADATA_ID` attribute.
- **Prefetch size for LONG column data** - Set this value to prefetch `LONG` or `LONG RAW` data to improve performance of ODBC applications. This enhancement improves

the performance of Oracle ODBC driver up to 10 times, depending on the prefetch size set by the user. The default value is 0. The maximum value that you can set is 64 KB (65536 bytes).

If the value of prefetch size is greater than 65536, the data fetched is only 65536 bytes. If you have LONG or LONG RAW data in the database that is greater than 65536 bytes, then set the prefetch size to 0 (the default value), which causes single-row fetch and fetches complete LONG data. If you pass a buffer size less than the prefetch size in nonpolling mode, a data truncation error occurs if the LONG data size in the database is greater than the buffer size.

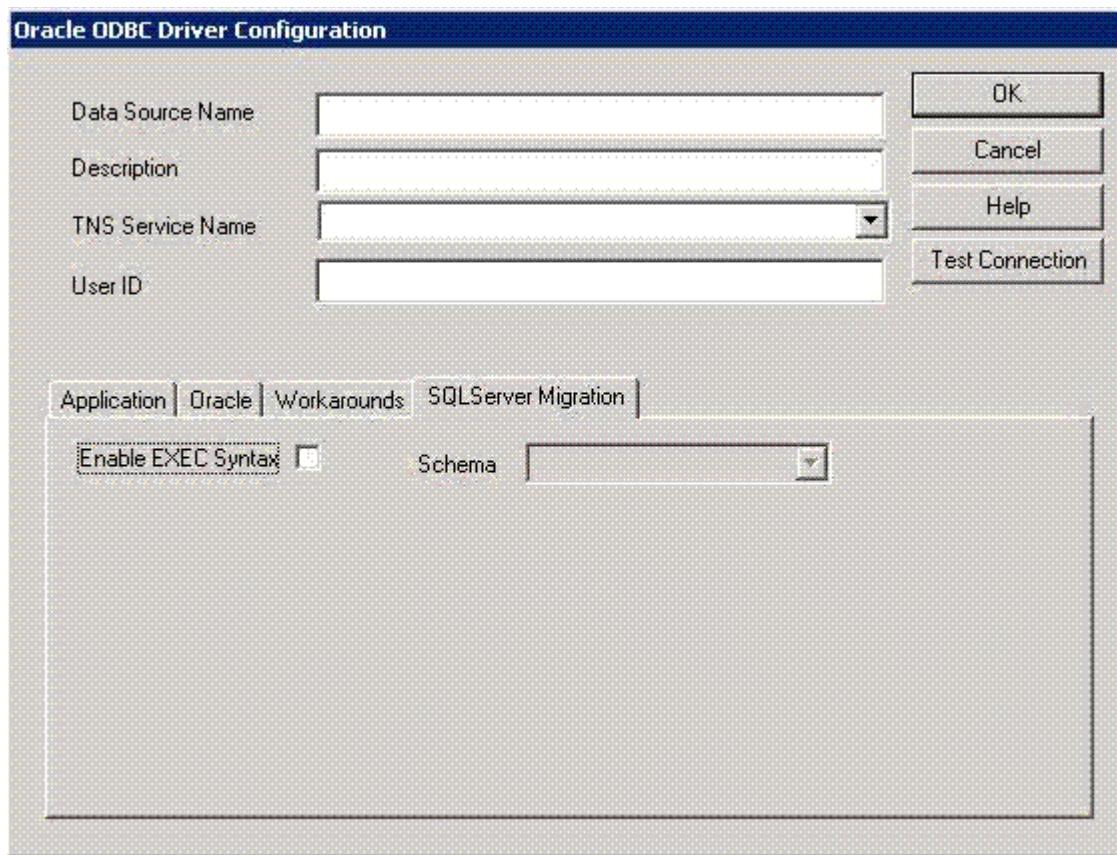
- **Disable SQLDescribeParam** - If the `SQLDescribeParam` function is enabled, the `SQL_VARCHAR` data type is returned for all parameters. If the Force `SQL_WCHAR` Support function is also enabled, the `SQL_WVARCHAR` data type is returned for all parameters. By default, this function is enabled.
- **Bind NUMBER as FLOAT** - Check this box to force the Oracle ODBC Driver to bind `NUMBER` column containing `FLOAT` data as `Float` instead of as the `Binary Float` (the default).
- **Disable RULE Hint** - Clear the check in this box to enable `RULE Hint` specified with catalogue queries. By default, `RULE Hint` option is disabled.
- **Use OCIDescribeAny** - Check this box to gain a performance improvement by forcing the driver to use `OCIDescribeAny()` when an application makes heavy calls to small packaged procedures that return `REF CURSORS`.

The main configuration setup options are described in the [Oracle ODBC Driver Configuration Dialog Box](#) topic.

SQL Server Migration Options

[Figure 21-7](#) is an example of the SQL Server Migration Options tab found on the Oracle ODBC Driver Configuration dialog box.

Figure 21–7 The SQL Server Migration Options Tab of the Oracle ODBC Driver Configuration Dialog Box



The fields of the SQL Server Migration Options tab in the preceding graphic are:

- **EXEC Syntax Enabled**, which enables support for SQL Server EXEC syntax. A subprogram call specified in an EXEC statement is translated to its equivalent Oracle subprogram call before being processed by an Oracle database server. By default this option is disabled.
- **Schema**, which is the translated Oracle subprogram assumed to be defined in the user's default schema. However, if all subprograms from the same SQL Server database are migrated to the same Oracle schema with their database name as the schema name, then set this field to database. If all subprograms owned by the same SQL Server user are defined in the same Oracle schema, then set this field to owner. This field is empty by default.

The main configuration setup options are described in the [Oracle ODBC Driver Configuration Dialog Box](#) topic.

21.2.3 Modifying the oraodbc.ini File

Topics:

- [Reducing Lock Timeout](#)

21.2.3.1 Reducing Lock Timeout

An Oracle server waits indefinitely for lock conflicts between transactions to be resolved. You can limit the amount of time that an Oracle server waits for locks to be resolved by setting the Oracle ODBC Driver's LockTimeOut entry in the oraodbc.ini

file. The value you enter for the `LockTimeOut` parameter is the number of seconds after which an Oracle server times out if it cannot obtain the requested locks. In the following example, the Oracle server times out after 60 seconds:

```
[Oracle ODBC Driver Common]  
LockTimeOut=60
```

21.2.4 Connecting to a Data Source

Topics:

- [Connecting to an Oracle Data Source](#)

21.2.4.1 Connecting to an Oracle Data Source

To connect to a Data Source, the Oracle ODBC Driver requires that the OCI client software be installed on your computer and the corresponding listener be running on the Oracle server. Oracle Net Services for Windows is a Dynamic Linked Library (DLL) based application. For more information about Oracle Net Services, see the Oracle Net Services documentation.

As part of the connection process, an application can prompt you for information. If an application prompts you for information about an Oracle data source, do the following:

1. In the TNS Service Name box, enter the name of the TNS service.
2. In the User Name box, enter the name you use to access an Oracle Database.
3. In the Password box, enter the password you use to access an Oracle Database.
4. Click **OK**.

An application must connect to a data source to access the data in it. Different applications connect to data sources at different times. For example, an application might connect to a data source only at your request, or it might connect automatically when it starts. For information about when an application connects to a data source, see the documentation for that application.

For additional information, click any of these links.

Related Topic for All Users

[Configuring the Data Source](#)

Related Topics for Programmers

[SQLDriverConnect Implementation](#)

[Data Source Configuration Options](#)

21.2.5 Troubleshooting

Topics:

- [Using the Oracle ODBC Driver for the First Time](#)
- [Expired Password](#)

21.2.5.1 Using the Oracle ODBC Driver for the First Time

Note: This information is only for Microsoft Windows users.

For the Oracle ODBC Driver to function successfully, *OracleHome/bin* must be in the system path. To verify this, type PATH from a command prompt.

The first time you use Oracle ODBC Driver and attempt to connect to a database or table, you might see these messages:

- Your machine or server system is missing required software
- Client/Server connectivity is incorrectly set up on your machine
- TNS Service Name does not match the name that was entered in the Oracle Net Configuration Assistant (NETCA)
- User does not have access to the database
- System Error: 182 or 193 when trying to create an ODBC DSN
- Translator Library could not be loaded system error code 31

Your machine or server system is missing required software

Problem: Oracle ODBC Driver fails to work.

Cause: Either OCI software is not installed on your machine or Oracle database software is not installed on your server system.

Recommended Action: Install the required OCI software on your client machine or Oracle database software on your server system or both.

Client/Server connectivity is incorrectly set up on your machine

Problem: Cannot connect to the server system from your machine.

Cause: Either the required transport software is not installed or is not configured correctly.

Recommended Action: As a test, for example when using TCP/IP, make sure that your machine can ping to the server where the Oracle database resides. Use *tnsping.exe* located in the */orant/bin* or */Oracle/ora90/bin* directory to ensure connectivity to a specific database service. For example:

C:/ORANT/BIN>tnsping database-service-name

TNS Service Name does not match the name that was entered in the Oracle Net Configuration Assistant (NETCA)

Problem: The user is returned an error message about the TNS Service Name while attempting to set up the Oracle ODBC Driver.

Cause: The TNS Service Name does not match the name entered in NETCA.

Recommended Action: Change the TNS Service Name in the Oracle ODBC Driver setup window to match the TNS Service Name in NETCA.

Note: Refer to the pull-down menu on the Datasource Configuration screen to view a list of all valid TNS service names on the system as well as names entered by users.

User does not have access to the database

Problem: The user is returned an access denied error message when attempting to gain access to an Oracle database while using the Oracle ODBC Driver.

Cause: You do not have access to an Oracle database.

Recommended Action: Ensure the proper privileges are assigned for the user to gain access to the Oracle database.

System Error: 182 or 193 when trying to create an ODBC DSN

Problem: You are trying to create a DSN with the ODBC Administrator but, after selecting the Oracle ODBC Driver, you received a System Error 182 or System Error 193.

Cause: This error is due to a mismatch in the `mfc42.dll` provided by Microsoft.

Recommended Action: Verified that `OracleHome/bin` is in your PATH and that you have no duplicate `oci.dll` outside the `OracleHome/bin`. If you still receive the error, copy the `mfc42.dll` from a working machine to the machine with the problem.

Translator Library could not be loaded system error code 31

Problem: On Windows with the Oracle ODBC Driver in the Microsoft ODBC Administrator trying to Add or Delete an Oracle ODBC DSN, you get the error "Translator Library could not be loaded System error code 31."

Cause: In some cases, a Windows machine contains an outdated version of the `mfc42.dll`. Oracle ODBC is built against the newer version of `mfc42.dll` (specifically, version 6.0.8665.0).

Recommended Action: An outdated version of the Microsoft DLL `mfc42.dll` is causing this error. Installing MDAC 2.5 SP1 does solve this problem. You must obtain a newer version of the `mfc42.dll` (version 6.0.8665.0 or higher) and replace the outdated `mfc42.dll` in the `%WINNT%/System32` directory. Either copy the DLL from another machine that works correctly or contact Microsoft to find out how to obtain the DLL.

21.2.5.2 Expired Password

This section contains information about expired passwords.

Expired Password Behavior

If you try to connect to the database and your password has expired, you are prompted to change your password. Upon making a successful password change, you are connected to the database. However, if you try to connect to the database with a `SQLDriverConnect` call with a `SQL_DRIVER_NOPROMPT` parameter value, the Oracle ODBC Driver does not prompt you for the password change. Instead, an error condition results, producing an error message and number that indicates that the password has expired.

21.3 For Advanced Users

Topics:

- [Creating Oracle ODBC Driver TNS Service Names](#)
- [SQL Statements](#)
- [Data Types](#)
- [Implementation of Data Types \(Advanced\)](#)
- [Limitations on Data Types](#)
- [Error Messages](#)

21.3.1 Creating Oracle ODBC Driver TNS Service Names

To create Oracle ODBC Driver TNS Service Names with Oracle Net Services, use the Oracle Net Configuration Assistant (NETCA), which is installed when you install Oracle Net Services. NETCA creates Oracle ODBC Driver TNS Service Name entries in the `tnsnames.ora` file.

21.3.2 SQL Statements

The Oracle ODBC Driver is broadly compatible with the SQL-99 Core specification which is a superset of the SQL-92 Entry Level specification. In addition to Oracle's grammar, the vendor-specific escape sequences outlined in Appendix C of the ODBC specifications are also supported. In accordance with the design of ODBC, the Oracle ODBC Driver passes native SQL syntax to the Oracle database.

Related Topic for Advanced Users

[Data Types](#)

Related Topic for Programmers

[Implementation of the ODBC SQL Syntax](#)

21.3.3 Data Types

The Oracle ODBC Driver maps Oracle database data types to ODBC SQL data types.

Note: All conversions in Appendix D of the *Microsoft ODBC 3.52 Software Development Kit and Programmer's Reference* are supported for the ODBC SQL data types listed from a call to `SQLGetInfo` with the appropriate information type.

Related Topics for Advanced Users

[Implementation of Data Types \(Advanced\)](#)

[Limitations on Data Types](#)

[SQL Statements](#)

Related Topic for Programmers

[Implementation of Data Types \(Programming\)](#)

21.3.4 Implementation of Data Types (Advanced)

Topics:

- [DATE and TIMESTAMP](#)
- [Floating Point Data Types](#)

DATE and TIMESTAMP

The semantics of Oracle DATE and TIMESTAMP data types do not correspond exactly with the ODBC data types with the same names. The Oracle DATE data type contains both date and time information while the `SQL_DATE` data type contains only date information. The Oracle TIMESTAMP data type also contains date and time information, but it has greater precision in fractional seconds. The ODBC Driver reports the data

types of both Oracle DATE and TIMESTAMP columns as SQL_TIMESTAMP to prevent information loss. Similarly the ODBC Driver binds SQL_TIMESTAMP parameters as Oracle TIMESTAMP values.

Floating Point Data Types

When connected to a 10.1 or later Oracle server, the ODBC Driver maps the Oracle floating point data types BINARY_FLOAT and BINARY_DOUBLE to the ODBC data types SQL_REAL and SQL_DOUBLE, respectively. In previous releases, SQL_REAL and SQL_DOUBLE mapped to the generic Oracle numeric data type.

Related Topic

[DATE and TIMESTAMP Data Types](#)

21.3.5 Limitations on Data Types

The Oracle ODBC Driver and the Oracle database impose limitations on data types. Table 21–3 describes these limitations.

Table 21–3 Limitations Imposed on Data Types by the Oracle ODBC Driver and Oracle Database

Limited Data Type	Description
Literals	Oracle database limits literals in SQL statements to 4,000 bytes.
SQL_LONGVARCHAR and SQL_WLONGVARCHAR	Oracle's limit for SQL_LONGVARCHAR data where the column type is LONG is 2,147,483,647 bytes. Oracle's limit for the SQL_LONGVARCHAR data where the column type is CLOB is 4 gigabytes. The limiting factor is the client workstation memory.
SQL_LONGVARCHAR and SQL_LONGVARBINARY	Oracle database allows only a single long data column per table. The long data types are SQL_LONGVARCHAR (LONG) and SQL_LONGVARBINARY (LONG RAW). Oracle recommends you use CLOB and BLOB columns instead. There is no restriction on the number of CLOB and BLOB columns in a table.

21.3.6 Error Messages

When an error occurs, the Oracle ODBC Driver returns the native error number, the SQLSTATE (an ODBC error code), and an error message. The driver derives this information both from errors detected by the driver and errors returned by the Oracle server.

Native Error

For errors that occur in the data source, the Oracle ODBC Driver returns the native error returned to it by the Oracle server. When the Oracle ODBC Driver or the Driver Manager detects an error, the Oracle ODBC Driver returns a native error of zero.

SQLSTATE

For errors that occur in the data source, the Oracle ODBC Driver maps the returned native error to the appropriate SQLSTATE. When the Oracle ODBC Driver detects an error, it generates the appropriate SQLSTATE. When the Driver Manager detects an error, it generates the appropriate SQLSTATE.

Error Message

For errors that occur in the data source, the Oracle ODBC Driver returns an error message based on the message returned by the Oracle server. For errors that occur in

the Oracle ODBC Driver or the Driver Manager, the Oracle ODBC Driver returns an error message based on the text associated with the SQLSTATE.

Error messages have the following format:

[vendor] [ODBC-component] [data-source] error-message

The prefixes in brackets ([]) identify the source of the error. [Table 21–4](#) shows the values of these prefixes returned by the Oracle ODBC Driver. When the error occurs in the data source, the [vendor] and [ODBC-component] prefixes identify the vendor and name of the ODBC component that received the error from the data source.

Table 21–4 Error Message Values of Prefixes Returned by the Oracle ODBC Driver

Error Source	Prefix	Value
Driver Manager	[vendor]	[Microsoft/unixODBC]
	[ODBC-component]	[ODBC Driver Manager]
	[data-source]	N/A
Oracle ODBC Driver	[vendor]	[ORACLE]
	[ODBC-component]	[ODBC Driver]
	[data-source]	N/A
Oracle server	[vendor]	[ORACLE]
	[ODBC-component]	[ODBC Driver]
	[data-source]	N/A

For example, if the error message does not contain the [Ora] prefix shown in the following format, the error is an Oracle ODBC Driver error and should be self-explanatory.

[Oracle] [ODBC] Error message text here

If the error message contains the [Ora] prefix shown in the following format, it is not an Oracle ODBC Driver error.

Note: Although the error message contains the [Ora] prefix, the actual error may be coming from one of several sources.

[Oracle] [ODBC] [Ora] Error message text here

If the error message text starts with the following prefix, you can obtain more information about the error in the Oracle server documentation.

ORA-

Oracle Net Services errors and Trace logging are located under the *ORACLE_HOME\NETWORK* directory on Windows systems or the *ORACLE_HOME/NETWORK* directory on UNIX systems where the OCI software is installed and specifically in the log and trace directories respectively. Database logging is located under the *ORACLE_HOME\RDBMS* directory on Windows systems or the *ORACLE_HOME/rdbms* directory on UNIX systems where the Oracle server software is installed.

See the Oracle server documentation for more information about server error messages.

21.4 For Programmers

Topics:

- [Format of the Connection String](#)
- [SQLDriverConnect Implementation](#)
- [Reducing Lock Timeout in a Program](#)
- [Linking with odbc32.lib \(Windows\) or libodbc.so \(UNIX\)](#)
- [Obtaining Information About rowids](#)
- [Rowids in a WHERE Clause](#)
- [Enabling Result Sets](#)
- [Enabling EXEC Syntax](#)
- [Enabling Event Notification for Connection Failures in an Oracle RAC Environment](#)
- [Using Implicit Results Feature Through ODBC](#)
- [Supported Functionality](#)
- [Unicode Support](#)
- [Performance and Tuning](#)

21.4.1 Format of the Connection String

Table 21–5 describes keywords that can be included in the connection string argument of the SQLDriverConnect function call. Missing keywords are read from the Administrator entry for the data source. Values specified in the connection string override those contained in the Administrator entry. See the *Microsoft ODBC 3.52 Software Development Kit and Programmer's Reference* for more information about the SQLDriverConnect function.

Table 21–5 Keywords that Can Be Included in the Connection String Argument of the SQLDriverConnect Function Call

Keyword	Meaning	Comments
DSN	ODBC Data Source Name	User-supplied name.
DBQ	TNS Service Name	User-supplied name.
UID	User ID or User Name	User-supplied name.
PWD	Password	User-supplied password. Specify PWD=; for an empty password.
DBA	Database Attribute	W=write access. R=read-only access.
APA	Applications Attributes	T=Thread Safety Enabled. F=Thread Safety Disabled.
RST	Result Sets	T=Result Sets Enabled. F=Result Sets Disabled.
QTO	Query Timeout Option	T=Query Timeout Enabled. F=Query Timeout Disabled.

Table 21-5 (Cont.) Keywords that Can Be Included in the Connection String Argument of the SQLDriverConnect Function Call

Keyword	Meaning	Comments
CSR	Close Cursor	T=Close Cursor Enabled. F=Close Cursor Disabled.
BNF	Bind NUMBER as FLOAT	T=Bind NUMBER as FLOAT. F=Bind NUMBER as NUMBER.
DRH	Disable Rule Hint	T=Disable Rule Hint. F=Enable Rule Hint.
BAM	Batch Autocommit Mode	IfAllSuccessful=Commit only if all statements are successful (old behavior). UpToFirstFailure=Commit up to first failing statement (V7 ODBC behavior). AllSuccessful=Commit all successful statements (only when connected to an Oracle database; against other databases, same behavior as V7).
FBS	Fetch Buffer Size	User-supplied numeric value (specify a value in bytes of 0 or greater). The default is 60,000 bytes.
FEN	Failover	T=Failover Enabled. F=Failover Disabled.
FRC	Failover Retry Count	User-supplied numeric value. The default is 10.
FDL	Failover Delay	User-supplied numeric value. The default is 10.
LOB	LOB Writes	T=LOBs Enabled. F=LOBs Disabled.
MTS	Microsoft Transaction Server Support	T=Disabled. F=Enabled.
FWC	Force SQL_WCHAR Support	T=Force SQL_WCHAR Enabled. F=Force SQL_WCHAR Disabled.
EXC	EXEC Syntax	T=EXEC Syntax Enabled. F=EXEC Syntax Disabled.
XSM	Schema Field	Default=Default. Database=Database Name. Owner=Owner Name.
MDI	Set Metadata ID Default	T=SQL_ATTR_METADATA_ID defaults to SQL_TRUE. F=SQL_ATTR_METADATA_ID defaults to SQL_FALSE.
DPM	Disable SQLDescribeParam	T=SQLDescribeParam Disabled. F=SQLDescribeParam Enabled.
BTD	Bind TIMESTAMP as DATE	T=Bind SQL_TIMESTAMP as Oracle DATE F=Bind SQL_TIMESTAMP as Oracle TIMESTAMP

Table 21–5 (Cont.) Keywords that Can Be Included in the Connection String Argument of the SQLDriverConnect Function Call

Keyword	Meaning	Comments
NUM	Numeric Settings	NLS=Use Oracle NLS numeric settings (to determine the decimal and group separator). MS=Use Microsoft regional settings. US=Use US settings.
ODA	Use OCIDescribeAny()	T= Use OCIDescribeAny() call to gain performance improvement when application makes heavy calls to small packaged procedures that return REF CURSORS. F= Do not use OCIDescribeAny(). By default, use OCIDescribeAny() value is FALSE.
STE	SQL Translate ORA Errors Specifies whether the Oracle ODBC Driver is to translate the Oracle error codes	T=Translate ORA errors. F=Do not translate any ORA error. By default, SQLTranslateErrors is FALSE.
TSZ	Token Size	User-supplied numeric value. Sets the token size to the nearest multiple of 1 KB (1024 bytes) beginning at 4 KB (4096 bytes). The default size is 8 KB (8192 bytes). The maximum value that can be set is 128 KB (131068 bytes).

If the following keyword is specified in the connection string, the Oracle ODBC Driver does not read values defined from the Administrator:

DRIVER={Oracle ODBC Driver}

Examples of valid connection strings are:

- 1) DSN=Personnel;UID=Kotzwinkle;PWD=;
- 2) DRIVER={Oracle ODBC Driver};UID=Kotzwinkle;PWD=whatever;DBQ=inst1_alias;DBA=W;

For additional information, click any of these links.

Related Topic for All Users

[Connecting to an Oracle Data Source](#)

Related Topic for Programmers

[SQLDriverConnect Implementation](#)

21.4.2 SQLDriverConnect Implementation

Table 21–6 describes the keywords required by the SQLDriverConnect connection string.

Table 21–6 Keywords Required by the SQLDriverConnect Connection String

Keyword	Description
DSN	The name of the data source.
DBQ	The TNS Service Name. See Section 21.3.1, "Creating Oracle ODBC Driver TNS Service Names." For more information, see the Oracle Net Services documentation.

Table 21–6 (Cont.) Keywords Required by the SQLDriverConnect Connection String

Keyword	Description
UID	The user login ID or user name.
PWD	The user-specified password.

21.4.3 Reducing Lock Timeout in a Program

The Oracle server waits indefinitely for lock conflicts between transactions to be resolved. You can limit the amount of time that the Oracle server waits for locks to be resolved by calling the ODBC SQLSetConnectAttr function before connecting to the data source. Specify a nonzero value for the SQL_ATTR_QUERY_TIMEOUT attribute in the ODBC SQLSetStmtAttr function.

If you specify a lock timeout value using the ODBC SQLSetConnectAttr function, it overrides any value specified in the oraodbc.ini file. Refer to [Reducing Lock Timeout](#) for more information on specifying a value in the oraodbc.ini file.

21.4.4 Linking with `odbc32.lib` (Windows) or `libodbc.so` (UNIX)

For Windows platforms, when you link your program, you must link it with the import library `odbc32.lib`.

For UNIX platforms, an ODBC application must be linked to `libodbc.so`.

21.4.5 Obtaining Information About rowids

The ODBC SQLSpecialColumns function returns information about the columns in a table. When used with the Oracle ODBC Driver, it returns information about the Oracle rowids associated with an Oracle table.

21.4.6 Rowids in a WHERE Clause

Rowids can be used in the `WHERE` clause of an SQL statement. However, the `rowid` value must be presented in a parameter marker.

21.4.7 Enabling Result Sets

Oracle reference cursors (Result Sets) allow an application to retrieve data using stored procedures and stored functions. The following information identifies how to use reference cursors to enable Result Sets through ODBC.

- The ODBC syntax for calling stored procedures must be used. Native PL/SQL is not supported through ODBC. The following identifies how to call the procedure or function without a package and within a package. The package name in this case is RSET.

```
Procedure call:  
{CALL Example1(?)}  
{CALL RSET.Example1(?)}  
Function Call:  
{? = CALL Example1(?)}  
{? = CALL RSET.Example1(?)}
```

- The PL/SQL reference cursor parameters are omitted when calling the procedure. For example, assume procedure Example2 is defined to have four parameters. Parameters 1 and 3 are reference cursor parameters and parameters 2 and 4 are character strings. The call is specified as:

```
{CALL RSET.Example2("Literal 1", "Literal 2")}
```

The following example application shows how to return a Result Set using the Oracle ODBC Driver:

```
/*
 * Sample Application using Oracle reference cursors via ODBC
 *
 * Assumptions:
 *
 * 1) Oracle Sample database is present with data loaded for the EMP table.
 *
 * 2) Two fields are referenced from the EMP table ename and mgr.
 *
 * 3) A data source has been setup to access the sample database.
 *
 *
 * Program Description:
 *
 * Abstract:
 *
 * This program demonstrates how to return result sets using
 * Oracle stored procedures
 *
 * Details:
 *
 * This program:
 * Creates an ODBC connection to the database.
 * Creates a Packaged Procedure containing two result sets.
 * Executes the procedure and retrieves the data from both result sets.
 * Displays the data to the user.
 * Deletes the package then logs the user out of the database.
 *
 *
 * The following is the actual PL/SQL this code generates to
 * create the stored procedures.
 *
*
DROP PACKAGE ODBCRefCur;
CREATE PACKAGE ODBCRefCur AS
TYPE ename_cur IS REF CURSOR;
TYPE mgr_cur IS REF CURSOR;
PROCEDURE EmpCurs(Ename IN OUT ename_cur, Mgr IN OUT mgr_cur, pjob IN VARCHAR2);

END;
/
CREATE PACKAGE BODY ODBCRefCur AS
PROCEDURE EmpCurs(Ename IN OUT ename_cur, Mgr IN OUT mgr_cur, pjob IN VARCHAR2)
AS
BEGIN
IF NOT Ename%ISOPEN
THEN
OPEN Ename for SELECT ename from emp;
END IF;

IF NOT Mgr%ISOPEN
THEN
OPEN Mgr for SELECT mgr from emp where job = pjob;
```

```
END IF;
END;
END;
/

/*
* End PL/SQL for Reference Cursor.
*/



/*
* Include Files
*/
#include <windows.h>
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>



/*
* Defines
*/
#define JOB_LEN 9
#define DATA_LEN 100
#define SQL_STMT_LEN 500



/*
* Procedures
*/
void DisplayError( SWORD HandleType, SQLHANDLE hHandle, char *Module );

/*
* Main Program
*/
int main()
{
SQLHENV hEnv;
SQLHDBC hDbc;
SQLHSTMT hStmt;
SQLRETURN rc;
char *DefUserName ="scott";
char *DefPassWord ="tiger";
SQLCHAR ServerName[DATA_LEN];
SQLCHAR *pServerName=ServerName;
SQLCHAR UserName[DATA_LEN];
SQLCHAR *pUserName=UserName;
SQLCHAR PassWord[DATA_LEN];
SQLCHAR *pPassWord=PassWord;
char Data[DATA_LEN];
SQLINTEGER DataLen;
char error[DATA_LEN];
char *charptr;
SQLCHAR SqlStmt[SQL_STMT_LEN];
SQLCHAR *pSqlStmt=SqlStmt;
```

```
char *pSalesMan = "SALESMAN";
SQLINTEGER sqlnts=SQL_NTS;

/*
 * Allocate the Environment Handle
 */
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv );
if (rc != SQL_SUCCESS)
{
printf( "Cannot Allocate Environment Handle/n");
printf( "/nHit Return to Exit/n");
charptr = gets ((char *)error);
exit(1);
}

/*
 * Set the ODBC Version
 */
rc = SQLSetEnvAttr( hEnv,
SQL_ATTR_ODBC_VERSION,
(void *)SQL_OV_ODBC3,
0);
if (rc != SQL_SUCCESS)
{
printf( "Cannot Set ODBC Version/n");
printf( "/nHit Return to Exit/n");
charptr = gets ((char *)error);
exit(1);
}

/*
 * Allocate the Connection handle
 */
rc = SQLAllocHandle( SQL_HANDLE_DBC, hEnv, &hDbc );
if (rc != SQL_SUCCESS)
{
printf( "Cannot Allocate Connection Handle/n");
printf( "/nHit Return to Exit/n");
charptr = gets ((char *)error);
exit(1);
}

/*
 * Get User Information
 */
lstrcpy( (char *) pUserName, DefUserName );
lstrcpy( (char *) pPassWord, DefPassWord );

/*
 * Data Source name
```

```

/*
printf( "/nEnter the ODBC Data Source Name/n" );
charptr = gets ((char *) ServerName);

/*
* User Name
*/
printf ( "/nEnter User Name Default [%s]/n", pUserName );
charptr = gets ((char *) UserName);
if (*charptr == '/0')
{
lstrcpy( (char *) pUserName, (char *) DefUserName );
}

/*
* Password
*/
printf ( "/nEnter Password Default [%s]/n", pPassWord );
charptr = gets ((char *) PassWord);
if (*charptr == '/0')
{
lstrcpy( (char *) pPassWord, (char *) DefPassWord );
}

/*
* Connection to the database
*/
rc = SQLConnect( hDbc,
pServerName,
(SQLSMALLINT) lstrlen((char *)pServerName),
pUserName,
(SQLSMALLINT) lstrlen((char *)pUserName),
pPassWord,
(SQLSMALLINT) lstrlen((char *)pPassWord));
if (rc != SQL_SUCCESS)
{
DisplayError(SQL_HANDLE_DBC, hDbc, "SQLConnect");
}

/*
* Allocate a Statement
*/
rc = SQLAllocHandle( SQL_HANDLE_STMT, hDbc, &hStmt );
if (rc != SQL_SUCCESS)
{
printf( "Cannot Allocate Statement Handle/n");
printf( "/nHit Return to Exit/n");
charptr = gets ((char *)error);
exit(1);
}

```

```

/*
 * Drop the Package
 */
lstrcpy( (char *) pSqlStmt, "DROP PACKAGE ODBCRefCur");
rc = SQLExecDirect(hStmt, pSqlStmt, lstrlen((char *)pSqlStmt));

/*
 * Create the Package Header
 */
lstrcpy( (char *) pSqlStmt, "CREATE PACKAGE ODBCRefCur AS/n");
lstrcat( (char *) pSqlStmt, " TYPE ename_cur IS REF CURSOR;/n");
lstrcat( (char *) pSqlStmt, " TYPE mgr_cur IS REF CURSOR;/n/n");
lstrcat( (char *) pSqlStmt, " PROCEDURE EmpCurs (Ename IN OUT ename_cur,");
lstrcat( (char *) pSqlStmt, "Mgr IN OUT mgr_cur,pjob IN VARCHAR2);/n/n");
lstrcat( (char *) pSqlStmt, "END;/n");
rc = SQLExecDirect(hStmt, pSqlStmt, lstrlen((char *)pSqlStmt));
if (rc != SQL_SUCCESS)
{
    DisplayError(SQL_HANDLE_STMT, hStmt, "SQLExecDirect");
}

/*
 * Create the Package Body
 */
lstrcpy( (char *) pSqlStmt, "CREATE PACKAGE BODY ODBCRefCur AS/n");
lstrcat( (char *) pSqlStmt, " PROCEDURE EmpCurs (Ename IN OUT ename_cur,");
lstrcat( (char *) pSqlStmt, "Mgr IN OUT mgr_cur, pjob IN VARCHAR2)/n AS/n
BEGIN/n");
lstrcat( (char *) pSqlStmt, " IF NOT Ename%ISOPEN/n THEN/n");
lstrcat( (char *) pSqlStmt, " OPEN Ename for SELECT ename from emp;/n");
lstrcat( (char *) pSqlStmt, " END IF;/n/n");
lstrcat( (char *) pSqlStmt, " IF NOT Mgr%ISOPEN/n THEN/n");
lstrcat( (char *) pSqlStmt, " OPEN Mgr for SELECT mgr from emp where job =
pjob;/n");
lstrcat( (char *) pSqlStmt, " END IF;/n");
lstrcat( (char *) pSqlStmt, " END;/n");
lstrcat( (char *) pSqlStmt, "END;/n");
rc = SQLExecDirect(hStmt, pSqlStmt, lstrlen((char *)pSqlStmt));
if (rc != SQL_SUCCESS)
{
    DisplayError(SQL_HANDLE_STMT, hStmt, "SQLExecDirect");
}

/*
 * Bind the Parameter
 */
rc = SQLBindParameter(hStmt,
1,
SQL_PARAM_INPUT,
SQL_C_CHAR,
SQL_CHAR,
JOB_LEN,
0,

```

```
    pSalesMan,
    0,
    &sqlnts);

/*
 * Call the Store Procedure which executes the Result Sets
 */
lstrcpy( (char *) pSqlStmt, "{CALL ODBCRefCur.EmpCurs(?)}");
rc = SQLExecDirect(hStmt, pSqlStmt, lstrlen((char *)pSqlStmt));
if (rc != SQL_SUCCESS)
{
    DisplayError(SQL_HANDLE_STMT, hStmt, "SQLExecDirect");
}

/*
 * Bind the Data
 */
rc = SQLBindCol( hStmt,
1,
SQL_C_CHAR,
Data,
sizeof(Data),
&DataLen);
if (rc != SQL_SUCCESS)
{
    DisplayError(SQL_HANDLE_STMT, hStmt, "SQLBindCol");
}

/*
 * Get the data for Result Set 1
 */
printf( "/nEmployee Names/n/n");
while ( rc == SQL_SUCCESS )
{
    rc = SQLFetch( hStmt );
    if ( rc == SQL_SUCCESS )
    {
        printf("%s/n", Data);
    }
    else
    {
        if (rc != SQL_NO_DATA)
        {
            DisplayError(SQL_HANDLE_STMT, hStmt, "SQLFetch");
        }
    }
}

printf( "/nFirst Result Set - Hit Return to Continue/n");
charptr = gets ((char *)error);
```

```
/*
 * Get the Next Result Set
 */
rc = SQLMoreResults( hStmt );
if (rc != SQL_SUCCESS)
{
    DisplayError(SQL_HANDLE_STMT, hStmt, "SQLMoreResults");
}

/*
 * Get the data for Result Set 2
 */
printf( "/nManagers/n/n");
while ( rc == SQL_SUCCESS )
{
    rc = SQLFetch( hStmt );
    if ( rc == SQL_SUCCESS )
    {
        printf("%s/n", Data);
    }
    else
    {
        if (rc != SQL_NO_DATA)
        {
            DisplayError(SQL_HANDLE_STMT, hStmt, "SQLFetch");
        }
    }
}

printf( "/nSecond Result Set - Hit Return to Continue/n");
charptr = gets ((char *)error);

/*
 * Should Be No More Results Sets
 */
rc = SQLMoreResults( hStmt );
if (rc != SQL_NO_DATA)
{
    DisplayError(SQL_HANDLE_STMT, hStmt, "SQLMoreResults");
}

/*
 * Drop the Package
 */
lstrcpy( (char *) pSqlStmt, "DROP PACKAGE ODBCRefCur");
rc = SQLExecDirect(hStmt, pSqlStmt, lstrlen((char *)pSqlStmt));

/*
 * Free handles close connections to the database
```

```
/*
SQLFreeHandle( SQL_HANDLE_STMT, hStmt );
SQLDisconnect( hDbc );
SQLFreeHandle( SQL_HANDLE_DBC, hDbc );
SQLFreeHandle( SQL_HANDLE_ENV, hEnv );

printf( "/nAll Done - Hit Return to Exit/n");
charptr = gets ((char *)error);
return(0);
}

/*
* Display Error Messages
*/
void DisplayError( SWORD HandleType, SQLHANDLE hHandle, char *Module )
{

SQLCHAR MessageText[255];
SQLCHAR SQLState[80];
SQLRETURN rc=SQL_SUCCESS;
LONG NativeError;
SWORD RetLen;
SQLCHAR error[25];
char *charptr;

rc = SQLGetDiagRec(HandleType,
hHandle,
1,
SQLState,
&NativeError,
MessageText,
255,
&RetLen);

printf( "Failure Calling %s/n", Module );
if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)
{
printf( "/t/t/t State: %s/n", SQLState);
printf( "/t/t/t Native Error: %d/n", NativeError );
printf( "/t/t/t Error Message: %s/n", MessageText );
}

printf( "/nHit Return to Exit/n");
charptr = gets ((char *)error);

exit(1);
}
```

21.4.8 Enabling EXEC Syntax

If the syntax of your SQL Server `EXEC` statement can be readily translated to an equivalent Oracle procedure call without change, the Oracle ODBC Driver can translate it if you enable this option.

The complete name of a SQL Server procedure consists of up to four identifiers:

- server name
- database name
- owner name
- procedure name

The format for the name is:

```
[[[server.] [database].] [owner_name].]procedure_name
```

During the migration of the SQL Server database to Oracle, the definition of each SQL Server procedure (or function) is converted to its equivalent Oracle syntax and is defined in a schema in Oracle. Migrated procedures are often reorganized (and created in schemas) in one of these ways:

- All procedures are migrated to one schema (the default option).
- All procedures defined in one SQL Server database are migrated to the schema named with that database name.
- All procedures owned by one user are migrated to the schema named with that user's name.

To support these three ways of organizing migrated procedures, you can specify one of these schema name options for translating procedure names. Object names in the translated Oracle procedure call are not case-sensitive.

21.4.9 Enabling Event Notification for Connection Failures in an Oracle RAC Environment

If the `SQL_ORCLATTR_FAILOVER_CALLBACK` and `SQL_ORCLATTR_FAILOVER_HANDLE` attributes of the `SQLSetConnectAttr` function are set when a connection failure occurs in an Oracle Real Application Clusters (Oracle RAC) Database environment, event notification is enabled. Both attributes are set using the `SQLSetConnectAttr` function. The symbols for the new attributes are defined in the file `sqora.h`.

The `SQL_ORCLATTR_FAILOVER_CALLBACK` attribute specifies the address of a routine to call when a failure event takes place.

The `SQL_ORCLATTR_FAILOVER_HANDLE` attribute specifies a context handle that is passed as a parameter in the callback routine. This attribute is necessary for the ODBC application to determine which connection the failure event is taking place on.

The function prototype for the callback routine is:

```
void failover_callback(void *handle, SQLINTEGER fo_code)
```

The 'handle' parameter is the value that was set by the `SQL_ORCLATTR_FAILOVER_HANDLE` attribute. Null is returned if the attribute has not been set.

The `fo_code` parameter identifies the failure event which is taking place. The failure events map directly to the events defined in the OCI programming interface. The list of possible events is:

- ODBC_FO_BEGIN
- ODBC_FO_ERROR
- ODBC_FO_ABORT
- ODBC_FO_REAUTH
- ODBC_FO_END

The following is a sample program which demonstrates using this feature:

```
/*
  NAME
  ODBC Callback Test

  DESCRIPTION
  Simple program to demonstrate the connection failover callback feature.

  PUBLIC FUNCTION(S)
  main

  PRIVATE FUNCTION(S)

  NOTES

  Command Line: ODBC Callback Test filename [odbc-driver]

*/
#include <windows.h>
#include <tchar.h>
#include <malloc.h>
#include <stdio.h>
#include <string.h>
#include <sql.h>
#include <sqlext.h>
#include "sqora.h"

/*
 ** Function Prototypes
 */
void display_errors(SQLSMALLINT HandleType, SQLHANDLE Handle);
void failover_callback(void *Handle, SQLINTEGER fo_code);

/*
 ** Macros
 */
#define ODBC_STS_CHECK(sts) \
    if (sts != SQL_SUCCESS) \
{ \
    display_errors(SQL_HANDLE_ENV, hEnv); \
    display_errors(SQL_HANDLE_DBC, hDbc); \
    display_errors(SQL_HANDLE_STMT, hStmt); \
    return FALSE; \
}

/*
 ** ODBC Handles
 */
SQLHENV *hEnv = NULL; // ODBC Environment Handle
SQLHANDLE *hDbc = NULL; // ODBC Connection Handle
```

```
SQLHANDLE *hStmt = NULL; // ODBC Statement Handle

/*
** Connection Information
*/
TCHAR *dsn = _T("odbctest");
TCHAR *uid = _T("scott");
TCHAR *pwd = _T("tiger");
TCHAR *szSelect = _T("select * from emp");

/*
** MAIN Routine
*/
main(int argc, char **argv)
{
    SQLRETURN rc;

    /*
    ** Allocate handles
    */
    rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, (SQLHANDLE *)&hEnv);
    ODBC_STS_CHECK(rc);

    rc = SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3, 0);
    ODBC_STS_CHECK(rc);

    rc = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, (SQLHANDLE *)&hDbc);
    ODBC_STS_CHECK(rc);

    /*
    ** Connect to the database
    */
    rc = SQLConnect(hDbc, dsn, (SQLSMALLINT)_tcslen(dsn),
                    uid, (SQLSMALLINT)_tcslen(uid),
                    pwd, (SQLSMALLINT)_tcslen(pwd));
    ODBC_STS_CHECK(rc);

    /*
    ** Set the connection failover attributes
    */
    rc = SQLSetConnectAttr(hDbc, SQL_ORCLATTR_FAILOVER_CALLBACK, &failover_callback,
0);
    ODBC_STS_CHECK(rc);

    rc = SQLSetConnectAttr(hDbc, SQL_ORCLATTR_FAILOVER_HANDLE, hDbc, 0);
    ODBC_STS_CHECK(rc);

    /*
    ** Allocate the statement handle
    */
    rc = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, (SQLHANDLE *)&hStmt);
    ODBC_STS_CHECK(rc);

    /*
    ** Wait for connection failovers
    */
    while (TRUE)
    {
        Sleep(5000);
```

```

rc = SQLExecDirect(hStmt, szSelect, _tcslen(szSelect));
ODBC_STS_CHECK(rc);

rc = SQLFreeStmt(hStmt, SQL_CLOSE);
ODBC_STS_CHECK(rc);
}

/*
** Free up the handles and close the connection
*/
rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
ODBC_STS_CHECK(rc);

rc = SQLDisconnect(hDbc);
ODBC_STS_CHECK(rc);

rc = SQLFreeHandle(SQL_HANDLE_DBC, hDbc);
ODBC_STS_CHECK(rc);

rc = SQLFreeHandle(SQL_HANDLE_ENV, hEnv);
ODBC_STS_CHECK(rc);

return TRUE;
}

/*
** Failover Callback Routine
*/
void failover_callback(void *Handle, SQLINTEGER fo_code)
{
    switch (fo_code) {

    case ODBC_FO_BEGIN:
printf("ODBC_FO_BEGIN recevied\n");
break;

    case ODBC_FO_ERROR:
printf("ODBC_FO_ERROR recevied\n");
break;

    case ODBC_FO_ABORT:
printf("ODBC_FO_ABORT recevied\n");
break;

    case ODBC_FO_REAUTH:
printf("ODBC_FO_REAUTH recevied\n");
break;

    case ODBC_FO_END:
printf("ODBC_FO_END recevied\n");
break;

    default:
printf("Invalid or unknown ODBC failover code recevied\n");
break;
    };
}

return;

```

```

}

/*
** Retrieve the errors associated with the handle passed
** and display them.
*/
void display_errors(SQLSMALLINT HandleType, SQLHANDLE Handle)
{
    SQLTCHAR MessageText[256];
    SQLTCHAR SqlState[5+1];
    SQLSMALLINT i=1;
    SQLINTEGER NativeError;
    SQLSMALLINT TextLength;
    SQLRETURN sts = SQL_SUCCESS;

    if (Handle == NULL) return;

    /* Make sure all SQLState text is null terminated */
    SqlState[5] = '\0';

    /*
    ** Fetch and display all diagnostic records that exist for this handle
    */
    while (sts == SQL_SUCCESS)
    {
        NativeError = 0;
        TextLength = 0;
        sts = SQLGetDiagRec(HandleType, Handle, i, SqlState, &NativeError,
        (SQLTCHAR *)&MessageText, sizeof(MessageText),
        &TextLength);

        if (sts == SQL_SUCCESS)
        {
            printf("[%s]%s\n", SqlState, MessageText);
            if (NativeError != 0)
            {
                printf("Native Error Code: %d\n", NativeError);
            }
            i++;
        }
    }

    return;
}

```

21.4.10 Using Implicit Results Feature Through ODBC

Use this option when you migrate any third party ODBC application to Oracle Database and you want to use implicit results functionality as supported by the previous vendor. Oracle ODBC driver supports implicit results with stored procedures or an anonymous PL/SQL block. For the current release, implicit results are returned only for SELECT statements.

The following code example shows an example ODBC test case using an anonymous SQL script for implicit results.

```

const char *query1="declare \
    c1 sys_refcursor; \
    c2 sys_refcursor; \
    begin \

```

```

        open c1 for select empno,ename from emp where rownum<=3; \
        dbms_sql.return_result(c1); \
        open c2 for select empno,ename from emp where rownum<=3; \
        dbms_sql.return_result(c2); end; ";

int main( )
{
    ...
    ...
    //Allocate all required handles and establish a connection to the database.

    //Prepare and execute the above anonymous PL/SQL block
    SQLPrepare (hstmt, (SQLCHAR *) query1, SQL_NTS);
    SQLExecute(hstmt);

    //Bind the columns for the results from the first SELECT statement in an
    //anonymous block.
    SQLBindCol (hstmt, 1, SQL_C_ULONG, &eno, 0, &jind);
    SQLBindCol (hstmt, 2, SQL_C_CHAR, empname, sizeof (empname),&enind);

    //Fetch implicit results through the SQLFetch( ) call.
    while((retCode = SQLFetch(hstmt)) != SQL_NO_DATA)
    {
        //Do whatever you want to do with the data.
    }

    retCode = SQLMoreResults(hstmt);

    if(retCode == SQL_SUCCESS)
    {
        printf("SQLMoreResults returned with SQL_SUCCESS\n");

        //Bind the columns for the results from the second SELECT statement in an
        //anonymous block.
        SQLBindCol (hstmt, 1, SQL_C_ULONG, &eno, 0, &jind);
        SQLBindCol (hstmt, 2, SQL_C_CHAR, empname, sizeof (empname),&enind);

        //Fetch implicit results through the SQLFetch( ) call.
        while((retCode = SQLFetch(hstmt)) != SQL_NO_DATA)
        {
            //Do whatever you want to do with data.
        }
    }
}

```

21.4.11 Supported Functionality

Topics:

- [API Conformance](#)
- [Implementation of ODBC API Functions](#)
- [Implementation of the ODBC SQL Syntax](#)
- [Implementation of Data Types \(Programming\)](#)

21.4.11.1 API Conformance

Oracle ODBC Driver release 9.2.0.0.0 and higher supports all Core, Level 2, and Level 1 functions.

Also, Oracle ODBC Driver release 9.2.0.0.0 and higher supports translation DLLs.

The following topics describe the ODBC API functions implemented by the Oracle ODBC Driver.

Related Topic for Advanced Users

[Error Messages](#)

Related Topic for Programmers

[Implementation of ODBC API Functions](#)

21.4.11.2 Implementation of ODBC API Functions

[Table 21–7](#) describes how the Oracle ODBC Driver implements specific functions:

Table 21–7 How Oracle ODBC Driver Implements Specific Functions

Function	Description
SQLConnect	SQLConnect requires only a DBQ, user ID, and password.
SQLDriverConnect	SQLDriverConnect uses the DSN, DBQ, UID, and PWD keywords.
SQLMoreResults	Implements ODBC support for implicit results. This is a new API implemented for Oracle Database 12c Release 1 (12.1.0.1). See http://msdn.microsoft.com/en-us/library/ms714673(v=VS.85).aspx for more information.
SQLSpecialColumns	If SQLSpecialColumns is called with the SQL_BEST_ROWID attribute, it returns the rowid column.
SQLProcedures and SQLProcedureColumns	See the information that follows.
All catalog functions	If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, a string argument is treated as an identifier argument, and its case is not significant. In this case, the underscore ("_") and the percent sign ("%") are treated as the actual character, not as a search pattern character. On the other hand, if this attribute is SQL_FALSE, it is either an ordinary argument or a pattern value argument and is treated literally, and its case is significant.

21.4.11.3 Implementation of the ODBC SQL Syntax

If a comparison predicate has a parameter marker as the second expression in the comparison and the value of that parameter is SQL_NULL_DATA with SQLBindParameter, the comparison fails. This is consistent with the null predicate syntax in ODBC SQL.

21.4.11.4 Implementation of Data Types (Programming)

For programmers, the noteworthy part of the implementation of the data types concerns the CHAR, VARCHAR, and VARCHAR2 data types.

For an fSqlType value of SQL_VARCHAR, SQLGetTypeInfo returns the Oracle database data type VARCHAR2. For an fSqlType value of SQL_CHAR, SQLGetTypeInfo returns the Oracle database data type CHAR.

21.4.12 Unicode Support

Topics:

- [Unicode Support Within the ODBC Environment](#)
- [Unicode Support in ODBC API](#)
- [Unicode Functions in the Driver Manager](#)
- [SQLGetData Performance](#)
- [Unicode Samples](#)

21.4.12.1 Unicode Support Within the ODBC Environment

The Microsoft or unixODBC ODBC Driver Manager (Driver Manager) makes all ODBC drivers, regardless if they support Unicode, appear as if they are Unicode compliant. This allows ODBC applications to be written independent of the Unicode capabilities of underlying ODBC drivers.

The extent to which the Driver Manager can emulate Unicode support for ANSI ODBC drivers is limited by the conversions possible between the Unicode data and the local code page. Data loss is possible when the Driver Manager is converting from Unicode to the local code page. Full Unicode support is not possible unless the underlying ODBC driver supports Unicode. The Oracle ODBC Driver provides full Unicode support.

21.4.12.2 Unicode Support in ODBC API

The ODBC API supports both Unicode and ANSI entry points using the "W" and "A" suffix convention. An ODBC application developer need not explicitly call entry points with the suffix. An ODBC application that is compiled with the `UNICODE` and `_UNICODE` preprocessor definitions generates the appropriate calls. For example, a call to `SQLPrepare` is compiled as `SQLPrepareW`.

The C data type, `SQL_C_WCHAR`, was added to the ODBC interface to allow applications to specify that an input parameter is encoded as Unicode or to request column data returned as Unicode. The macro `SQL_C_TCHAR` is useful for applications that must be built as both Unicode and ANSI. The `SQL_C_TCHAR` macro compiles as `SQL_C_WCHAR` for Unicode applications and as `SQL_C_CHAR` for ANSI applications.

The SQL data types, `SQL_WCHAR`, `SQL_WVARCHAR`, and `SQL_WLONGVARCHAR`, have been added to the ODBC interface to represent columns defined in a table as Unicode. Potentially, these values are returned from calls to `SQLDescribeCol`, `SQLColAttribute`, `SQLColumns`, and `SQLProcedureColumns`.

Unicode encoding is supported for SQL column types `NCHAR`, `NVARCHAR2`, and `NCLOB`. Also, Unicode encoding is also supported for SQL column types `CHAR` and `VARCHAR2` if the character semantics are specified in the column definition.

The ODBC Driver supports these SQL column types and maps them to ODBC SQL data types.

[Table 21-8](#) lists the supported SQL data types and the equivalent ODBC SQL data type.

Table 21-8 Supported SQL Data Types and the Equivalent ODBC SQL Data Type

SQL Data Types	ODBC SQL Data Types
<code>CHAR</code>	<code>SQL_CHAR</code> or <code>SQL_WCHAR</code> ¹

Table 21–8 (Cont.) Supported SQL Data Types and the Equivalent ODBC SQL Data Type

SQL Data Types	ODBC SQL Data Types
VARCHAR2	SQL_VARCHAR or SQL_WVARCHAR ²
NCHAR	SQL_WCHAR
NVARCHAR2	SQL_WVARCHAR
NCLOB	SQL_WLONGVARCHAR

¹ CHAR maps to SQL_WCHAR if the character semantics were specified in the column definition and if the character set for the database is Unicode.

² VARCHAR2 maps to SQL_WVARCHAR if the character semantics were specified in the column definition and if the character set for the database is Unicode.

21.4.12.3 Unicode Functions in the Driver Manager

The Driver Manager performs the following functions when it detects that the underlying ODBC driver does not support Unicode:

- Convert Unicode function calls to ANSI function calls before calling the ANSI ODBC driver. String arguments are converted from Unicode to the local code page. For example, a call to SQLPrepareW is converted to call SQLPrepare. The text of the SQL statement parameter is converted from Unicode to the local code page.
- Convert return parameters that are character data from the local code page to Unicode. For example, returning the column name through SQLColAttribute.
- Convert data from the local code page to Unicode for columns bound as SQL_C_WCHAR.
- Convert data from Unicode to the local code page for input parameters bound as SQL_C_WCHAR.

21.4.12.4 SQLGetData Performance

The SQLGetData function allows an ODBC application to specify the data type to receive a column as after the data has been fetched. OCI requires the Oracle ODBC Driver to specify the data type before it is fetched. In this case, the Oracle ODBC Driver uses the knowledge it has about the data type of the column as defined in the database to determine how to best default to fetching the column through OCI.

If a column that contains character data is not bound by SQLBindCol, the Oracle ODBC Driver must determine if it must fetch the column as Unicode or as the local code page. The driver could default to receiving the column as Unicode, however, this may result in as many as two unnecessary conversions. For example, if the data were encoded in the database as ANSI, there would be an ANSI to Unicode conversion to fetch the data into the Oracle ODBC Driver. If the ODBC application then requested the data as SQL_C_CHAR, there would be an additional conversion to revert the data back to its original encoding.

The default encoding of the Oracle client is used when fetching data. However, an ODBC application can overwrite this default and fetch the data as Unicode by binding the column or the parameter as the WCHAR data type.

21.4.12.5 Unicode Samples

As the Oracle ODBC Driver itself was implemented using TCHAR macros, Oracle recommends that ODBC application programs use TCHAR to take advantage of the driver.

The following links are program examples showing how to use TCHAR, which becomes the WCHAR data type in case you compile with `UNICODE` and `_UNICODE`.

- [Example 1: Connection to Database](#)
- [Example 2: Simple Retrieval](#)
- [Example 3: Retrieval Using SQLGetData \(Binding After Fetch\)](#)
- [Example 4: Simple Update](#)
- [Example 5: Update and Retrieval for Long Data \(CLOB\)](#)

Example 1: Connection to Database

No difference other than specifying Unicode literals for `SQLConnect`.

```
HENV envHnd;
HDBC conHnd
;
HSTMT stmtHnd;
RETCODE rc;

rc = SQL_SUCCESS;

// ENV is allocated
rc = SQLAllocEnv(&envHnd);
// Connection Handle is allocated
rc = SQLAllocConnect(envHnd, &conHnd);
rc = SQLConnect(conHnd, _T("stpc19"), SQL_NTS, _T("scott"), SQL_NTS, _T("tiger"),
SQL_NTS);
.
.
.
if (conHnd)
SQLFreeConnect(conHnd);
if (envHnd)
SQLFreeEnv(envHnd);
```

Example 2: Simple Retrieval

The following example retrieves the employee names and the job titles from the `EMP` table. With the exception that you must specify `TCHAR` compliant data to every `ODBC` function, there is no difference to the `ANSI` case. If the case is a `Unicode` application, you have to specify the length of the buffer to the `BYTE` length when you call `SQLBindCol` (for example, `sizeof(ename)`).

```
/*
** Execute SQL, bind columns, and Fetch.
** Procedure:
**
** SQLExecDirect
** SQLBindCol
** SQLFetch
**
*/
static SQLTCHAR *sqlStmt = _T("SELECT ename, job FROM emp");
SQLTCHAR ename[50];
SQLTCHAR job[50];
SQLINTEGER enamelen, joblen;
```

```

_tprintf(_T("Retrieve ENAME and JOB using SQLBindCol 1.../n[%s]/n"), sqlStmt);

// Step 1: Prepare and Execute
rc = SQLExecDirect(stmtHnd, sqlStmt, SQL_NTS); // select
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

// Step 2: Bind Columns
rc = SQLBindCol(stmtHnd,
1,
SQL_C_TCHAR,
ename,
sizeof(ename),
&enameLen);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

rc = SQLBindCol(stmtHnd,
2,
SQL_C_TCHAR,
job,
sizeof(job),
&jobLen);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

do
{
// Step 3: Fetch Data
rc = SQLFetch(stmtHnd);
if (rc == SQL_NO_DATA)
break;
checkSQLErr(envHnd, conHnd, stmtHnd, rc);
_tprintf(_T("ENAME = %s, JOB = %s/n"), ename, job);
} while (1);
_tprintf(_T("Finished Retrieval/n/n"));

```

Example 3: Retrieval Using SQLGetData (Binding After Fetch)

This example shows how to use `SQLGetData`. For those who are not familiar with ODBC programming, the fetch is allowed before binding the data using `SQLGetData`, unlike in an OCI program. There is no difference to the ANSI application in terms of Unicode-specific issues.

```

/*
** Execute SQL, bind columns, and Fetch.
** Procedure:
**
** SQLExecDirect
** SQLFetch
** SQLGetData
*/
static SQLTCHAR *sqlStmt = _T("SELECT ename,job FROM emp"); // same as Case 1.
SQLTCHAR ename[50];
SQLTCHAR job[50];

_tprintf(_T("Retrieve ENAME and JOB using SQLGetData.../n[%s]/n"), sqlStmt);
if (rc != SQL_SUCCESS)
{
_tprintf(_T("Failed to allocate STMT/n"));
goto exit2;
}

```

```

// Step 1: Prepare and Execute
rc = SQLExecDirect(stmtHnd, sqlStmt, SQL_NTS); // select
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

do
{
    // Step 2: Fetch
    rc = SQLFetch(stmtHnd);
    if (rc == SQL_NO_DATA)
        break;

    checkSQLErr(envHnd, conHnd, stmtHnd, rc);

    // Step 3: GetData
    rc = SQLGetData(st
        mtHnd,
        1,
        SQL_C_TCHAR,
        (SQLPOINTER)ename,
        sizeof(ename),
        NULL);
    checkSQLErr(envHnd, conHnd, stmtHnd, rc);
    rc = SQLGetData(stmtHnd,
        2,
        SQL_C_TCHAR,
        (SQLPOINTER)job,
        sizeof(job),
        NULL);
    checkSQLErr(envHnd, conHnd, stmtHnd, rc);
    _tprintf(_T("ENAME = %s, JOB = %s/n"), ename, job);
} while (1);
_tprintf(_T("Finished Retrieval/n/n"));

```

Example 4: Simple Update

This example shows how to update data. Likewise, the length of data for SQLBindParameter has to be specified with the `BYTE` length, even in the case of a Unicode application.

```

/
*
** Execute SQL, bind columns, and Fetch.
** Procedure:
**
** SQLPrepare
** SQLBindParameter
** SQLExecute
*/
static SQLCHAR *sqlStmt = _T("INSERT INTO emp(empno,ename,job) VALUES(?, ?, ?)");
static SQLCHAR *empno = _T("9876"); // Emp No
static SQLCHAR *ename = _T("ORACLE"); // Name
static SQLCHAR *job = _T("PRESIDENT"); // Job

_tprintf(_T("Insert User ORACLE using SQLBindParameter.../n[%s]/n"), sqlStmt);

// Step 1: Prepar
rc = SQLPrepare(stmtHnd, sqlStmt, SQL_NTS); // select

```

```

checkSQLErr(envHnd, conHnd, stmtHnd, rc);

// Step 2: Bind Parameter
rc = SQLBindParameter(stmtHnd,
1,
SQL_PARAM_INPUT,
SQL_C_TCHAR,
SQL_DECIMAL,
4, // 4 digit
0,
(SQLPOINTER)empno,
0,
NULL);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

rc = SQLBindParameter(stmtHnd,
2,
SQL_PARAM_INPUT,
SQL_C_TCHAR,
SQL_CHAR,
1strlen(ename)*sizeof(TCHAR),
0,
(SQLPOINTER)ename,
1strlen(ename)*sizeof(TCHAR),
NULL);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

rc = SQLBindParameter(stmtHnd,
3,
SQL_PARAM_INPUT,
SQL_C_TCHAR,
SQL_CHAR,
1strlen(job)*sizeof(TCHAR),
0,
(SQLPOINTER)job,
1strlen(job)*sizeof(TCHAR),
NULL);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

// Step 3: Execute
rc = SQLExecute(stmtHnd);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

```

Example 5: Update and Retrieval for Long Data (CLOB)

This example may be the most complicated case to update and retrieve data for long data, like CLOB, in Oracle. Because the length of data must be the BYTE length, `1strlen(TCHAR data)*sizeof(TCHAR)` is needed to derive the BYTE length.

```

/*
** Execute SQL, bind columns, and Fetch.
** Procedure:
**
** SQLPrepare
** SQLBindParameter
** SQLExecute
** SQLParamData
** SQLPutData
**
** SQLExecDirect

```

```

** SQLFetch
** SQLGetData
*/
static SQLTCHAR *sqlStmt1 = _T("INSERT INTO clobtbl(clob1) VALUES(?)");
static SQLTCHAR *sqlStmt2 = _T("SELECT clob1 FROM clobtbl");
SQLTCHAR clobdata[1001];
SQLTCHAR resultdata[1001];
SQLINTEGER ind = SQL_DATA_AT_EXEC;
SQLTCHAR *bufp;
int clobdatalen, chunksize, dtsize, retchklen;

_tprintf(_T("Insert CLOB1 using SQLPutData.../n[%s]/n"), sqlStmt1);

// Set CLOB Data
{
int i;
SQLTCHAR ch;
for (i=0, ch=_T('A'); i< sizeof(clobdata)/sizeof(SQLTCHAR); ++i, ++ch)
{
if (ch > _T('Z'))
ch = _T('A');
clobdata[i] = ch;
}
clobdata[sizeof(clobdata)/sizeof(SQLTCHAR)-1] = _T('/0');
}
clobdatalen = lstrlen(clobdata); // length of characters
chunksize = clobdatalen / 7; // 7 times to put

// Step 1: Prepare
rc = SQLPrepare(stmtHnd, sqlStmt1, SQL_NTS);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

// Step 2: Bind Parameter with SQL_DATA_AT_EXEC
rc = SQLBindParameter(stmtHnd,
1,
SQL_PARAM_INPUT,
SQL_C_TCHAR,
SQL_LONGVARCHAR,
clobdatalen*sizeof(TCHAR),
0,
(SQLPOINTER)clobdata,
clobdatalen*sizeof(TCHAR),
&ind);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

// Step 3: Execute
rc = SQLEexecute(stmtHnd);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

// Step 4: ParamData (initiation)
rc = SQLParamData(stmtHnd, (SQLPOINTER*)&bufp); // set value
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

for (dtsize=0, bufp = clobdata;
dtsize < clobdatalen;
dtsize += chunksize, bufp += chunksize)
{
int len;
if (dtsize+chunksize<clobdatalen)
len = chunksize;

```

```
else
len = clobdatalen-dtsize;

// Step 5: PutData
rc = SQLPutData(stmtHnd, (SQLPOINTER)bufp, len*sizeof(TCHAR));
checkSQLErr(envHnd, conHnd, stmtHnd, rc);
}

// Step 6: ParamData (termination)
rc = SQLParamData(stmtHnd, (SQLPOINTER*)&bufp);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

rc = SQLFreeStmt(stmtHnd, SQL_CLOSE);
_tprintf(_T("Finished Update/n/n"));
rc = SQLAllocStmt(conHnd, &stmtHnd);
if (rc != SQL_SUCCESS)
{
_tprintf(_T("Failed to allocate STMT/n"));
goto exit2;
}

// Clear Result Data
memset(resultdata, 0, sizeof(resultdata));
chunksize = clobdatalen / 15; // 15 times to put

// Step 1: Prepare
rc = SQLExecDirect(stmtHnd, sqlStmt2, SQL_NTS); // select
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

// Step 2: Fetch
rc = SQLFetch(stmtHnd);
checkSQLErr(envHnd, conHnd, stmtHnd, rc);

for(dtsize=0, bufp = resultdata;
dtsize < sizeof(resultdata)/sizeof(TCHAR) && rc != SQL_NO_DATA;
dtsize += chunksize-1, bufp += chunksize-1)
{
int len; // len should contain the space for NULL termination
if (dtsize+chunksize<sizeof(resultdata)/sizeof(TCHAR))
len = chunksize;
else
len = sizeof(resultdata)/sizeof(TCHAR)-dtsize;

// Step 3: GetData
rc = SQLGetData(stmtHnd,
1,
SQL_C_TCHAR,
(SQLPOINTER)bufp,
len*sizeof(TCHAR),
&retchklen);
}
if (!_tcscmp(resultdata, clobdata))
{
_tprintf(_T("Succeeded!!/n/n"));
}
else
{
_tprintf(_T("Failed!!/n/n"));
}
```

21.4.13 Performance and Tuning

Topics:

- [General ODBC Programming Tips](#)
- [Data Source Configuration Options](#)
- [DATE and TIMESTAMP Data Types](#)

21.4.13.1 General ODBC Programming Tips

This section describes some general programming tips to improve the performance of an ODBC application.

- Enable connection pooling if the application will frequently connect and disconnect from a data source. Reusing pooled connections is extremely efficient compared to reestablishing a connection.
- Minimize the number of times a statement must be prepared. Where possible, use bind parameters to make a statement reusable for different parameter values. Preparing a statement once and executing it several times is much more efficient than preparing the statement for every `SQLExecute`.
- Do not include columns in a `SELECT` statement if you know the application will not retrieve them; especially `LONG` columns. Due to the nature of the database server protocols, the ODBC Driver must fetch the entire contents of a `LONG` column if it is included in the `SELECT` statement, regardless if the application binds the column or does a `SQLGetData`.
- If you are performing transactions that do not update the data source, set the `SQL_ATTR_ACCESS_MODE` attribute of the ODBC `SQLSetConnectAttr` function to `SQL_MODE_READ_ONLY`.
- If you are not using ODBC escape clauses, set the `SQL_ATTR_NOSCAN` attribute of the ODBC `SQLSetConnectAttr` function or the ODBC `SQLSetStmtAttr` function to true.
- Use the ODBC `SQLFetchScroll` function instead of the ODBC `SQLFetch` function for retrieving data from tables that have a large number of rows.
- Enable OCI statement caching when the same SQL statements are used multiple times (`StatementCache=T`).
- Binding `NUMBER` columns as `FLOAT` speeds up query execution (`BindAsFLOAT=T`).
- While fetching `LONG` or `LONG RAW` set `MaxLargeData=<value>` for optimum performance.
- Setting `UseOCIDescribeAny=T` for applications making heavy calls to small packaged procedures that return `Ref Cursor` improves performance.

21.4.13.2 Data Source Configuration Options

This topic discusses performance implications of the following ODBC data source configuration options:

Topics:

- [Enable Result Sets](#)
- [Enable LOBs](#)
- [Bind TIMESTAMP as DATE](#)

- [Enable Closing Cursors](#)
- [Enable Thread Safety](#)
- [Fetch Buffer Size](#)

Enable Result Sets

This option enables the support of returning result sets (for example, `RefCursor`) from procedure calls. The default is enabling the returning of result sets.

The ODBC Driver must query the database server to determine the set of parameters for a procedure and their data types to determine if there are any `RefCursor` parameters. This query incurs an additional network round trip the first time any procedure is prepared and executed.

Enable LOBs

This option enables the support of inserting and updating LOBs. The default is enabled.

The ODBC Driver must query the database server to determine the data types of each parameter in an `INSERT` or `UPDATE` statement to determine if there are any LOB parameters. This query incurs an additional network round trip the first time any `INSERT` or `UPDATE` is prepared and executed.

Bind TIMESTAMP as DATE

Binds `SQL_TIMESTAMP` parameters as the appropriate Oracle data type. If this option is `TRUE`, `SQL_TIMESTAMP` binds as the Oracle `DATE` data type. If this option is `FALSE`, `SQL_TIMESTAMP` binds as the Oracle `TIMESTAMP` data type (which is the default).

Enable Closing Cursors

The `SQL_CLOSE` option of the ODBC function, `SQLFreeStmt`, is supposed to close associated cursors with a statement and discard all pending results. The application can reopen the cursor by executing the statement again without doing a `SQLPrepare` again. A typical scenario for this is an application that is idle for a while but reuses the same SQL statement. While the application is idle, it might free up associated server resources.

The Oracle Call Interface (OCI), on which the Oracle ODBC Driver is layered, does not support the functionality of closing cursors. So, by default, the `SQL_CLOSE` option has no effect in the Oracle ODBC Driver. The cursor and associated resources remain open on the database server.

Enabling this option causes the associated cursor to be closed on the database server. However, this results in the parse context of the SQL statement being lost. The ODBC application can execute the statement again without calling `SQLPrepare`. However, internally the ODBC Driver must prepare and execute the statement all over. Enabling this option severely impacts performance of applications that prepare a statement once and execute it repeatedly.

Enable this option only if freeing the resources on the server is absolutely necessary.

Enable Thread Safety

If an application is single-threaded, this option can be disabled. By default, the ODBC Driver ensures that access to all internal structures (environment, connection, statement) are thread-safe. Single-threaded applications can eliminate some of the thread safety overhead by disabling this option. Disabling this option typically shows a minor performance improvement.

Fetch Buffer Size

Set the Fetch Buffer Size in the [Oracle Options](#) tab of the [Oracle ODBC Driver Configuration Dialog Box](#) to a value specified in bytes. This value determines how many rows of data at a time the ODBC Driver prefetches from an Oracle database to the client's cache, regardless of the number of rows the application program requests in a single query, thus improving performance.

Applications that typically fetch fewer than 20 rows of data at a time improve their response time, particularly over slow network connections or to heavily loaded servers. Setting this too high can worsen response time or consume large amounts of memory. The default is 64,000 bytes. Choose a value that works best for your application.

Note: When LONG and LOB data types are present, the number of rows prefetched by the ODBC Driver is not determined by the Fetch Buffer Size. The inclusion of the LONG and LOB data types minimizes the performance improvement and could result in excessive memory use. The ODBC Driver disregards the Fetch Buffer Size and prefetch a set number of rows in the presence of the LONG and LOB data types.

21.4.13.3 DATE and TIMESTAMP Data Types

If a DATE column in the database is used in a WHERE clause and the column has an index, there can be an impact on performance. For example:

```
SELECT * FROM EMP WHERE HIREDATE = ?
```

In this example, an index on the HIREDATE column could be used to make the query execute quickly. But, because HIREDATE is actually a DATE value and the ODBC Driver is supplying the parameter value as TIMESTAMP, the Oracle server's query optimizer must apply a conversion function. To prevent incorrect results (as might happen if the parameter value had nonzero fractional seconds), the optimizer applies the conversion to the HIREDATE column resulting in the following statement:

```
SELECT * FROM EMP WHERE TO_TIMESTAMP(HIREDATE) = ?
```

Unfortunately, this has the effect of disabling the use of the index on the HIREDATE column and instead the server performs a sequential scan of the table. If the table has many rows, this can take a long time. As a workaround for this situation, the ODBC Driver has the connection option to Bind TIMESTAMP as DATE. When this option is enabled, the ODBC Driver binds SQL_TIMESTAMP parameters as the Oracle DATE data type instead of the Oracle TIMESTAMP data type. This allows the query optimizer to use any index on the DATE columns.

Note: This option is intended for use only with Microsoft Access or other similar programs that bind DATE columns as TIMESTAMP columns. Do not use this option when there are actual TIMESTAMP columns present or when data loss may occur. Microsoft Access executes such queries using whatever columns are selected as the primary key.

Related Topic

[Implementation of Data Types \(Advanced\)](#)

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 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 Oracle Database with the knowledge to recognize EPC coding schemes, support efficient storage and component level retrieval of EPC data, and 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 allows developers to use 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 and 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 encoding category, registering the encoding type, and then registering the components associated with each encoding type.

Topics:

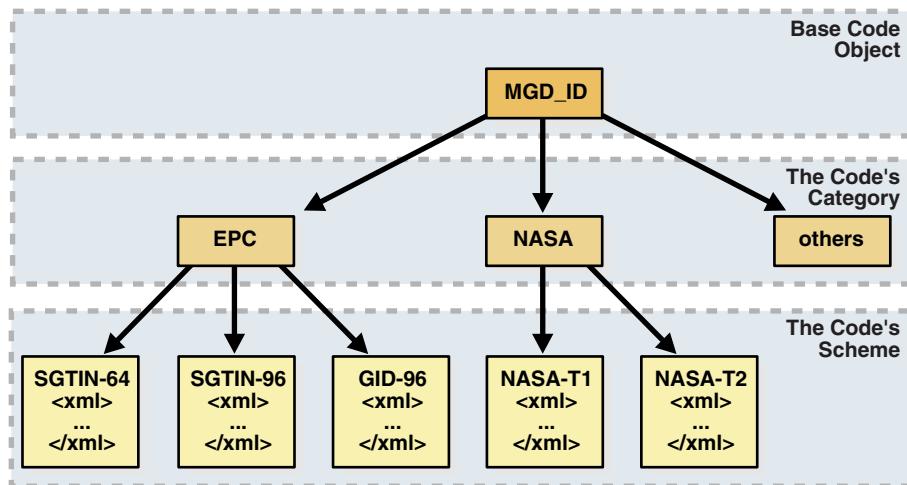
- [Identity Concepts](#)
- [What is the Identity Code Package?](#)
- [Using the Identity Code Package](#)
- [Identity Code Package Types](#)
- [DBMS_MGD_ID_UTL Package](#)
- [Identity Code Metadata Tables and Views](#)
- [Electronic Product Code \(EPC\) Concepts](#)
- [Oracle Database Tag Data Translation Schema](#)

22.1 Identity Concepts

A database object `MGD_ID` is defined that lets users use EPC standard identity codes and use their own existing identity codes. See [Section 22.7](#) 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 on) representing different encoding types (defined in the EPC standard v1.1) is loaded by default into the database. Users can define encoding their own categories and schemes as shown in [Figure 22–1](#) and load these into the database as well.

Figure 22–1 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 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. Also, 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 22–1](#). 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 22–1 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)
SGTIN-64	2	Filter Value (3), Company Prefix Index (14), Item Reference 20), Serial Number (25)

Table 22-1 (Cont.) General Structure of EPC Encodings

Encoding Name	Header Length in bits	Field Names (parameter_list name-value pairs) and (length in bits)
SGTIN-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 on). Each of these schemes has various representations; today, the most often used are BINARY, TAG_URI, and PURE_IDENTITY. For example, information in an SGTIN-64 can be represented in these ways:

```
BINARY: 100110000000000010000111011000100001000001111110011000110010
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.example.com
```

Some representations contain all information about the tag (BINARY and TAG_URI), while other representations contain 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 more 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 metadata.

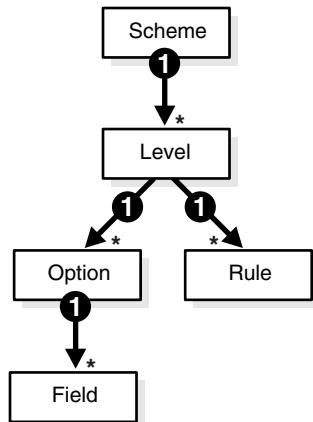
The Oracle Database metadata structure is similar, but not identical to the TDT standard. To fit the EPCglobal TDT specification, the Oracle RFID package must be able to ingest any TDT compatible scheme and seamlessly translate it into the generic Oracle Database defined metadata. See the `EPC_TO_ORACLE` Function in [Table 22-4](#) 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 22-3](#) 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 [Section 22.8](#) for the actual Oracle Database TDT XML schema. Developers can refer to this Oracle Database TDT XML schema to define their own tag structures.

[Figure 22-2](#) shows the Oracle Database Tag Data Translation Markup Language Schema diagram.

Figure 22-2 Oracle Database 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 fields. See the EPCGlobal Tag Translator Specification for more information.

22.2 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 and new and evolving tag encodings that are not included in the current EPC standard.

The Identity Code Package defines these ADTs:

- `MGD_ID` -- defines these (see `MGD_ID` ADT in [Table 22-2](#) 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 ADTs.

[Section 22.3](#) describes how to use these ADTs and member functions.

[Section 22.4](#) and [Section 22.5](#) briefly describe the reference information for these ADTs 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 these topics and files for more information:

- [Section 22.7](#) describes the EPC spec v1.1 product code and its family of coding schemes.
- [Section 22.6](#) 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 these topics for more information:

- [Section 22.3.2](#) describes how to create a function based index or bitmap function based index using the member functions of the `MGD_ID` ADT.

The Identity Code Package provides a utility package that consists of various utility subprograms. See this topic for more information:

- [Section 22.4](#) and [Section 22.5](#) describes each of the member subprograms. A proxy utility sets and removes proxy information. A metadata utility gets a category ID, refreshes a tag scheme for a category, removes a tag scheme for a category, and validates a tag scheme. A conversion utility translates standard EPCglobal Tag Data Translation (TDT) files into Oracle Database TDT files.

The Identity Code Package is extensible and lets you create your own identity code types for your new or evolving RFID tags. You can define your identity code types, `category_id` attribute values, and components structures for your own encoding types. See these topics for more information:

- [Section 22.3.4.1](#) describes how to create your own identity codes by first registering the encoding category, and then registering the schemes associated to the encoding category.
- [Section 22.6](#) 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.

22.3 Using the Identity Code Package

Topics:

- [Storing RFID Tags in Oracle Database Using MGD_ID ADT](#)
- [Building a Function-Based Index Using the Member Functions of the MGD_ID Column Type](#)
- [Using MGD_ID ADT Functions](#)
- [Defining a Category of Identity Codes and Adding Encoding Schemes to an Existing Category](#)

22.3.1 Storing RFID Tags in Oracle Database Using MGD_ID ADT

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](#)

22.3.1.1 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*Plus command:

```
describe warehouse_info;
```

Result:

Name	Null?	Type
CODE	NOT NULL	MGDSYS.MGD_ID
ARRIVAL_TIME		TIMESTAMP(6)
LOCATION		VARCHAR2(256)

22.3.1.2 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 \(if null, then the latest version is used\), 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](#)

22.3.1.2.1 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();

@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')))

.
```

22.3.1.2.2 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();
```

@constructor22.sql

```
MGD_ID('1', MGD_ID_COMPONENT_VARRAY(MGD_ID_COMPONENT('filter', '3'),
    MGD_ID_COMPONENT('schemas', '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')))

.
.
.
```

22.3.1.2.3 Constructing an MGD_ID object (SGTIN-64) and Passing in the Category Name, Category Version (if null, then the latest version is 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('schemas', 'SGTIN-64')
    )
)
) from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();

@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('schemas', 'SGTIN-64')
)
)
```

22.3.1.2.4 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;schema=SGTIN-64') from DUAL;
call DBMS_MGD_ID_UTL.remove_proxy();

@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')
        )
    )
```

22.3.1.3 Inserting an MGD_ID Object into a Database Table

This 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');

COMMIT;
call DBMS_MGD_ID_UTL.remove_proxy();

```

22.3.1.4 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 ADT. 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 [Section 22.3.3.1](#) for more information and see [Table 22–3](#) for a list of member functions.

22.3.2 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'));
```

22.3.3 Using MGD_ID ADT Functions

The MGD_ID ADT contains member subprograms that operate on these ADTs. See [Table 22–2](#) for MGD_ID_COMPONENT, MGD_ID_COMPONENT_VARRAY, MGD_ID ADT reference information. See the `mgdtyp.sql` file for the MGD_ID ADT definition and its member subprograms.

Topics:

- [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](#)

22.3.3.1 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 [Section 22.3.4](#) for more information about how to create a 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 `itemref` as a component, you can use this 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 22-3](#) for a list of other member functions.

22.3.3.2 Parsing Tag Data from Standard Representations

RFID readers read the bit strings stored in the tags. The tag data and other information, such as the reader ID and the time stamp, first go through an edge server to be processed, normalized, and preliminarily filtered. Then, in many application scenarios, the information must be persistently stored and later on be retrieved. The Oracle Database understands the code structures representations of various EPC tags as described in [Table 22-1](#) because these code representation schemes defined in the EPC Standard are preregistered. 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 [Section 22.1](#), each of the EPCGlobal tag schemes (GID-96, SGTIN-64, SGTIN-96, and so on) has various representations with the most often used being `BINARY`, `TAG_URI`, and `PURE_IDENTITY`.

Some representations contain all the information about the tag (`BINARY` and `TAG_URI`), while representations contain 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 more information, namely the filter value.

One `MGD_ID` constructor takes in four fields, the category name (such as `EPC`), the category version, the tag identifier (for `EPC`, the identifier must be in a representation previously described), and a parameter list for any additional parameters required to parse the tag representation. For example, this code creates an `MGD_ID` object from its `BINARY` representation.

```
SELECT MGD_ID
  ('EPC',
  null,
  '10011000000000001000011101100010000100001111110011000110010',
  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')
)
)
)
```

22.3.3.3 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, this 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

```

22.3.3.4 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
-----
100110000000000001000011101100010000100001111110011000110010

```

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.

22.3.4 Defining a Category of Identity Codes and Adding Encoding Schemes to an Existing Category

Topics:

- [Creating a Category of Identity Codes](#)
- [Adding Two Metadata Schemes to a Newly Created Category](#)

22.3.4.1 Creating a Category of Identity Codes

Because the EPCglobal TDT standard is powerful and highly extensible, the Oracle Database RFID standard metadata is a close relative of the TDT specification. Thus, the Identity Code package is extensible: You can create your own categories and tag structures using generic metadata. To create a category of identity codes, use the DBMS_MGD_ID_UTIL.create_category function.

For example, suppose you want to create a category called MGD_SAMPLE_CATEGORY, which has two types of tags, a CONTRACTOR_TAG and an EMPLOYEE_TAG. This category and its two metadata schemes might be used within a company that must 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. This script creates a 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 [Section 22.3.4.2](#) for an example.

22.3.4.2 Adding Two Metadata Schemes to a Newly Created Category

Next, create an CONTRACTOR_TAG metadata scheme such as:

```

<?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">
<schema 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"/>
  </level>
</schema>

```

```
    </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">
<schema 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>
</schema>
</TagDataTranslation>
```

To add these schemes to the category ID previously created, use the `DBMS_MGD_ID_UTIL.add_scheme` function.

This 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 it exists
  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">
<schema 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>
</schema>
</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">

```

```

<schema 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>
</schema>
</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();

@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
.

.

.

```

22.4 Identity Code Package Types

Table 22-2 describes the Identity Code Package ADTs.

Table 22–2 Identity Code Package ADTs

ADT Name	Description
MGD_ID_COMPONENT ADT	A data type that specifies the name and value pair attributes that define a component.
MGD_ID_COMPONENT_VARRAY ADT	A data type 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 ADT	Represents an identity code type that specifies the category identifier for the code category for this identity code and its list of components.

[Table 22–3](#) describes the subprograms in the MGD_ID ADT.

All the values and names passed to the subprograms defined in the MGD_ID ADT are case-insensitive unless otherwise noted. To preserve case, enclose values in double quotation marks.

Table 22–3 MGD_ID ADT Subprograms

Subprogram	Description
MGD_ID Constructor Function	Creates an identity code type object, MGD_ID, and returns self.
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.

22.5 DBMS_MGD_ID_UTL Package

[Table 22–4](#) 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 ADT are case-insensitive unless otherwise noted. To preserve case, enclose values in double quotation marks.

Table 22–4 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 category or a version of a category.
EPC_TO_ORACLE Function	Converts the EPCglobal tag data translation (TDT) XML to Oracle Database 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.

Table 22-4 (Cont.) DBMS_MGD_ID_UTL Package Utility Subprograms

Subprogram	Description
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 Database tag data translation XML for the specified scheme.
GET_VALIDATOR Function	Returns the Oracle Database tag data translation schema.
REFRESH_CATEGORY Function	Refreshes the metadata information about the Java stack for the specified category.
REMOVE_CATEGORY 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.
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 Database tag data translation schema.

22.6 Identity Code Metadata Tables and Views

This topic 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 is 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 ADT is designed to understand the encodings if the metadata for the encodings are stored in the meta tables. If an application developer uses only 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` stores the encoding category information defined by the session user.
- `user_mgd_id_scheme` stores the encoding type information defined by the session user.

You can query the following read-only views to see the system's predefined encoding metadata and the metadata defined by the user:

- `mgd_id_category` lets you query the encoding category information defined by the system or the session user

- `mgd_id_scheme` lets you 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 22–5](#), [Table 22–6](#), [Table 22–7](#), and [Table 22–8](#) describe the metadata view definitions for the `MGD_ID_CATEGORY`, `USER_ID_CATEGORY`, `MGD_ID_SCHEME`, and `USER_MGD_ID_SCHEME` respectively as defined in the `mgdview.sql` file.

Table 22–5 Definition and Description of the `MGD_ID_CATEGORY` Metadata View

Column Name	Data Type	Description
<code>CATEGORY_ID</code>	NUMBER (4)	Category identifier
<code>CATEGORY_NAME</code>	VARCHAR2 (256)	Category name
<code>AGENCY</code>	VARCHAR2 (256)	Organization that defined the category
<code>VERSION</code>	VARCHAR2 (256)	Category version
<code>URI</code>	VARCHAR2 (256)	URI that describes the category

Table 22–6 Definition and Description of the `USER_MGD_ID_CATEGORY` Metadata View

Column Name	Data Type	Description
<code>CATEGORY_ID</code>	NUMBER (4)	Category identifier
<code>CATEGORY_NAME</code>	VARCHAR2 (256)	Category name
<code>AGENCY</code>	VARCHAR2 (256)	Organization that defined the category
<code>VERSION</code>	VARCHAR2 (256)	Category version
<code>URI</code>	VARCHAR2 (256)	URI that describes the category

Table 22–7 Definition and Description of the `MGD_ID_SCHEME` Metadata View

Column Name	Data Type	Description
<code>CATEGORY_ID</code>	NUMBER (4)	Category identifier
<code>TYPE_NAME</code>	VARCHAR2 (256)	Encoding scheme name, for example, SGTIN-96, GID-96, and so on
<code>TDT_XML</code>	CLOB	Tag data translation XML for this encoding scheme
<code>ENCODINGS</code>	VARCHAR2 (256)	Encodings separated by a comma (,), for example, LEGACY, TAG_ENCODING, PURE_IDENTITY, BINARY (for SGTIN-96)

Table 22-7 (Cont.) Definition and Description of the MGD_ID_SCHEME Metadata View

Column Name	Data Type	Description
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 22-8 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 on
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)

22.7 Electronic Product Code (EPC) Concepts

Topics:

- [RFID Technology and EPC v1.1 Coding Schemes](#)
- [Product Code Concepts and Their Current Use](#)

22.7.1 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 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. Therefore, 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 these 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 features to support EPC data encoding and decoding, and 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 data types, metadata tables, and PL/SQL packages for these purposes.

22.7.2 Product Code Concepts and Their Current Use

This topic describes these 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](#)

22.7.2.1 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, and 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 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 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 EPCCGlobal 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 URLs, 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®).

22.7.2.1.1 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.

22.7.2.1.2 EPC Encoding EPC encoding is a pure identity with more information, such as filter value, rendered into a specific syntax (typically consisting of value fields of specific sizes). A given pure identity might have several possible encodings, such as a Barcode Encoding, various Tag Encodings, and various URI Encodings. Encodings can 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.

22.7.2.1.3 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.

22.7.2.1.4 EPC Identity URI The EPC identity URI is a representation of a pure identity as a Uniform Resource Identifier (URI).

22.7.2.1.5 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.

22.7.2.1.6 EPC Encoding Procedure The EPC encoding procedure generates an EPC tag bit-level encoding using various information.

22.7.2.1.7 EPC Decoding Procedure The EPC decoding procedure converts an EPC tag bit-level encoding to an EAN.UCC code.

22.7.2.2 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 an 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).

22.7.2.3 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.

22.7.2.4 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. Also, 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.

22.7.2.5 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.

22.7.2.6 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 intended for assignment to individual objects. Global Individual Asset Identifier (GIAI) uniquely identifies an entity that is part of the fixed inventory of a company. The GIAI identifies any fixed asset of an organization.

22.7.2.7 RFID EPC Network

The RFID EPC network identifies, tracks, and locates assets. Physical objects are identified by a unique RFID enabled EPC.

22.8 Oracle Database Tag Data Translation Schema

The Oracle Database Tag Data Translation Schema is closely related to the EPCglobal TDT schema, however it is not exact. The Oracle Database 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>
```

Understanding Schema Object Dependency

If the definition of object A references object B, then A depends on B. This chapter explains dependencies among schema objects, and how Oracle Database automatically tracks and manages these dependencies. Because of this automatic dependency management, A never uses an obsolete version of B, and you almost never have to explicitly recompile A after you change B.

Topics:

- [Overview of Schema Object Dependency](#)
- [Querying Object Dependencies](#)
- [Object Status](#)
- [Invalidation of Dependent Objects](#)
- [Guidelines for Reducing Invalidation](#)
- [Object Revalidation](#)
- [Name Resolution in Schema Scope](#)
- [Local Dependency Management](#)
- [Remote Dependency Management](#)
- [Remote Procedure Call \(RPC\) Dependency Management](#)
- [Shared SQL Dependency Management](#)

23.1 Overview of Schema Object Dependency

Some types of schema objects can reference other objects in their definitions. For example, a view is defined by a query that references tables or other views, and the body of a subprogram can include SQL statements that reference other objects. If the definition of object A references object B, then A is a **dependent object** (of B) and B is a **referenced object** (of A).

Example 23–1 shows how to display the dependent and referenced object types in your database (if you are logged in as DBA).

Example 23–1 Displaying Dependent and Referenced Object Types

Display dependent object types:

```
SELECT DISTINCT TYPE
  FROM DBA_DEPENDENCIES
 ORDER BY TYPE;
```

Result:

```
TYPE
-----
DIMENSION
EVALUATION CONTEXT
FUNCTION
INDEX
INDEXTYPE
JAVA CLASS
JAVA DATA
MATERIALIZED VIEW
OPERATOR
PACKAGE
PACKAGE BODY
PROCEDURE
RULE
RULE SET
SYNONYM
TABLE
TRIGGER
TYPE
TYPE BODY
UNDEFINED
VIEW
XML SCHEMA
```

22 rows selected.

Display referenced object types:

```
SELECT DISTINCT REFERENCED_TYPE
FROM DBA_DEPENDENCIES
ORDER BY REFERENCED_TYPE;
```

Result:

```
REFERENCED_TYPE
-----
EVALUATION CONTEXT
FUNCTION
INDEX
INDEXTYPE
JAVA CLASS
LIBRARY
OPERATOR
PACKAGE
PROCEDURE
SEQUENCE
SYNONYM
TABLE
TYPE
VIEW
XML SCHEMA
```

14 rows selected.

If you alter the definition of a referenced object, dependent objects might not continue to function without error, depending on the type of alteration. For example, if you drop a table, no view based on the dropped table is usable.

As an example of a schema object change that invalidates some dependents but not others, consider the two views in [Example 23–2](#), which are based on the HR.EMPLOYEES table.

[Example 23–2](#) creates two views from the EMPLOYEES table: SIXFIGURES, which selects all columns in the table, and COMMISSIONED, which does not include the EMAIL column. As the example shows, changing the EMAIL column invalidates SIXFIGURES, but not COMMISSIONED.

Example 23–2 Schema Object Change that Invalidates Some Dependents

```
CREATE OR REPLACE VIEW sixfigures AS
SELECT * FROM employees
WHERE salary >= 100000;

CREATE OR REPLACE VIEW commissioned AS
SELECT first_name, last_name, commission_pct
FROM employees
WHERE commission_pct > 0.00;
```

SQL*Plus formatting command:

```
COLUMN object_name FORMAT A16
```

Query:

```
SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW'
ORDER BY object_name;
```

Result:

OBJECT_NAME	STATUS
COMMISSIONED	VALID
EMP_DETAILS_VIEW	VALID
SIXFIGURES	VALID

3 rows selected.

Lengthen EMAIL column of EMPLOYEES table:

```
ALTER TABLE employees MODIFY email VARCHAR2(100);
```

Query:

```
SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW'
ORDER BY object_name;
```

Result:

OBJECT_NAME	STATUS
COMMISSIONED	VALID
EMP_DETAILS_VIEW	INVALID
SIXFIGURES	VALID

A view depends on every object referenced in its query. The view in [Example 23–3](#), depends on the tables employees and departments.

Example 23–3 View that Depends on Multiple Objects

```
CREATE OR REPLACE VIEW v AS
  SELECT last_name, first_name, department_name
  FROM employees e, departments d
  WHERE e.department_id = d.department_id
  ORDER BY last_name;
```

Notes:

- CREATE statements automatically update all dependencies.
- Dynamic SQL statements do not create dependencies. For example, this statement does not create a dependency on tab1:

```
EXECUTE IMMEDIATE 'SELECT * FROM tab1'
```

23.2 Querying Object Dependencies

The static data dictionary views `USER_DEPENDENCIES`, `ALL_DEPENDENCIES`, and `DBA_DEPENDENCIES` describe dependencies between database objects.

The `utldtree.sql` SQL script creates the view `DEPTREE`, which contains information on the object dependency tree, and the view `IDEPTREE`, a presorted, pretty-print version of `DEPTREE`.

See Also: *Oracle Database Reference* for more information about the `DEPTREE`, `IDEPTREE`, and `utldtree.sql` script

23.3 Object Status

Every database object has a status value described in [Table 23–1](#).

Table 23–1 Database Object Status

Status	Meaning
Valid	The object was successfully compiled, using the current definition in the data dictionary.
Compiled with errors	The most recent attempt to compile the object produced errors.
Invalid	The object is marked invalid because an object that it references has changed. (Only a dependent object can be invalid.)
Unauthorized	An access privilege on a referenced object was revoked. (Only a dependent object can be unauthorized.)

Note: The static data dictionary views `USER_OBJECTS`, `ALL_OBJECTS`, and `DBA_OBJECTS` do not distinguish between "Compiled with errors," "Invalid," and "Unauthorized"—they describe all of these as `INVALID`.

23.4 Invalidation of Dependent Objects

If object A depends on object B, which depends on object C, then A is a **direct dependent** of B, B is a direct dependent of C, and A is an **indirect dependent** of C.

Direct dependents are invalidated only by changes to the referenced object that affect them (changes to the signature of the referenced object).

Indirect dependents can be invalidated by changes to the reference object that do not affect them. If a change to C invalidates B, it invalidates A (and all other direct and indirect dependents of B). This is called **cascading invalidation**.

With **coarse-grained invalidation**, a data definition language (DDL) statement that changes a referenced object invalidates all of its dependents.

With **fine-grained invalidation**, a DDL statement that changes a referenced object invalidates only dependents for which either of these statements is true:

- The dependent relies on the attribute of the referenced object that the DDL statement changed.
- The compiled metadata of the dependent is no longer correct for the changed referenced object.

For example, if view v selects columns c1 and c2 from table t, a DDL statement that changes only column c3 of t does not invalidate v.

The DDL statement `CREATE OR REPLACE object` has no effect under these conditions:

- *object* is a PL/SQL object, the new PL/SQL source text is identical to the existing PL/SQL source text, and the PL/SQL compilation parameter settings stored with *object* are identical to those in the session environment.

For information about PL/SQL compilation parameter settings, see *Oracle Database PL/SQL Language Reference*.

- *object* is a synonym and the statement does not change the target object.

The operations in the left column of [Table 23–2](#) cause fine-grained invalidation, except in the cases in the right column. The cases in the right column, and all operations not listed in [Table 23–2](#), cause coarse-grained invalidation.

Table 23–2 Operations that Cause Fine-Grained Invalidation

Operation	Exceptions
<code>ALTER TABLE table ADD column</code>	<ul style="list-style-type: none"> ▪ Dependent object (except a view) uses <code>SELECT * on table</code>. ▪ Dependent object uses <code>table%rowtype</code>. ▪ Dependent object performs <code>INSERT</code> on <code>table</code> without specifying column list. ▪ Dependent object references <code>table</code> in query that contains a join. ▪ Dependent object references <code>table</code> in query that references a PL/SQL variable.

Table 23–2 (Cont.) Operations that Cause Fine-Grained Invalidation

Operation	Exceptions
ALTER TABLE <i>table</i> {MODIFY RENAME DROP SET UNUSED} <i>column</i> ALTER TABLE <i>table</i> DROP CONSTRAINT <i>not_null_constraint</i>	<ul style="list-style-type: none"> ▪ Dependent object directly references <i>column</i>. ▪ Dependent object uses SELECT * on <i>table</i>. ▪ Dependent object uses <i>table</i>%ROWTYPE. ▪ Dependent object performs INSERT on <i>table</i> without specifying column list. ▪ Dependent object is a trigger that depends on an entire row (that is, it does not specify a column in its definition). ▪ Dependent object is a trigger that depends on a column to the right of the dropped column.
CREATE OR REPLACE VIEW <i>view</i> Online Table Redefinition (DBMS_REDEFINITION)	<p>Column lists of new and old definitions differ, and at least one of these is true:</p> <ul style="list-style-type: none"> ▪ Dependent object references column that is modified or dropped in new view or table definition. ▪ Dependent object uses <i>view</i>%rowtype or <i>table</i>%rowtype. ▪ Dependent object performs INSERT on view or table without specifying column list. ▪ New view definition introduces new columns, and dependent object references view or table in query that contains a join. ▪ New view definition introduces new columns, and dependent object references view or table in query that references a PL/SQL variable. ▪ Dependent object references view or table in RELIES ON clause.
CREATE OR REPLACE SYNONYM <i>synonym</i>	<ul style="list-style-type: none"> ▪ New and old <i>synonym</i> targets differ, and one is not a table. ▪ Both old and new <i>synonym</i> targets are tables, and the tables have different column lists or different privilege grants. ▪ Both old and new <i>synonym</i> targets are tables, and dependent object is a view that references a column that participates in a unique index on the old target but not in a unique index on the new target.

Table 23–2 (Cont.) Operations that Cause Fine-Grained Invalidation

Operation	Exceptions
DROP INDEX	<ul style="list-style-type: none"> ▪ The index is a function-based index and the dependent object is a trigger that depends either on an entire row or on a column that was added to <i>table</i> after a function-based index was created. ▪ The index is a unique index, the dependent object is a view, and the view references a column participating in the unique index.
CREATE OR REPLACE {PROCEDURE FUNCTION}	<p>Call signature changes. Call signature is the parameter list (order, names, and types of parameters), return type, ACCESSIBLE BY clause ("white list"), purity¹, determinism, parallelism, pipelining, and (if the procedure or function is implemented in C or Java) implementation properties.</p>
CREATE OR REPLACE PACKAGE	<ul style="list-style-type: none"> ▪ ACCESSIBLE BY clause ("white list") changes. ▪ Dependent object references a dropped or renamed package item. ▪ Dependent object references a package procedure or function whose call signature or entry-point number², changed. <p>If referenced procedure or function has multiple overload candidates, dependent object is invalidated if any overload candidate's call signature or entry point number changed, or if a candidate was added or dropped.</p> <ul style="list-style-type: none"> ▪ Dependent object references a package cursor whose call signature, rowtype, or entry point number changed. ▪ Dependent object references a package type or subtype whose definition changed. ▪ Dependent object references a package variable or constant whose name, data type, initial value, or offset number changed. ▪ Package purity¹ changed.

¹ Purity refers to a set of rules for preventing side effects (such as unexpected data changes) when invoking PL/SQL functions within SQL queries. Package purity refers to the purity of the code in the package initialization block.

² The entry-point number of a procedure or function is determined by its location in the PL/SQL package code. A procedure or function added to the end of a PL/SQL package is given a new entry-point number.

Note: A dependent object that is invalidated by an operation in [Table 23-2](#) appears in the static data dictionary views `*_OBJECTS` and `*_OBJECTS_AE` only after an attempt to reference it (either during compilation or execution) or after invoking one of these subprograms:

- `DBMS_UTILITY.COMPILE_SCHEMA` (described in *Oracle Database PL/SQL Packages and Types Reference*)
 - Any `UTL_RECOMP` subprogram (described in *Oracle Database PL/SQL Packages and Types Reference*)
-

Topics:

- [Session State and Referenced Packages](#)
- [Security Authorization](#)

23.4.1 Session State and Referenced Packages

Each session that references a package construct has its own instantiation of that package, including a persistent state of any public and private variables, cursors, and constants. All of a session's package instantiations, including state, can be lost if any of the session's instantiated packages are subsequently invalidated and revalidated.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about package instantiation
- *Oracle Database PL/SQL Language Reference* for information about package state

23.4.2 Security Authorization

When a data manipulation language (DML) object or system privilege is granted to, or revoked from, a user or `PUBLIC`, Oracle Database invalidates all the owner's dependent objects, to verify that an owner of a dependent object continues to have the necessary privileges for all referenced objects.

23.5 Guidelines for Reducing Invalidation

To reduce invalidation of dependent objects, follow these guidelines:

- [Section 23.5.1, "Add Items to End of Package"](#)
- [Section 23.5.2, "Reference Each Table Through a View"](#)

23.5.1 Add Items to End of Package

When adding items to a package, add them to the end of the package. This preserves the entry point numbers of existing top-level package items, preventing their invalidation.

For example, consider this package:

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER IS
  FUNCTION get_var RETURN VARCHAR2;
END;
/
```

Adding an item to the end of `pkg1`, as follows, does not invalidate dependents that reference the `get_var` function:

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER IS
    FUNCTION get_var RETURN VARCHAR2;
    PROCEDURE set_var (v VARCHAR2);
END;
/
```

Inserting an item between the `get_var` function and the `set_var` procedure, as follows, invalidates dependents that reference the `set_var` function:

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER IS
    FUNCTION get_var RETURN VARCHAR2;
    PROCEDURE assert_var (v VARCHAR2);
    PROCEDURE set_var (v VARCHAR2);
END;
/
```

23.5.2 Reference Each Table Through a View

Reference tables indirectly, using views, enabling you to:

- Add columns to the table without invalidating dependent views or dependent PL/SQL objects
- Modify or delete columns not referenced by the view without invalidating dependent objects

The statement `CREATE OR REPLACE VIEW` does not invalidate an existing view or its dependents if the new `ROWTYPE` matches the old `ROWTYPE`.

23.6 Object Revalidation

An object that is not valid when it is referenced must be validated before it can be used. Validation occurs automatically when an object is referenced; it does not require explicit user action.

If an object is not valid, its status is either compiled with errors, unauthorized, or invalid. For definitions of these terms, see [Table 23–1](#).

Topics:

- [Revalidation of Objects that Compiled with Errors](#)
- [Revalidation of Unauthorized Objects](#)
- [Revalidation of Invalid SQL Objects](#)
- [Revalidation of Invalid PL/SQL Objects](#)

23.6.1 Revalidation of Objects that Compiled with Errors

The compiler cannot automatically revalidate an object that compiled with errors. The compiler recompiles the object, and if it recompiles without errors, it is revalidated; otherwise, it remains invalid.

23.6.2 Revalidation of Unauthorized Objects

The compiler checks whether the unauthorized object has access privileges to all of its referenced objects. If so, the compiler revalidates the unauthorized object without recompiling it. If not, the compiler issues appropriate error messages.

23.6.3 Revalidation of Invalid SQL Objects

The SQL compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid.

23.6.4 Revalidation of Invalid PL/SQL Objects

For an invalid PL/SQL program unit (procedure, function, or package), the PL/SQL compiler checks whether any referenced object changed in a way that affects the invalid object. If so, the compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid. If not, the compiler revalidates the invalid object without recompiling it.

23.7 Name Resolution in Schema Scope

Object names referenced in SQL statements have one or more pieces. Pieces are separated by periods—for example, `hr.employees.department_id` has three pieces.

Oracle Database uses the following procedure to try to resolve an object name.

Note: For the procedure to succeed, all pieces of the object name must be visible in the current edition. For more information, see [Section 24.1.1.1, "Name Resolution for Editioned and Noneditioned Objects."](#)

1. Try to qualify the first piece of the object name.

If the object name has only one piece, then that piece is the first piece. Otherwise, the first piece is the piece to the left of the leftmost period; for example, in `hr.employees.department_id`, the first piece is `hr`.

The procedure for trying to qualify the first piece is:

- If the object name is a table name that appears in the `FROM` clause of a `SELECT` statement, and the object name has multiple pieces, go to step d. Otherwise, go to step b.
- Search the current schema for an object whose name matches the first piece.
If found, go to step 2. Otherwise, go to step c.
- Search for a public synonym that matches the first piece.
If found, go to step 2. Otherwise, go to step d.
- Search for a schema whose name matches the first piece.
If found, and if the object name has a second piece, go to step e. Otherwise, return an error—the object name cannot be qualified.
- Search the schema found at step d for a table or SQL function whose name matches the second piece of the object name.

If found, go to step 2. Otherwise, return an error—the object name cannot be qualified.

Note: A SQL function found at this step has been redefined by the schema found at step d.

2. A schema object has been qualified. Any remaining pieces of the object name must match a valid part of this schema object.

For example, if the object name is `hr.employees.department_id`, `hr` is qualified as a schema. If `employees` is qualified as a table, `department_id` must correspond to a column of that table. If `employees` is qualified as a package, `department_id` must correspond to a public constant, variable, procedure, or function of that package.

Because of how Oracle Database resolves references, an object can depend on the nonexistence of other objects. This situation occurs when the dependent object uses a reference that would be interpreted differently if another object were present.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about how name resolution differs in SQL and PL/SQL
- *Oracle Database Administrator's Guide* for information about name resolution in a distributed database system

23.8 Local Dependency Management

Local dependency management occurs when Oracle Database manages dependencies among the objects in a single database. For example, a statement in a procedure can reference a table in the same database.

23.9 Remote Dependency Management

Remote dependency management occurs when Oracle Database manages dependencies in distributed environments across a network. For example, an Oracle Forms trigger can depend on a schema object in the database. In a distributed database, a local view can reference a remote table.

Oracle Database also manages distributed database dependencies. For example, an Oracle Forms application might contain a trigger that references a table. The database system must account for dependencies among such objects. Oracle Database uses different mechanisms to manage remote dependencies, depending on the objects involved.

Topics:

- [Dependencies Among Local and Remote Database Procedures](#)
- [Dependencies Among Other Remote Objects](#)
- [Dependencies of Applications](#)

23.9.1 Dependencies Among Local and Remote Database Procedures

Dependencies among stored procedures (including functions, packages, and triggers) in a distributed database system are managed using either time-stamp checking (see [Section 23.10.1](#)) or signature checking (see [Section 23.10.2](#)).

The dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE` determines whether time stamps or signatures govern remote dependencies.

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

23.9.2 Dependencies Among Other Remote Objects

Oracle Database does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies.

For example, assume that a local view is created and defined by a query that references a remote table. Also assume that a local procedure includes a SQL statement that references the same remote table. Later, the definition of the table is altered.

Therefore, the local view and procedure are never invalidated, even if the view or procedure is used after the table is altered, and even if the view or procedure now returns errors when used. In this case, the view or procedure must be altered manually so that errors are not returned. In such cases, lack of dependency management is preferable to unnecessary recompilations of dependent objects.

23.9.3 Dependencies of Applications

Code in database applications can reference objects in the connected database. For example, Oracle Call Interface (OCI) and precompiler applications can submit anonymous PL/SQL blocks. Triggers in Oracle Forms applications can reference a schema object.

Such applications are dependent on the schema objects they reference. Dependency management techniques vary, depending on the development environment. Oracle Database does not automatically track application dependencies.

See Also: Manuals for your application development tools and your operating system for more information about managing the remote dependencies within database applications

23.10 Remote Procedure Call (RPC) Dependency Management

Remote procedure call (RPC) dependency management occurs when a local stored procedure calls a remote procedure in a distributed database system. The dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE` controls the dependency mode. The choice is either time-stamp dependency mode or RPC-signature dependency mode.

Topics:

- [Time-Stamp Dependency Mode](#)
- [RPC-Signature Dependency Mode](#)
- [Controlling Dependency Mode](#)

23.10.1 Time-Stamp Dependency Mode

Whenever a procedure is compiled, its **time stamp** is recorded in the data dictionary. The time stamp shows when the procedure was created, altered, or replaced.

A compiled procedure contains information about each remote procedure that it calls, including the schema, package name, procedure name, and time stamp of the remote procedure.

In time-stamp dependency mode, when a local stored procedure calls a remote procedure, Oracle Database compares the time stamp that the local procedure has for the remote procedure to the current time stamp of the remote procedure. If the two time stamps match, both the local and remote procedures run. Neither is recompiled.

If the two time stamps do not match, the local procedure is invalidated and an error is returned to the calling environment. All other local procedures that depend on the remote procedure with the new time stamp are also invalidated.

Time stamp comparison occurs when a statement in the body of the local procedure calls the remote procedure. Therefore, statements in the local procedure that precede the invalid call might run successfully. Statements after the invalid call do not run. The local procedure must be recompiled.

If DML statements precede the invalid call, they roll back only if they and the invalid call are in the same PL/SQL block. For example, the `UPDATE` statement rolls back in this code:

```
BEGIN
    UPDATE table SET ...
    invalid_proc;
    COMMIT;
END;
```

But the `UPDATE` statement does not roll back in this code:

```
UPDATE table SET ...
EXECUTE invalid_proc;
COMMIT;
```

The disadvantages of time-stamp dependency mode are:

- Dependent objects across the network are often recompiled unnecessarily, degrading performance.
- If the client-side application uses PL/SQL, this mode can cause situations that prevent the application from running on the client side.

An example of such an application is Oracle Forms. During installation, you must recompile the client-side PL/SQL procedures that Oracle Forms uses at the client site. Also, if a client-side procedure depends on a server procedure, and if the server procedure changes or is automatically recompiled, you must recompile the client-side PL/SQL procedure. However, no PL/SQL compiler is available on the client. Therefore, the developer of the client application must distribute new versions of the application to all customers.

23.10.2 RPC-Signature Dependency Mode

Oracle Database provides **RPC signatures** to handle remote dependencies. RPC signatures do not affect local dependencies, because recompilation is always possible in the local environment.

An RPC signature is associated with each compiled stored program unit. It identifies the unit by these characteristics:

- Name
- Number of parameters
- Data type class of each parameter
- Mode of each parameter

- Data type class of return value (for a function)

An RPC signature changes only when at least one of the preceding characteristics changes.

Note: An RPC signature does not include DETERMINISTIC, PARALLEL_ENABLE, or purity information. If these settings change for a function on remote system, optimizations based on them are not automatically reconsidered. Therefore, calling the remote function in a SQL statement or using it in a function-based index might cause incorrect query results.

A compiled program unit contains the RPC signature of each remote procedure that it calls (and the schema, package name, procedure name, and time stamp of the remote procedure).

In RPC-signature dependency mode, when a local program unit calls a subprogram in a remote program unit, the database ignores time-stamp mismatches and compares the RPC signature that the local unit has for the remote subprogram to the current RPC signature of the remote subprogram. If the RPC signatures match, the call succeeds; otherwise, the database returns an error to the local unit, and the local unit is invalidated.

For example, suppose that this procedure, `get_emp_name`, is stored on a server in Boston (BOSTON_SERVER):

```
CREATE OR REPLACE PROCEDURE get_emp_name (
    emp_number  IN NUMBER,
    hiredate    OUT VARCHAR2,
    emp_name    OUT VARCHAR2) AUTHID DEFINER AS
BEGIN
    SELECT last_name, TO_CHAR(hire_date, 'DD-MON-YY')
    INTO emp_name, hiredate
    FROM employees
    WHERE employee_id = emp_number;
END;
/
```

When `get_emp_name` is compiled on BOSTON_SERVER, Oracle Database records both its RPC signature and its time stamp.

Suppose that this PL/SQL procedure, `print_name`, which calls `get_emp_name`, is on a server in California:

```
CREATE OR REPLACE PROCEDURE print_ename (emp_number IN NUMBER) AUTHID DEFINER AS
    hiredate  VARCHAR2(12);
    ename     VARCHAR2(10);
BEGIN
    get_emp_name@BOSTON_SERVER(emp_number, hiredate, ename);
    dbms_output.put_line(ename);
    dbms_output.put_line(hiredate);
END;
/
```

When `print_name` is compiled on the California server, the database connects to the Boston server, sends the RPC signature of `get_emp_name` to the California server, and records the RPC signature of `get_emp_name` in the compiled state of `print_ename`.

At runtime, when `print_name` calls `get_emp_name`, the database sends the RPC signature of `get_emp_name` that was recorded in the compiled state of `print_ename` to the Boston server. If the recorded RPC signature matches the current RPC signature of `get_emp_name` on the Boston server, the call succeeds; otherwise, the database returns an error to `print_name`, which is invalidated.

Topics:

- [Changing Names and Default Values of Parameters](#)
- [Changing Specification of Parameter Mode IN](#)
- [Changing Subprogram Body](#)
- [Changing Data Type Classes of Parameters](#)
- [Changing Package Types](#)

23.10.2.1 Changing Names and Default Values of Parameters

Changing the name or default value of a subprogram parameter does not change the RPC signature of the subprogram. For example, procedure `P1` has the same RPC signature in these two examples:

```
PROCEDURE P1 (Param1 IN NUMBER := 100);
PROCEDURE P1 (Param2 IN NUMBER := 200);
```

However, if your application requires that callers get the new default value, you must recompile the called procedure.

23.10.2.2 Changing Specification of Parameter Mode IN

Because the subprogram parameter mode `IN` is the default, you can specify it either implicitly or explicitly. Changing its specification from implicit to explicit, or the reverse, does not change the RPC signature of the subprogram. For example, procedure `P1` has the same RPC signature in these two examples:

```
PROCEDURE P1 (Param1 NUMBER);      -- implicit specification
PROCEDURE P1 (Param1 IN NUMBER);  -- explicit specification
```

23.10.2.3 Changing Subprogram Body

Changing the body of a subprogram does not change the RPC signature of the subprogram.

[Example 23–4](#) changes only the body of the procedure `get_hire_date`; therefore, it does not change the RPC signature of `get_hire_date`.

Example 23–4 Changing Body of Procedure `get_hire_date`

```
CREATE OR REPLACE PROCEDURE get_hire_date (
    emp_number  IN NUMBER,
    hiredate    OUT VARCHAR2,
    emp_name    OUT VARCHAR2) AUTHID DEFINER AS
BEGIN
    SELECT last_name, TO_CHAR(hire_date, 'DD-MON-YY')
    INTO emp_name, hiredate
    FROM employees
    WHERE employee_id = emp_number;
END;
/

CREATE OR REPLACE PROCEDURE get_hire_date (
```

```

emp_number  IN  NUMBER,
hiredate    OUT VARCHAR2,
emp_name    OUT VARCHAR2) AUTHID DEFINER AS
BEGIN
  -- Change date format model
  SELECT last_name, TO_CHAR(hire_date, 'DD/MON/YYYY')
  INTO emp_name, hiredate
  FROM employees
  WHERE employee_id = emp_number;
END;
/

```

23.10.2.4 Changing Data Type Classes of Parameters

Changing the data type of a parameter to another data type in the same class does not change the RPC signature, but changing the data type to a data type in another class does.

Table 23–3 lists the data type classes and the data types that comprise them. Data types not listed in Table 23–3, such as NCHAR, do not belong to a data type class. Changing their type always changes the RPC signature.

Table 23–3 Data Type Classes

Data Type Class	Data Types in Class
Character	CHAR CHARACTER
VARCHAR	VARCHAR VARCHAR2 STRING LONG ROWID
Raw	RAW LONG RAW
Integer	BINARY_INTEGER PLS_INTEGER SIMPLE_INTEGER BOOLEAN NATURAL NATURALN POSITIVE POSITIVEN
Number	NUMBER INT INTEGER SMALLINT DEC DECIMAL REAL FLOAT NUMERIC DOUBLE PRECISION

Table 23–3 (Cont.) Data Type Classes

Data Type Class	Data Types in Class
Datetime	DATE TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE INTERVAL YEAR TO MONTH INTERVAL DAY TO SECOND

Example 23–5 changes the data type of the parameter `hiredate` from `VARCHAR2` to `DATE`. `VARCHAR2` and `DATE` are not in the same data type class, so the RPC signature of the procedure `get_hire_date` changes.

Example 23–5 Changing Data Type Class of `get_hire_date` Parameter

```
CREATE OR REPLACE PROCEDURE get_hire_date (
    emp_number    IN NUMBER,
    hiredate      OUT DATE,
    emp_name      OUT VARCHAR2) AS
BEGIN
    SELECT last_name, TO_CHAR(hire_date, 'DD/MON/YYYY')
    INTO emp_name, hiredate
    FROM employees
    WHERE employee_id = emp_number;
END;
/
```

23.10.2.5 Changing Package Types

Changing the name of a package type, or the names of its internal components, does not change the RPC signature of the package.

Example 23–6 defines a record type, `emp_data_type`, inside the package `emp_package`. Next, it changes the names of the record fields, but not their types. Finally, it changes the name of the type, but not its characteristics. The RPC signature of the package does not change.

Example 23–6 Changing Names of Fields in Package Record Type

```
CREATE OR REPLACE PACKAGE emp_package AUTHID DEFINER AS
    TYPE emp_data_type IS RECORD (
        emp_number    NUMBER,
        hiredate      VARCHAR2(12),
        emp_name      VARCHAR2(10)
    );
    PROCEDURE get_emp_data (
        emp_data    IN OUT emp_data_type
    );
END;
/

CREATE OR REPLACE PACKAGE emp_package AUTHID DEFINER AS
    TYPE emp_data_type IS RECORD (
        emp_num      NUMBER,
        hire_dat    VARCHAR2(12),
        empname     VARCHAR2(10)
    );
    PROCEDURE get_emp_data (
        emp_data    IN OUT emp_data_type
    );

```

```
    );
END;
/

CREATE OR REPLACE PACKAGE emp_package AUTHID DEFINER AS
  TYPE emp_data_record_type IS RECORD (
    emp_num    NUMBER,
    hire_dat   VARCHAR2(12),
    empname    VARCHAR2(10)
  );
  PROCEDURE get_emp_data (
    emp_data  IN OUT emp_data_record_type
  );
END;
/
```

23.10.3 Controlling Dependency Mode

The dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE` controls the dependency mode. If the initialization parameter file contains this specification, then only time stamps are used to resolve dependencies (if this is not explicitly overridden dynamically):

```
REMOTE_DEPENDENCIES_MODE = TIMESTAMP
```

If the initialization parameter file contains this parameter specification, then RPC signatures are used to resolve dependencies (if this not explicitly overridden dynamically):

```
REMOTE_DEPENDENCIES_MODE = SIGNATURE
```

You can alter the mode dynamically by using the DDL statements. For example, this example alters the dependency mode for the current session:

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = {SIGNATURE | TIMESTAMP}
```

This example alters the dependency mode systemwide after startup:

```
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE = {SIGNATURE | TIMESTAMP}
```

If the `REMOTE_DEPENDENCIES_MODE` parameter is not specified, either in the `init.ora` parameter file or using the `ALTER SESSION` or `ALTER SYSTEM` statements, `TIMESTAMP` is the default value. Therefore, unless you explicitly use the `REMOTE_DEPENDENCIES_MODE` parameter, or the appropriate DDL statement, your server is operating using the time-stamp dependency mode.

When you use `REMOTE_DEPENDENCIES_MODE=SIGNATURE`:

- If you change the initial value of a parameter of a remote procedure, then the local procedure calling the remote procedure is not invalidated. If the call to the remote procedure does not supply the parameter, then the initial value is used. In this case, because invalidation and recompilation does not automatically occur, the old initial value is used. To see the new initial values, recompile the calling procedure manually.
- If you add an overloaded procedure in a package (a procedure with the same name as an existing one), then local procedures that call the remote procedure are not invalidated. If it turns out that this overloading results in a rebinding of existing calls from the local procedure under the time-stamp mode, then this rebinding does not happen under the RPC signature mode, because the local

procedure does not get invalidated. You must recompile the local procedure manually to achieve the rebinding.

- If the types of parameters of an existing package procedure are changed so that the new types have the same shape as the old ones, then the local calling procedure is not invalidated or recompiled automatically. You must recompile the calling procedure manually to get the semantics of the new type.

Topics:

- [Dependency Resolution](#)
- [Suggestions for Managing Dependencies](#)

23.10.3.1 Dependency Resolution

When `REMOTE_DEPENDENCIES_MODE = TIMESTAMP` (the default value), dependencies among program units are handled by comparing time stamps at runtime. If the time stamp of a called remote procedure does not match the time stamp of the called procedure, then the calling (dependent) unit is invalidated and must be recompiled. In this case, if there is no local PL/SQL compiler, then the calling application cannot proceed.

In the time-stamp dependency mode, RPC signatures are not compared. If there is a local PL/SQL compiler, then recompilation happens automatically when the calling procedure is run.

When `REMOTE_DEPENDENCIES_MODE = SIGNATURE`, the recorded time stamp in the calling unit is first compared to the current time stamp in the called remote unit. If they match, then the call proceeds. If the time stamps do not match, then the RPC signature of the called remote subprogram, as recorded in the calling subprogram, is compared with the current RPC signature of the called subprogram. If they do not match (using the criteria described in [Section 23.10.2.4](#)), then an error is returned to the calling session.

23.10.3.2 Suggestions for Managing Dependencies

Follow these guidelines for setting the `REMOTE_DEPENDENCIES_MODE` parameter:

- Server-side PL/SQL users can set the parameter to `TIMESTAMP` (or let it default to that) to get the time-stamp dependency mode.
- Server-side PL/SQL users can use RPC-signature dependency mode if they have a distributed system and they want to avoid possible unnecessary recompilations.
- Client-side PL/SQL users must set the parameter to `SIGNATURE`. This allows:
 - Installation of applications at client sites without recompiling procedures.
 - Ability to upgrade the server, without encountering time stamp mismatches.
- When using RPC signature mode on the server side, add procedures to the end of the procedure (or function) declarations in a package specification. Adding a procedure in the middle of the list of declarations can cause unnecessary invalidation and recompilation of dependent procedures.

23.11 Shared SQL Dependency Management

In addition to managing dependencies among schema objects, Oracle Database also manages dependencies of each shared SQL area in the shared pool. If a table, view, synonym, or sequence is created, altered, or dropped, or a procedure or package

specification is recompiled, all dependent shared SQL areas are invalidated. At a subsequent execution of the cursor that corresponds to an invalidated shared SQL area, Oracle Database reparses the SQL statement to regenerate the shared SQL area.

24

Using Edition-Based Redefinition

Edition-based redefinition (EBR) lets you upgrade the database component of an application while it is in use, thereby minimizing or eliminating downtime.

To upgrade an application while it is in use, you copy the database objects that comprise the application and redefine the copied objects in isolation. Your changes do not affect users of the application—they continue to run the unchanged application. When you are sure that your changes are correct, you make the upgraded application available to all users.

Using EBR means using one or more of its component features. The features you use, and the downtime, depend on these factors:

- What kind of database objects you redefine
- How available the database objects must be to users while you are redefining them
- Whether you make the upgraded application available to some users while others continue to use the older version of the application

You always use the **edition** feature to copy the database objects and redefine the copied objects in isolation; that is why the procedure that this chapter describes for upgrading applications online is called edition-based redefinition (EBR).

If every object that you will redefine is **editioned** (defined in [Section 24.1.1](#)), then the edition is the only feature you use.

Tables are not editioned objects. If you change the structure of one or more tables, then you also use the **editioning view** feature.

If other users must be able to change data in the tables while you are changing their structure, then you also use **forward crossedition triggers**. If the pre- and post-upgrade applications will be in ordinary use at the same time (**hot rollover**), then you also use **reverse crossedition triggers**. Crossedition triggers are not a permanent part of the application—you drop them when all users are using the post-upgrade application.

An EBR operation that you can perform on an application in one edition while the application runs in other editions is a **live operation**.

Topics:

- [Editions](#)
- [Editioning Views](#)
- [Crossedition Triggers](#)
- [Displaying Information About EBR Features](#)

- [Using EBR to Upgrade an Application](#)

24.1 Editions

Editions are nonschema objects; as such, they do not have owners. Editions are created in a single namespace, and multiple editions can coexist in the database.

The database must have at least one edition. Every newly created or upgraded Oracle Database starts with one edition named `ora$base`.

Note: In a multitenant container database (CDB), the scope of an edition, editioning view, or crossedition trigger is the pluggable database (PDB) in which the feature was created. In a non-CDB, the scope of each of these features is the entire database. For information about CDBs and PDBs, see *Oracle Database Administrator's Guide*.

Topics:

- [Editioned and Noneditioned Objects](#)
- [Creating an Edition](#)
- [Editioned Objects and Copy-on-Change](#)
- [Making an Edition Available to Some Users](#)
- [Making an Edition Available to All Users](#)
- [Current Edition and Session Edition](#)
- [Retiring an Edition](#)
- [Dropping an Edition](#)

24.1.1 Editioned and Noneditioned Objects

Note: The terms **user** and **schema** are synonymous. The **owner** of a schema object is the user/schema that owns it.

An **editioned object** has both a schema object type that is editionable in its owner and the `EDITIONABLE` property. An edition has its own copy of an editioned object, and only that copy is visible to the edition.

A **noneditioned object** has either a schema object type that is noneditionable in its owner or the `NONEDITIONABLE` property. An edition cannot have its own copy of a noneditioned object. A noneditioned object is visible to all editions.

An object is **potentially editioned** if enabling editions for its type in its owner would make it an editioned object (for details, see [Section 24.1.1.4, "Enabling Editions for a User"](#)).

An editioned object belongs to both a schema and an edition, and is uniquely identified by its `OBJECT_NAME`, `OWNER`, and `EDITION_NAME`. A noneditioned object belongs only to a schema, and is uniquely identified by its `OBJECT_NAME` and `OWNER`—its `EDITION_NAME` is `NULL`. (Strictly speaking, the `NAMESPACE` of an object is also required to uniquely identify the object, but you can ignore this fact, because any statement that references the object implicitly or explicitly specifies its `NAMESPACE`.)

You can display the OBJECT_NAME, OWNER, and EDITION_NAME of an object with the static data dictionary views `*_OBJECTS` and `*_OBJECTS_AE`.

You need not know the EDITION_NAME of an object to refer to that object (and if you do know it, you cannot specify it). The context of the reference implicitly specifies the edition. If the context is a data definition language (DDL) statement, then the edition is the current edition of the session that issued the command (for information about the current edition, see [Section 24.1.6](#)). If the context is source code, then the edition is the one in which the object is actual (see [Section 24.1.3](#)).

Topics:

- [Name Resolution for Editioned and Noneditioned Objects](#)
- [Noneditioned Objects That Can Depend on Editioned Objects](#)
- [Editionable and Noneditionable Schema Object Types](#)
- [Enabling Editions for a User](#)
- [EDITIONABLE and NONEDITIONABLE Properties](#)
- [Rules for Editioned Objects](#)

24.1.1.1 Name Resolution for Editioned and Noneditioned Objects

To try to resolve an object name, Oracle Database uses the procedure described in [Section 23.7](#). For the procedure to succeed, all pieces of the object name must be visible in the current edition.

During name resolution for an editioned object, both editioned objects in the current edition and noneditioned objects are visible.

During name resolution for a noneditioned object, only noneditioned objects are visible. Therefore, if you try to create a noneditioned object that references an editioned object (except in the cases described in [Section 24.1.1.2](#)), the creation fails with an error.

When you change a referenced editioned object, all of its dependents (direct and indirect) become invalid. When an invalid object is referenced, the database tries to validate that object.

See Also:

- [Section 24.1.6, "Current Edition and Session Edition"](#)
- [Chapter 23, "Understanding Schema Object Dependency,"](#) for general information about dependencies among schema objects, including invalidation, revalidation, and name resolution

24.1.1.2 Noneditioned Objects That Can Depend on Editioned Objects

Ordinarily, a noneditioned object cannot depend on an editioned object, because the editioned object is invisible during name resolution. However, if a noneditioned object specifies an edition to search for editioned objects during name resolution—an **evaluation edition**—then it *can* depend on editioned objects. To specify an evaluation edition, a noneditioned object must be one of the following:

- Materialized view
- Virtual column

Topics:

- [Materialized Views](#)
- [Virtual Columns](#)

24.1.1.2.1 Materialized Views A materialized view is a noneditioned object that can specify an evaluation edition, thereby enabling it to depend on editioned objects. A materialized view that depends on editioned objects may be eligible for query rewrite only in a specific range of editions, which you specify in the *query_rewrite_clause*.

The simplified syntax for creating a materialized view is:

```
CREATE MATERIALIZED VIEW [ schema.] materialized_view other_clauses
[ evaluation_edition_clause ] [ query_rewrite_clause ] AS subquery
```

Where *evaluation_edition_clause* is:

```
EVALUATE USING { CURRENT EDITION | EDITION edition | NULL EDITION }
```

And *query_rewrite_clause* is:

```
{ DISABLE | ENABLE } QUERY REWRITE
[ unusable_before_clause ] [ unusable_beginning_clause ]
```

Where *unusable_before_clause* is:

```
UNUSABLE BEFORE { CURRENT EDITION | EDITION edition }
```

And *unusable_beginning_clause* is:

```
UNUSABLE BEGINNING WITH { CURRENT EDITION | EDITION edition | NULL EDITION }
```

CURRENT EDITION is the edition in which the DDL statement runs. Specifying NULL EDITION is equivalent to omitting the clause that includes it. If you omit *evaluation_edition_clause*, then editioned objects are invisible during name resolution.

For the complete syntax and semantics of the CREATE MATERIALIZED VIEW statement, see *Oracle Database SQL Language Reference*.

To disable, enable, or change the evaluation edition or unusable editions, use the ALTER MATERIALIZED VIEW statement, described in *Oracle Database SQL Language Reference*.

To display the evaluation editions and unusable editions of materialized views, use the static data dictionary views *_{_}MVIEWS (described in *Oracle Database Reference*).

Dropping the evaluation edition invalidates the materialized view. Dropping an edition where the materialized view is usable does not invalidate the materialized view.

24.1.1.2.2 Virtual Columns A virtual column (also called a "generated column") does not consume disk space. The database generates the values in a virtual column on demand by evaluating an expression. The expression can invoke PL/SQL functions (which can be editioned objects). A virtual column can specify an evaluation edition, thereby enabling it to depend on an expression that invokes editioned PL/SQL functions.

The syntax for creating a virtual column is:

```
column [ datatype ] [ GENERATED ALWAYS ] AS ( column_expression )
[ VIRTUAL ] [ evaluation_edition_clause ]
[ unusable_before_clause ] [ unusable_beginning_clause ]
[ inline_constraint ]...
```

Where *evaluation_edition_clause* is as described in [Section 24.1.1.2.1](#).

The database does not maintain dependencies on the functions that a virtual column invokes. Therefore, if you drop the evaluation edition, or if a virtual column depends on a noneditioned function and the function becomes editioned, then any of the following can raise an exception:

- Trying to query the virtual column
- Trying to update a row that includes the virtual column
- A trigger that tries to access the virtual column

For the complete syntax and semantics of the virtual column definition, see *Oracle Database SQL Language Reference*.

To display the evaluation editions of virtual columns, use the static data dictionary views `*_TAB_COLS` (described in *Oracle Database Reference*).

24.1.1.3 Editable and Noneditionable Schema Object Types

Before a schema object type can be editionable in a schema, it must be editionable in the database. The schema object types that are editionable in the database are determined by the value of the `COMPATIBLE` initialization parameter (described in *Oracle Database Administrator's Guide*) and are shown by the dynamic performance view `V$EDITIONABLE_TYPES` (described in *Oracle Database Reference*).

If the value of `COMPATIBLE` is 12 or greater, then these schema object types are editionable in the database:

- SYNONYM
- VIEW
- SQL translation profile
- All PL/SQL object types:
 - FUNCTION
 - LIBRARY
 - PACKAGE and PACKAGE BODY
 - PROCEDURE
 - TRIGGER
 - TYPE and TYPE BODY

All other schema object types are noneditionable in the database and in every schema, and objects of that type are always noneditioned. `TABLE` is an example of a noneditionable schema object type. Tables are always noneditioned objects.

If a schema object type is editionable in the database, then it can be editionable in schemas. For details, see [Section 24.1.1.4](#).

24.1.1.4 Enabling Editions for a User

Note:

- Enabling editions is not a live operation.
 - When a database is upgraded from Release 11.2 to Release 12.1, users who were enabled for editions in the pre-upgrade database are enabled for editions in the post-upgrade database and the default schema object types are editionable in their schemas. The default schema object types are displayed by the static data dictionary view `DBA_EDITIONED_TYPES` (described in *Oracle Database Reference*). Users who were not enabled for editions in the pre-upgrade database are not enabled for editions in the post-upgrade database and no schema object types are editionable in their schemas.
 - To see which users already have editions enabled, see the `EDITIONS_ENABLED` column of the static data dictionary view `DBA_USERS` (described in *Oracle Database Reference*) or `USER_USERS` (described in *Oracle Database Reference*).
-

To enable editions for a user, use the `ENABLE EDITIONS` clause of either the `CREATE USER` or `ALTER USER` statement.

With the `ALTER USER` statement, you can specify the schema object types that become editionable in the schema:

```
ALTER USER user ENABLE EDITIONS [ FOR type [, type ]... ]
```

Any type that you omit from the `FOR` list is noneditionable in the schema, despite being editionable in the database. (If a type is noneditionable in the database, then it is always noneditionable in every schema.)

If you omit the `FOR` list from the `ALTER USER` statement, or use the `CREATE USER` statement to enable editions for a user, then the types that become editionable in the schema are those shown for that schema by the static data dictionary view `DBA_EDITIONED_TYPES` (described in *Oracle Database Reference*).

See Also: *Oracle Database SQL Language Reference* for the complete syntax and semantics of the `CREATE USER` and `ALTER USER` statements

Enabling editions is retroactive and irreversible. When you enable editions for a user, that user is editions-enabled forever. When you enable editions for a schema object type in a schema, that type is editions-enabled forever in that schema. Every object that an editions-enabled user owns or will own becomes an editioned object if its type is editionable in the schema and it has the `EDITIONABLE` property. For information about the `EDITIONABLE` property, see [Section 24.1.1.5](#).

Topics:

- [Potentially Editioned Objects with Noneditioned Dependents](#)
- [Users Who Cannot Have Editions Enabled](#)

24.1.1.4.1 Potentially Editioned Objects with Noneditioned Dependents If a potentially editioned object has a noneditioned dependent, then you can enable editions for the owner of the potentially editioned object only if one of the following is true:

- Enabling editions for the owner of the potentially editioned object would cause the noneditioned dependent to become editioned.
- You specify **FORCE**:

```
ALTER USER user ENABLE EDITIONS [ FOR type [, type ]... ] FORCE;
```

The preceding statement enables editions for the specified user and invalidates noneditioned dependents of editioned objects. For information about invalidation of dependent objects, see [Section 23.4](#).

Note: If the preceding statement invalidates a noneditioned dependent object that contains an Abstract Data Type (ADT), and you drop the edition that contains the editioned object on which the invalidated object depends, then you cannot recompile the invalidated object. Therefore, the object remains invalid.

FORCE is useful in the following situation: You must editions-enable users A and B. User A owns potentially editioned objects a1 and a2. User B owns potentially editioned objects b1 and b2. Object a1 depends on object b1. Object b2 depends on object a2. Editions-enable users A and B like this:

1. Using **FORCE**, enable editions for user A:

```
ALTER USER A ENABLE EDITIONS FORCE;
```

Now a1 and a2 are editioned objects, and noneditioned object b2 (which depends on a2) is invalid.

2. Enable editions for user B:

```
ALTER USER B ENABLE EDITIONS;
```

Now b1 and b2 are editioned objects; however, b2 is still invalid.

3. Recompile b2, using the appropriate **ALTER** statement with **COMPILE**. For a PL/SQL object, also specify **REUSE SETTINGS**.

For example, if b2 is a procedure, use this statement:

```
ALTER PROCEDURE b2 COMPILE REUSE SETTINGS
```

For information about the **ALTER** statements for PL/SQL objects, see *Oracle Database PL/SQL Language Reference*.

For information about the **ALTER** statements for SQL objects, see *Oracle Database SQL Language Reference*.

FORCE is unnecessary in the following situation: You must editions-enable user C, who owns potentially editioned object c1. Object c1 has dependent d1, a potentially editioned object owned by user D. User D owns no potentially editioned objects that have dependents owned by C. If you editions-enable D first, making d1 an editioned object, then you can editions-enable C without violating the rule that a noneditioned object cannot depend on an editioned object.

24.1.1.4.2 Users Who Cannot Have Editions Enabled

You cannot enable editions for these users:

- Oracle-supplied users
- Common users in a CDB

For information about common users in a CDB, see *Oracle Database Administrator's Guide*.

- A user who owns one or more evolved ADTs.

Trying to do so causes error ORA-38820. If an ADT has no table dependents, you can use the ALTER TYPE RESET statement to reset its version to 1, so that it is no longer considered to be evolved. (Resetting the version of an ADT to 1 invalidates its dependents.) For the syntax of the ALTER TYPE RESET statement, see *Oracle Database PL/SQL Language Reference*.

24.1.1.5 EDITIONABLE and NONEDITIONABLE Properties

Note: When a database is upgraded from Release 11.2 to Release 12.1, objects in user-created schemas get the EDITIONABLE property and public synonyms get the NONEDITIONABLE property.

The CREATE and ALTER statements for the schema object types that are editionable in the database let you specify that the object you are creating or altering is EDITIONABLE or NONEDITIONABLE.

The DBMS_SQL_TRANSLATOR.CREATE_PROFILE procedure lets you specify that the SQL translation profile that you are creating is EDITIONABLE or NONEDITIONABLE.

To see which objects are EDITIONABLE, see the EDITIONABLE column of the static data dictionary view *_OBJECTS (described in *Oracle Database Reference*) or *_OBJECTS_AE (described in *Oracle Database Reference*).

Topics:

- [Creating New EDITIONABLE and NONEDITIONABLE Objects](#)
- [Replacing or Altering EDITIONABLE and NONEDITIONABLE Objects](#)

See Also:

- *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the CREATE and ALTER statements for PL/SQL schema objects
- *Oracle Database SQL Language Reference* for the syntax and semantics of the CREATE and ALTER statements for SQL schema objects
- *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_SQL_TRANSLATOR.CREATE_PROFILE procedure

24.1.1.5.1 Creating New EDITIONABLE and NONEDITIONABLE Objects When you create a new schema object whose type is editionable in the database, you can specify the property EDITIONABLE or NONEDITIONABLE. If you omit the property, then the object is EDITIONABLE by default unless it is one of the following:

- PUBLIC SYNONYM, which is NONEDITABLE by default
- PACKAGE BODY, which inherits the property of the package specification
- TYPE BODY, which inherits the property of the type specification

For PACKAGE BODY or TYPE BODY, if you specify a property, then it must match the property of the corresponding package or type specification.

When you create an EDITIONABLE object of a type that is editionable in its schema, the new object is an editioned object that is visible only in the edition that is current when the object is created. (For information about the current edition, see [Section 24.1.6](#).) Creating an editioned object is a live operation with respect to the editions in which the new object is invisible.

When you create either an object with the NONEDITABLE property or an object whose type is noneditionable in its schema, the new object is a noneditioned object, which is visible to all editions.

Suppose that in the current edition, your schema has no schema object named `obj`, but in another edition, your schema has an editioned object named `obj`. You can create an object named `obj` in your schema in the current edition, but it must be an editioned object (that is, uniquely identified by its `OBJECT_NAME`, `OWNER`, and `EDITION_NAME`). The type of the new object (which can be different from the type of the existing editioned object with the same name) must be editionable in your schema and the new object must have the `EDITIONABLE` property. See [Example 24-2](#) and [Example 24-3](#).

24.1.1.5.2 Replacing or Altering EDITIONABLE and NONEDITABLE Objects When you replace or alter an existing object (with the `CREATE OR REPLACE` or `ALTER` statement):

- If the schema is not enabled for editions, then you can change the property of the object from `EDITIONABLE` to `NONEDITABLE`, or the reverse.
- If the schema is enabled for editions for the type of the object being replaced or altered, then you cannot change the property of the object from `EDITIONABLE` to `NONEDITABLE`, or the reverse.

Altering an editioned object is a live operation with respect to the editions in which the altered object is invisible.

24.1.1.6 Rules for Editioned Objects

- A noneditioned object usually cannot depend on an editioned object (for more information, see [Section 24.1.1](#)).
- An Abstract Data Type (ADT) cannot be both editioned and evolved.
For information about type evolution, see *Oracle Database Object-Relational Developer's Guide*.
- An editioned object cannot be the starting or ending point of a FOREIGN KEY constraint.

This rule affects only editioned views. An editioned view can be either an ordinary view or an editioning view.

24.1.2 Creating an Edition

Note: Oracle recommends against creating editions in the Root of a CDB. For information about CDBs, see *Oracle Database Administrator's Guide*.

To create an edition, use the SQL statement `CREATE EDITION`.

You must create the edition as the child of an existing edition. The parent of the first edition created with a `CREATE EDITION` statement is `ora$base`. This statement creates the edition `e2` as the child of `ora$base`:

```
CREATE EDITION e2
```

([Example 24-1](#) and others use the preceding statement.)

An edition can have at most one child.

The **descendents** of an edition are its child, its child's child, and so on. The **ancestors** of an edition are its parent, its parent's parent, and so on. The **root edition** has no parent, and a **leaf edition** has no child.

See Also: *Oracle Database SQL Language Reference* for information about the `CREATE EDITION` statement, including the privileges required to use it

24.1.3 Editioned Objects and Copy-on-Change

When you create an edition, all editioned objects in its parent edition are copied to it. Changes to an editioned object in one edition do not affect copies of that editioned object in other editions.

The preceding paragraph describes what happens conceptually. In practice, to optimize performance, Oracle Database copies an editioned object from an ancestor edition to a descendent edition only when the descendent edition changes the object. This strategy is called **copy-on-change**.

An editioned object that was conceptually (but not actually) copied to a descendent edition is called an **inherited object**. When a user of the descendent edition references an inherited object in a DDL statement, Oracle Database actually copies the object to the descendent edition. This copying operation is called **actualization**, and it creates an **actual object** in the descendent edition.

Note: There is one exception to the actualization rule in the preceding paragraph: When a `CREATE OR REPLACE` object statement replaces an inherited object with an identical object (that is, an object with the same source code and settings), Oracle Database *does not* create an actual object in the descendent edition.

[Example 24-1](#) creates a procedure named `hello` in the edition `ora$base`, and then creates the edition `e2` as a child of `ora$base`. When `e2` invokes `hello`, it invokes the inherited procedure in `ora$base`. Then `e2` changes `hello`, actualizing it. Now when `e2` invokes `hello`, it invokes its own actual procedure. The procedure `hello` in the edition `ora$base` remains unchanged.

Example 24–1 Editioned Objects and Copy-on-Change

1. Assume that this procedure is an editioned object in ora\$base:

```
CREATE OR REPLACE PROCEDURE hello IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, edition 1.');
END hello;
/
```

2. In ora\$base, invoke the procedure:

```
BEGIN hello(); END;
/
```

Result:

Hello, edition 1.

PL/SQL procedure successfully completed.

3. Create a child edition:

```
CREATE EDITION e2;
```

Conceptually, the procedure is copied to the child edition, and only the copy is visible in the child edition. The copy is an inherited object, not an actual object.

4. Use the child edition:

```
ALTER SESSION SET EDITION = e2;
```

For information about ALTER SESSION SET EDITION, see [Section 24.1.6.2](#).

5. Invoke the procedure:

```
BEGIN hello(); END;
/
```

Conceptually, the child edition invokes its own copy of the procedure (which is identical to the procedure in the parent edition, ora\$base). However, the child edition actually invokes the procedure in the parent edition. Result:

Hello, edition 1.

PL/SQL procedure successfully completed.

6. Change the procedure:

```
CREATE OR REPLACE PROCEDURE hello IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, edition 2.');
END hello;
/
```

Oracle Database actualizes the procedure in the child edition, and the change affects only the actual object in the child edition, not the procedure in the parent edition.

7. Invoke the procedure:

```
BEGIN hello(); END;
/
```

The child edition invokes its own actual procedure:

```
Hello, edition 2.
```

PL/SQL procedure successfully completed.

8. Return to the parent edition:

```
ALTER SESSION SET EDITION = ora$base;
```

9. Invoke the procedure and see that it has not changed:

```
BEGIN hello(); END;  
/
```

Result:

```
Hello, edition 1.
```

PL/SQL procedure successfully completed.

Example 24–2 creates a procedure named `goodbye` in the edition `ora$base`, and then creates edition `e2` as a child of `ora$base`. After `e2` drops `goodbye`, it can no longer invoke it, but `ora$base` can still invoke it. (For more information about the `DROP PROCEDURE` statement, including the privileges required to use it, see *Oracle Database PL/SQL Language Reference*.)

Example 24–2 Dropping an Editioned Object

1. Assume that this procedure is an editioned object in `ora$base`:

```
CREATE OR REPLACE PROCEDURE goodbye IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE('Good-bye! ');\n    END goodbye;  
/
```

2. Invoke the procedure:

```
BEGIN goodbye; END;  
/
```

Result:

```
Good-bye!
```

PL/SQL procedure successfully completed.

3. Create edition `e2` as a child of `ora$base`:

```
CREATE EDITION e2;
```

In `e2`, the procedure is an inherited object.

4. Use edition `e2`:

```
ALTER SESSION SET EDITION = e2;
```

ALTER SESSION SET EDITION must be a top-level SQL statement. For more information, see [Section 24.1.6.2](#).

5. In `e2`, invoke the procedure:

```
BEGIN goodbye; END;  
/
```

e2 invokes the procedure in ora\$base:

Good-bye!

PL/SQL procedure successfully completed.

6. In e2, drop the procedure:

```
DROP PROCEDURE goodbye;
```

7. In e2, try to invoke the dropped procedure:

```
BEGIN goodbye; END;
/
```

Result:

```
BEGIN goodbye; END;
*
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00201: identifier 'GOODBYE' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

8. Return to ora\$base:

```
ALTER SESSION SET EDITION = ora$base;
```

9. In ora\$base, invoke the procedure:

```
BEGIN goodbye; END;
/
```

Result:

Good-bye!

PL/SQL procedure successfully completed.

Because e2 dropped the procedure goodbye:

- Its descendants do not inherit the procedure goodbye.
- No object named goodbye is visible in e2, so e2 can create an object named goodbye, but it must be an editioned object (for details, see [Section 24.1.5.1](#)). If e2 creates a new editioned object named goodbye, then the descendants of e2 inherit that object.

In [Example 24-3](#), e2 creates a function named goodbye and then an edition named e3 as a child of e2. When e3 tries to invoke the *procedure* goodbye (which e2 dropped), an error occurs, but e3 successfully invokes the *function* goodbye (which e2 created).

Example 24-3 Creating an Object with the Name of a Dropped Inherited Object

1. Return to e2:

```
ALTER SESSION SET EDITION = e2;
```

For information about ALTER SESSION SET EDITION, see [Section 24.1.6.2](#).

2. In e2, create a function named goodbye:

```
CREATE OR REPLACE FUNCTION goodbye
  RETURN BOOLEAN
```

```

IS
BEGIN
  RETURN(TRUE);
END goodbye;
/

```

This function must be an editioned object. It has the EDITIONABLE property by default. If the type FUNCTION is not editionable in the schema, then you must use the ALTER USER statement to make it so (for instructions, see [Section 24.1.1.4](#)).

3. Create edition e3:

```
CREATE EDITION e3 AS CHILD OF e2;
```

Edition e3 inherits the function goodbye.

4. Use edition e3:

```
ALTER SESSION SET EDITION = e3;
```

5. In e3, try to invoke the *procedure* goodbye:

```

BEGIN
  goodbye;
END;
/

```

Result:

```

ERROR at line 2:
ORA-06550: line 2, column 3:
PLS-00306: wrong number or types of arguments in call to 'GOODBYE'
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored

```

6. In e3, invoke *function* goodbye:

```

BEGIN
  IF goodbye THEN
    DBMS_OUTPUT.PUT_LINE('Good-bye! ');
  END IF;
END;
/

```

Result:

```

Good-bye!
PL/SQL procedure successfully completed.

```

24.1.4 Making an Edition Available to Some Users

As the creator of the edition, you automatically have the USE privilege WITH GRANT OPTION on it. To grant the USE privilege on the edition to other users, use the SQL statement GRANT USE ON EDITION. For information about the GRANT statement, see *Oracle Database SQL Language Reference*.

24.1.5 Making an Edition Available to All Users

To make an edition available to all users, either:

- Grant the USE privilege on the edition to PUBLIC:

```
GRANT USE ON EDITION edition_name TO PUBLIC
```

For information about the GRANT statement, see *Oracle Database SQL Language Reference*.

- Make the edition the database default edition:

```
ALTER DATABASE DEFAULT EDITION = edition_name
```

This has the side effect of allowing all users to use the edition, because it effectively grants the USE privilege on *edition_name* to PUBLIC.

For information about the ALTER DATABASE statement, see *Oracle Database SQL Language Reference*.

24.1.6 Current Edition and Session Edition

Each database session uses exactly one edition at a time. The edition that a database session is using at any one time is called its **current edition**. When a database session begins, its current edition is its **session edition**, which is the edition in which it begins. If you change the session edition, the current edition changes to the same thing. However, there are situations in which the current edition and session edition differ.

Topics:

- [Your Initial Session Edition](#)
- [Changing Your Session Edition](#)
- [Displaying the Names of the Current and Session Editions](#)
- [When the Current Edition Might Differ from the Session Edition](#)

24.1.6.1 Your Initial Session Edition

When you connect to the database, you can specify your initial session edition. Your initial session edition can be the database default edition or any edition on which you have the USE privilege. To see the names of the editions that are available to you, use this query:

```
SELECT EDITION_NAME FROM ALL_EDITIONS;
```

How you specify your initial session edition at connection time depends on how you connect to the database—see the documentation for your interface.

See Also:

- *Oracle Database Administrator's Guide* for information about setting the database default edition
- *SQL*Plus User's Guide and Reference* for information about connecting to the database with SQL*Plus
- *Oracle Call Interface Programmer's Guide* for information about connecting to the database with Oracle Call Interface (OCI)
- *Oracle Database JDBC Developer's Guide* for information about connecting to the database with JDBC

As of Oracle Database 11g Release 2 (11.2.0.2), if you do not specify your session edition at connection time, then:

- If you use a database service to connect to the database, and an initial session edition was specified for that service, then the initial session edition for the service is your initial session edition.
- Otherwise, your initial session edition is the database default edition.

As of Release 11.2.0.2, when you create or modify a database service, you can specify its initial session edition.

To create or modify a database service, Oracle recommends using the `srvctl add service` or `srvctl modify service` command. To specify the default initial session edition of the service, use the `-edition` option.

Alternatively, you can create or modify a database service with the `DBMS_SERVICE.CREATE_SERVICE` or `DBMS_SERVICE.MODIFY_SERVICE` procedure, and specify the default initial session edition of the service with the `EDITION` attribute.

Note: As of Oracle Database 11g Release 2 (11.2.0.1), the `DBMS_SERVICE.CREATE_SERVICE` and `DBMS_SERVICE.MODIFY_SERVICE` procedures are deprecated in databases managed by Oracle Clusterware and Oracle Restart.

See Also:

- *Oracle Database Administrator's Guide* for information about the `-edition` option of the `srvctl add service` command
- *Oracle Database Administrator's Guide* for information about the `-edition` option of the `srvctl modify service` command
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `EDITION` attribute of the `DBMS_SERVICE.CREATE_SERVICE` procedure
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `EDITION` attribute of the `DBMS_SERVICE.MODIFY_SERVICE` procedure

24.1.6.2 Changing Your Session Edition

After connecting to the database, you can change your session edition with the SQL statement `ALTER SESSION SET EDITION`. You can change your session edition to the database default edition or any edition on which you have the `USE` privilege. When you change your session edition, your current edition changes to that same edition.

These statements from [Example 24-1](#) and [Example 24-2](#) change the session edition (and current edition) first to `e2` and later to `ora$base`:

```
ALTER SESSION SET EDITION = e2
...
ALTER SESSION SET EDITION = ora$base
```

Note: `ALTER SESSION SET EDITION` must be a top-level SQL statement. To defer an edition change (in a logon trigger, for example), use the `DBMS_SESSION.SET_EDITION_DEFERRED` procedure.

See Also:

- *Oracle Database SQL Language Reference* for more information about the ALTER SESSION SET EDITION statement
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_SESSION.SET_EDITION_DEFERRED procedure

24.1.6.3 Displaying the Names of the Current and Session Editions

This statement returns the name of the current edition:

```
SELECT SYS_CONTEXT('USERENV', 'CURRENT_EDITION_NAME') FROM DUAL;
```

This statement returns the name of the session edition:

```
SELECT SYS_CONTEXT('USERENV', 'SESSION_EDITION_NAME') FROM DUAL;
```

See Also: *Oracle Database SQL Language Reference* for more information about the SYS_CONTEXT function

24.1.6.4 When the Current Edition Might Differ from the Session Edition

The current edition might differ from the session edition in these situations:

- A crossedition trigger fires.
For details, see [Section 24.3.3, "Crossedition Trigger Interaction with Editions."](#)
- You run a statement by calling the DBMS_SQLPARSE procedure, specifying the edition in which the statement is to run, as in [Example 24-4](#).

While the statement is running, the current edition is the specified edition, but the session edition does not change. For information about the DBMS_SQLPARSE procedure, see *Oracle Database PL/SQL Packages and Types Reference*.

[Example 24-4](#) creates a function that returns the names of the session edition and current edition. Then it creates a child edition, which invokes the function twice. The first time, the session edition and current edition are the same. The second time, they are not, because a different edition is passed as a parameter to the DBMS_SQLPARSE procedure.

Example 24-4 Current Edition Differs from Session Edition

1. Create function that returns the names of the session edition and current edition:

```
CREATE OR REPLACE FUNCTION session_and_current_editions
  RETURN VARCHAR2
IS
BEGIN
  RETURN
    'Session: ' || SYS_CONTEXT('USERENV', 'SESSION_EDITION_NAME') || 
    ' / ' ||
    'Current: ' || SYS_CONTEXT('USERENV', 'CURRENT_EDITION_NAME');
END session_and_current_editions;
/
```

2. Create child edition:

```
CREATE EDITION e2 AS CHILD OF ora$base;
```

3. Use child edition:

```
ALTER SESSION SET EDITION = e2;
```

4. Invoke function:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE (session_and_current_editions());
END;
/
```

Result:

```
Session: E2 / Current: E2
```

PL/SQL procedure successfully completed.

5. Invoke function again:

```
DECLARE
  c      NUMBER := DBMS_SQL.OPEN_CURSOR();
  v      VARCHAR2(200);
  dummy NUMBER;
  stmt  CONSTANT VARCHAR2(32767)
        := 'SELECT session_and_current_editions() FROM DUAL';
BEGIN
  DBMS_SQLPARSE (c => c,
                  statement => stmt,
                  language_flag => DBMS_SQL.NATIVE,
                  edition => 'ora$base');

  DBMS_SQL.DEFINE_COLUMN (c, 1, v, 200);
  dummy := DBMS_SQL.EXECUTE_AND_FETCH (c, true);
  DBMS_SQL.COLUMN_VALUE (c, 1, v);
  DBMS_SQL.CLOSE_CURSOR(c);
  DBMS_OUTPUT.PUT_LINE (v);
END;
/
```

Result:

```
Session: E2 / Current: ORA$BASE
```

PL/SQL procedure successfully completed.

24.1.7 Retiring an Edition

After making a new edition (an upgraded application) available to all users, retire the old edition (the original application), so that no user except `SYS` can use the old edition.

Note: If the old edition is the database default edition, make another edition the database default edition before you retire the old edition:

```
ALTER DATABASE DEFAULT EDITION = edition_name
```

For information about the `ALTER DATABASE` statement, see *Oracle Database SQL Language Reference*.

To retire an edition, you must revoke the `USE` privilege on the edition from every grantee. To list the grantees, use this query, where `:e` is a placeholder for the name of the edition to be dropped:

```
SELECT GRANTEE, PRIVILEGE
  FROM DBA_TAB_PRIVS
 WHERE TABLE_NAME = :e
  /
```

For information about the REVOKE statement, see *Oracle Database SQL Language Reference*.

When you retire an edition, update the evaluation editions and unusable editions of noneditioned objects accordingly. For information about changing evaluation editions and unused editions, see [Section 24.1.1.2, "Noneditioned Objects That Can Depend on Editioned Objects."](#)

24.1.8 Dropping an Edition

Note: If the edition includes crossedition triggers, see [Section 24.3.6, "Dropping the Crossedition Triggers,"](#) before you drop the edition.

To drop an edition, use the `DROP EDITION` statement, described in *Oracle Database SQL Language Reference*. If the edition has actual objects, you must specify the `CASCADE` clause, which drops the actual objects.

If a `DROP EDITION edition CASCADE` statement is interrupted before finishing normally (from a power failure, for example), the static data dictionary view `*_EDITIONS` shows that the value of `USABLE` for `edition` is `NO`. The only operation that you can perform on such an unusable `edition` is `DROP EDITION CASCADE`.

You drop an edition in these situations:

- You want to roll back the application upgrade.
- (Optional) You have retired the edition.

You can drop an edition only if all of these statements are true:

- The edition is either the root edition or a leaf edition.
- If the edition is the root, it has no objects that its descendants inherit. (That is, each object inherited from the root edition was either actualized or dropped.)
- The edition is not in use (that is, it is not the current edition or session edition of a session).
- The edition is not the database default edition.

Note: After you have dropped an edition, you cannot recompile a noneditioned object that depends on an editioned object if both of the following are true:

- The noneditioned object contains an ADT.
 - The noneditioned object was invalidated when the owner of the editioned object on which it depends was enabled for editions using `FORCE`.
-

To explicitly actualize an inherited object in the child edition:

1. Make the child edition your session edition.

For instructions, see [Section 24.1.6.2](#).

2. Recompile the object, using the appropriate ALTER statement with COMPILE. For a PL/SQL object, also specify REUSE SETTINGS.

For example, this statement actualizes the procedure p1:

```
ALTER PROCEDURE p1 COMPILE REUSE SETTINGS
```

For information about the ALTER statements for PL/SQL objects, see *Oracle Database PL/SQL Language Reference*.

For information about the ALTER statements for SQL objects, see *Oracle Database SQL Language Reference*.

When you drop an edition, update the evaluation editions and unusable editions of noneditioned objects accordingly. For information about changing evaluation editions and unused editions, see [Section 24.1.1.2, "Noneditioned Objects That Can Depend on Editioned Objects."](#)

See Also:

- *Oracle Database SQL Language Reference* for information about the ALTER LIBRARY statement
- *Oracle Database SQL Language Reference* for information about the ALTER VIEW statement
- *Oracle Database PL/SQL Language Reference* for information about the ALTER FUNCTION statement
- *Oracle Database PL/SQL Language Reference* for information about the ALTER PACKAGE statement
- *Oracle Database PL/SQL Language Reference* for information about the ALTER PROCEDURE statement
- *Oracle Database PL/SQL Language Reference* for information about the ALTER TRIGGER statement
- *Oracle Database PL/SQL Language Reference* for information about the ALTER TYPE statement

24.2 Editioning Views

On a noneditioning view, the only type of trigger that you can define is an INSTEAD OF trigger. On an editioning view, you can define every type of trigger that you can define on a table (except crossedition triggers, which are temporary, and INSTEAD OF triggers). Therefore, and because they can be editioned, editioning views let you treat their base tables as if the base tables were editioned. However, you cannot add indexes or constraints to an editioning view; if your upgraded application requires new indexes or constraints, you must add them to the base table.

Note: If you will change a base table or an index on a base table, then see ["Nonblocking and Blocking DDL Statements"](#) on page 6-31.

An editioning view selects a subset of the columns from a single base table and, optionally, provides aliases for them. In providing aliases, the editioning view maps physical column names (used by the base table) to logical column names (used by the application). An editioning view is like an API for a table.

There is no performance penalty for accessing a table through an editioning view, rather than directly. That is, if a SQL SELECT, INSERT, UPDATE, DELETE, or MERGE statement uses one or more editioning views, one or more times, and you replace each editioning view name with the name of its base table and adjust the column names if necessary, performance does not change.

The static data dictionary view `*_EDITIONING_VIEWS` describes every editioning view in the database that is visible in the session edition. `*_EDITIONING_VIEWS_AE` describes every actual object in every editioning view in the database, in every edition.

Topics:

- [Creating an Editioning View](#)
- [Partition-Extended Editioning View Names](#)
- [Changing the Writability of an Editioning View](#)
- [Replacing an Editioning View](#)
- [Dropping or Renaming the Base Table](#)
- [Adding Indexes and Constraints to the Base Table](#)
- [SQL Optimizer Index Hints](#)

See Also: *Oracle Database Reference* for more information about the static data dictionary views `*_EDITIONING_VIEWS` and `*_EDITIONING_VIEWS_AE`.

24.2.1 Creating an Editioning View

Before an editioning view is created, its owner must be editions-enabled and the schema object type `VIEW` must be editionable in its owner. (For instructions, see [Section 24.1.1.4](#).)

To create an editioning view, use the SQL statement `CREATE VIEW` with the keyword `EDITIONING`. To make the editioning view read-only, specify `WITH READ ONLY`; to make it read-write, omit `WITH READ ONLY`. Do not specify `NONEDITABLE`, or an error occurs.

If an editioning view is **read-only**, users of the unchanged application can see the data in the base table, but cannot change it. The base table has **semi-availability**.

Semi-availability is acceptable for applications such as online dictionaries, which users read but do not change. Make the editioning view read-only if you do not define crossedition triggers on the base table.

If an editioning view is **read-write**, users of the unchanged application can both see and change the data in the base table. The base table has **maximum availability**. Maximum availability is required for applications such as online stores, where users submit purchase orders. If you define crossedition triggers on the base table, make the editioning view read-write.

Because an editioning view must do no more than select a subset of the columns from the base table and provide aliases for them, the `CREATE VIEW` statement that creates an editioning view has restrictions. Violating the restrictions causes the creation of the view to fail, even if you specify `FORCE`.

See Also: *Oracle Database SQL Language Reference* for more information about using the `CREATE VIEW` statement to create editioning views, including the restrictions

24.2.2 Partition-Extended Editioning View Names

An editioning view defined on a partitioned table can have a partition-extended name, with partition and subpartition names that refer to the partitions and subpartitions of the base table.

The data manipulation language (DML) statements that support partition-extended table names also support partition-extended editioning view names. These statements are:

- DELETE
- INSERT
- SELECT
- UPDATE

See Also: *Oracle Database SQL Language Reference* for information about referring to partitioned tables

24.2.3 Changing the Writability of an Editioning View

To change an existing editioning view from read-only to read-write, use the SQL statement `ALTER VIEW READ WRITE`. To change an existing editioning view from read-write to read-only, use the SQL statement `ALTER VIEW READ ONLY`.

See Also: *Oracle Database SQL Language Reference* for more information about the `ALTER VIEW` statement

24.2.4 Replacing an Editioning View

To replace an editioning view, use the SQL statement `CREATE VIEW` with the `OR REPLACE` clause and the keyword `EDITIONING`.

You can replace an editioning view only with another editioning view. Any triggers defined on the replaced editioning view are retained.

24.2.5 Dropping or Renaming the Base Table

If you drop or rename the base table on which an editioning view is defined, the editioning view is not dropped, but the editioning view and its dependents become invalid. However, any triggers defined on the editioning view remain.

24.2.6 Adding Indexes and Constraints to the Base Table

If your upgraded application requires new indexes or constraints, you must add them to the base table. You cannot add them to the editioning view.

If the new indexes might negatively impact the old edition (the original application), make them invisible. In the crossedition triggers that must use the new indexes, specify them in `INDEX` hints.

When all users are using only the upgraded application:

- If the new indexes were used only by the crossedition triggers, drop them.
- If the new indexes are helpful in the upgraded application, make them visible.

See Also:

- [Section 9.1, "Guidelines for Managing Indexes"](#)
- [Oracle Database SQL Language Reference](#) for information about INDEX hints
- [Section 24.2.7, "SQL Optimizer Index Hints"](#)

24.2.7 SQL Optimizer Index Hints

SQL optimizer index hints are specified in terms of the logical names of the columns participating in the index. Any SQL optimizer index hints specified on an editioning view using logical column names must be mapped to an index on the corresponding physical column in the base table.

See Also: [Oracle Database SQL Language Reference](#) for information about using hints

24.3 Crossedition Triggers

The most important difference between crossedition triggers and noncrossedition triggers is how they interact with editions. A crossedition trigger is visible only in the edition in which it is actual, never in a descendent edition. Forward crossedition triggers move data from columns used by the old edition to columns used by the new edition; reverse crossedition triggers do the reverse.

Other important differences are:

- Crossedition triggers can be ordered with triggers defined on other tables, while noncrossedition triggers can be ordered only with other triggers defined on the same table.
- Crossedition triggers are temporary—you drop them after you have made the restructured tables available to all users.

Topics:

- [Forward Crossedition Triggers](#)
- [Reverse Crossedition Triggers](#)
- [Crossedition Trigger Interaction with Editions](#)
- [Creating a Crossedition Trigger](#)
- [Transforming Data from Pre- to Post-Upgrade Representation](#)
- [Dropping the Crossedition Triggers](#)

See Also: [Oracle Database PL/SQL Language Reference](#) for general information about triggers

24.3.1 Forward Crossedition Triggers

The DML changes that you make to the table in the post-upgrade edition are written only to new columns or new tables, never to columns that users of pre-upgrade (ancestor) editions might be reading or writing. However, if the user of an ancestor edition changes the table data, the editioning view that you see must accurately reflect these changes. This is accomplished with forward crossedition triggers.

A forward crossedition trigger defines a **transform**, which is a rule for transforming an old row to one or more new rows. An **old row** is a row of data in the pre-upgrade representation. A **new row** is a row of data in the post-upgrade representation. The name of the trigger refers to the trigger itself and to the transform that the trigger defines.

24.3.2 Reverse Crossedition Triggers

If the pre- and post-upgrade editions will be in ordinary use at the same time (hot rollover), use reverse crossedition triggers to ensure that when users of the post-upgrade edition make changes to the table data, the changes are accurately reflected in the pre-upgrade editions.

24.3.3 Crossedition Trigger Interaction with Editions

The most important difference between crossedition triggers and noncrossedition triggers is how they interact with editions.

In this topic, the **current edition** is the edition in which the triggering DML statement runs. The current edition might differ from the session edition (for details, see [Section 24.1.6.4](#)).

Topics:

- [Which Triggers Are Visible](#)
- [What Kind of Triggers Can Fire](#)
- [Firing Order](#)
- [Crossedition Trigger Execution](#)

24.3.3.1 Which Triggers Are Visible

Editions inherit noncrossedition triggers in the same way that they inherit other editioned objects (see [Section 24.1.3, "Editioned Objects and Copy-on-Change"](#)).

Editions do not inherit crossedition triggers. A crossedition trigger might fire in response to a DML statement that another edition runs, but its name is visible only in the edition in which it was created. Therefore, an edition can reuse the name of a crossedition trigger created in an ancestor edition. Reusing the name of a crossedition trigger does not change the conditions under which the older trigger fires.

Crossedition triggers that appear in static data dictionary views are actual objects in the current edition.

24.3.3.2 What Kind of Triggers Can Fire

What kind of triggers can fire depends on the category of the triggering DML statement.

Categories:

- [Section 24.3.3.2.1, "Forward Crossedition Trigger SQL"](#)
- [Section 24.3.3.2.2, "Reverse Crossedition Trigger SQL"](#)
- [Section 24.3.3.2.3, "Application SQL"](#)

Note: The APPEND hint on a SQL INSERT statement does not prevent crossedition triggers from firing. For information about the APPEND hint, see *Oracle Database SQL Language Reference*.

24.3.3.2.1 Forward Crossedition Trigger SQL

Forward crossedition trigger SQL is SQL that is executed in either of these ways:

- Directly from the body of a forward crossedition trigger

This category includes SQL in an invoked subprogram only if the subprogram is local to the forward crossedition trigger.

- By invoking the DBMS_SQLPARSE procedure with a non-NULL value for the apply_crossedition_trigger parameter

The only valid non-NULL value for the apply_crossedition_trigger parameter is the unqualified name of a forward crossedition trigger. For more information about the DBMS_SQLPARSE procedure, see *Oracle Database PL/SQL Packages and Types Reference*.

If a forward crossedition trigger invokes a subprogram in another compilation unit, the SQL in the subprogram is forward crossedition trigger SQL only if it is invoked by the DBMS_SQLPARSE procedure with a non-NULL value for the apply_crossedition_trigger parameter.

Forward crossedition trigger SQL can fire only triggers that satisfy all of these conditions:

- They are forward crossedition triggers.
- They were created either in the current edition or in a descendent of the current edition.
- They explicitly follow the running forward crossedition trigger.

24.3.3.2.2 Reverse Crossedition Trigger SQL

Reverse crossedition trigger SQL is SQL that is executed directly from the body of a reverse crossedition trigger. This category includes SQL in an invoked subprogram only if the subprogram is local to the reverse crossedition trigger.

Reverse crossedition trigger SQL can fire only triggers that satisfy all of these conditions:

- They are reverse crossedition triggers.
- They were created either in the current edition or in an ancestor of the current edition.
- They explicitly precede the running reverse crossedition trigger.

24.3.3.2.3 Application SQL

Application SQL is all SQL except crossedition trigger SQL, including these DML statements:

- Dynamic SQL DML statements coded with the DBMS_SQL package (for information about these statements, see *Oracle Database PL/SQL Language Reference*).
- DML statements executed by Java stored procedures and external procedures (even when these procedures are invoked by CALL triggers)

Application SQL fires both noncrossedition and crossedition triggers, according to these rules:

Kind of Trigger	Conditions Under Which Trigger Can Fire
Noncrossedition	Trigger is both visible and enabled in the current edition.
Forward crossedition	Trigger was created in a descendent of the current edition.
Reverse crossedition	Trigger was created either in the current edition or in an ancestor of the current edition.

24.3.3.3 Firing Order

For a trigger to fire in response to a specific DML statement, the trigger must:

- Be the right kind (see [Section 24.3.3.2, "What Kind of Triggers Can Fire"](#))
- Satisfy the selection criteria (for example, the type of DML statement and the `WHEN` clause)
- Be enabled

For the triggers that meet these requirements, firing order depends on the `FOLLOWS` and `PRECEDES` clauses, the trigger type, and the edition.

Topics:

- [FOLLOWS and PRECEDES Clauses](#)
- [Trigger Type and Edition](#)

See Also: *Oracle Database PL/SQL Language Reference* for general information about trigger firing order

24.3.3.3.1 FOLLOWS and PRECEDES Clauses When triggers A and B are to be fired at the same timing point, A fires before B fires if either of these is true:

- A explicitly precedes B.
- B explicitly follows A.

This rule is independent of conditions such as:

- Whether the triggers are enabled or disabled
- Whether the columns specified in the `UPDATE OF` clause are modified
- Whether the `WHEN` clauses are satisfied
- Whether the triggers are associated with the same kinds of DML statements (`INSERT`, `UPDATE`, or `DELETE`)
- Whether the triggers have overlapping timing points

The firing order of triggers that do not explicitly follow or precede each other is unpredictable.

24.3.3.3.2 Trigger Type and Edition For each timing point associated with a triggering DML statement, eligible triggers fire in this order. In categories 1 through 3, `FOLLOWS` relationships apply; in categories 4 and 5, `PRECEDES` relationships apply.

1. Noncrossedition triggers
2. Forward crossedition triggers created in the current edition
3. Forward crossedition triggers created in descendants of the current edition, in the order that the descendants were created (child, grandchild, and so on)

4. Reverse crossedition triggers created in the current edition
5. Reverse crossedition triggers created in the ancestors of the current edition, in the reverse order that the ancestors were created (parent, grandparent, and so on)

24.3.3.4 Crossedition Trigger Execution

A crossedition trigger runs using the edition in which it was created. Any code that the crossedition trigger calls (including package references, PL/SQL subprogram calls, and SQL statements) also runs in the edition in which the crossedition trigger was created.

If a PL/SQL package is actual in multiple editions, then the package variables and other state are private in each edition, even within a single session. Because each crossedition trigger and the code that it calls run using the edition in which the crossedition trigger was created, the same session can instantiate two or more versions of the package, with the same name.

24.3.4 Creating a Crossedition Trigger

Before a crossedition trigger is created, its owner must be editions-enabled and the schema object type TRIGGER must be editionable in its owner. (For instructions, see [Section 24.1.1.4](#).)

Create a crossedition trigger with the SQL statement `CREATE TRIGGER`, observing these rules:

- A crossedition trigger must be defined on a table, not a view.
- A crossedition trigger must have the `EDITIONABLE` property.
- A crossedition trigger must be a DML trigger (simple or compound).
The DML statement in a crossedition trigger body can be either a static SQL statement (described in *Oracle Database PL/SQL Language Reference*) or a native dynamic SQL statement (described in *Oracle Database PL/SQL Language Reference*).
- A crossedition trigger is forward unless you specify `REVERSE`. (Specifying `FORWARD` is optional.)
- The `FOLLOWS` clause is allowed only when creating a forward crossedition trigger or a noncrossedition trigger. (The `FOLLOWS` clause indicates that the trigger being created is to fire after the specified triggers fire.)
- The `PRECEDES` clause is allowed only when creating a reverse crossedition trigger. (The `PRECEDES` clause indicates that the trigger being created is to fire before the specified triggers fire.)
- The triggers specified in the `FOLLOWS` or `PRECEDES` clause must exist, but need not be enabled or successfully compiled.
- Like a noncrossedition trigger, a crossedition trigger is created in the enabled state unless you specify `DISABLE`. (Specifying `ENABLE` is optional.)

Tip: Create crossedition triggers in the disabled state, and enable them after you are sure that they compile successfully. If you create them in the enabled state, and they fail to compile, the failure affects users of the existing application.

- The operation in a crossedition trigger body must be **idempotent** (that is, performing the operation multiple times is redundant; it does not change the result).

See Also: *Oracle Database PL/SQL Language Reference* for more information about using the CREATE TRIGGER statement to create crossedition triggers

24.3.4.1 Coding the Forward Crossedition Trigger Body

The operation in the body of a forward crossedition trigger must be idempotent, because it is impossible to predict:

- The context in which the body will first run for an old row.
The possibilities are:
 - When a user of an ancestor edition runs a DML statement that fires the trigger (a **serendipitous change**)
 - When you **apply the transform** that the trigger defines
For information about applying transforms, see [Section 24.3.5](#).
- How many times the body will run for each old row.

Topics:

- [Handling Data Transformation Collisions](#)
- [Handling Changes to Other Tables](#)

24.3.4.1.1 Handling Data Transformation Collisions If a forward crossedition trigger populates a new table (rather than new columns of a table), its body must handle data transformation collisions.

For example, suppose that a column of the new table has a **UNIQUE** constraint. A serendipitous change fires the forward crossedition trigger, which inserts a row in the new table. Later, another serendipitous change fires the forward crossedition trigger, or you apply the transform defined by the trigger. The trigger tries to insert a row in the new table, violating the **UNIQUE** constraint.

If your collision-handling strategy depends on why the trigger is running, you can determine the reason with the function **APPLYING_CROSSEDITION_TRIGGER**. When called directly from a trigger body, this function returns the **BOOLEAN** value **TRUE** if the trigger is running because of a serendipitous change and **FALSE** if the trigger is running because you are applying the transform. (**APPLYING_CROSSEDITION_TRIGGER** is defined in the package **DBMS_STANDARD**. It has no parameters.)

To ignore collisions and insert the rows that do not collide with existing rows, put the **IGNORE_ROW_ON_DUPKEY_INDEX** hint in the **INSERT** statement.

If you do not want to ignore such collisions, but want to know where they occur so that you can handle them, put the **CHANGE_DUPKEY_ERROR_INDEX** hint in the **INSERT** or **UPDATE** statement, specifying either an index or set of columns. Then, when a unique key violation occurs for that index or set of columns, **ORA-38911** is reported instead of **ORA-00001**. You can write an exception handler for **ORA-38911**.

Note: Although they have the syntax of hints, **IGNORE_ROW_ON_DUPKEY_INDEX** and **CHANGE_DUPKEY_ERROR_INDEX** are mandates. The optimizer always uses them.

[Example 24-5](#) creates a crossedition trigger that uses the **APPLYING_CROSSEDITION_TRIGGER** function and the **IGNORE_ROW_ON_DUPKEY_INDEX** and **CHANGE_DUPKEY_ERROR_INDEX** hints.

INDEX hints to handle data transformation collisions. The trigger transforms old rows in table1 to new rows in table2. The tables were created as follows:

```
CREATE TABLE table1 (key NUMBER, value VARCHAR2(20));

CREATE TABLE table2 (key NUMBER, value VARCHAR2(20), last_updated TIMESTAMP);
CREATE UNIQUE INDEX i2 on table2(key);
```

Example 24–5 Crossedition Trigger that Handles Data Transformation Collisions

```
CREATE OR REPLACE TRIGGER trigger1
  BEFORE INSERT OR UPDATE ON table1
  FOR EACH ROW
  CROSSEDITITION
DECLARE
  row_already_present EXCEPTION;
  PRAGMA EXCEPTION_INIT(row_already_present, -38911);
BEGIN
  IF APPLYING_CROSSEDITITION_TRIGGER THEN
    /* Trigger is running because of serendipitous change.
       Insert new row into table2 unless it is already there. */
    INSERT /*+ IGNORE_ROW_ON_DUPKEY_INDEX(table2(key)) */
    INTO table2
    VALUES(:new.key, :new.value, to_date('1900-01-01', 'YYYY-MM-DD'));
  ELSE
    /* Trigger is running because you are applying transform.
       If transform has not yet inserted new row in table2, insert new row;
       otherwise, update new row. */
    BEGIN
      INSERT /*+ CHANGE_DUPKEY_ERROR_INDEX(table2(key)) */
      INTO table2
      VALUES(:new.key, :new.value, SYSTIMESTAMP);
    EXCEPTION WHEN row_already_present THEN
      UPDATE table2
      SET value = :new.value, last_updated = SYSTIMESTAMP
      WHERE key = :new.key;
    END;
  END IF;
END;
/
```

See Also:

- *Oracle Database SQL Language Reference* for more information about `IGNORE_ROW_ON_DUPKEY_INDEX`
- *Oracle Database SQL Language Reference* for more information about `CHANGE_DUPKEY_ERROR_INDEX`
- *Oracle Database SQL Language Reference* for general information about hints

24.3.4.1.2 Handling Changes to Other Tables If the body of a forward crossedition trigger includes explicit SQL statements that change tables other than the one on which the trigger is defined, and if the rows of those tables do not have a one-to-one correspondence with the rows of the table on which the trigger is defined, then the body code must implement a locking mechanism that correctly handles these situations:

- Two or more users of ancestor editions simultaneously issue DML statements for the table on which the trigger is defined.

- At least one user of an ancestor edition issues a DML statement for the table on which the trigger is defined.

24.3.5 Transforming Data from Pre- to Post-Upgrade Representation

After redefining the database objects that comprise the application that you are upgrading (in the new edition), you must transform the application data from its pre-upgrade representation (in the old edition) to its post-upgrade representation (in the new edition). The rules for this transformation are called **transforms**, and they are defined by forward crossedition triggers. (For general information about forward crossedition triggers, see [Section 24.3.1](#).)

Some old rows might have been transformed to new rows by **serendipitous changes**; that is, by changes that users of the pre-upgrade application made, which fired forward crossedition triggers. However, any rows that were not transformed by serendipitous changes are still in their pre-upgrade representation. To ensure that all old rows are transformed to new rows, you must **apply the transforms** that you defined on the tables that store the application data.

There are three ways to apply a transform:

- Fire the trigger that defines the transform on every row of the table, one row at a time.
- Instead of firing the trigger, run a SQL statement that does what the trigger would do, but faster, and then fire any triggers that follow that trigger.

This second way is recommended over the first way if you have replaced an entire table or created a new table.

- Invoke the procedure `DBMS_EDITIONS_UTILITIES.SET_NULL_COLUMN_VALUES_TO_EXPR` to use a metadata operation to apply the transform to the new column.

This third way has the fastest installation time, but there are restrictions on the expression that represents the transform, and queries of the new column are slower until the metadata is replaced by actual data.

Metadata is replaced by actual data:

- In an individual column element that is updated.
- In every element of a column whose table is "compacted" using online table redefinition.

For the syntax, semantics, and restrictions of the `DBMS_EDITIONS_UTILITIES.SET_NULL_COLUMN_VALUES_TO_EXPR` procedure, see *Oracle Database PL/SQL Packages and Types Reference*.

For the first two ways of applying the transform, invoke either the `DBMS_SQLPARSE` procedure or the subprograms in the `DBMS_PARALLEL_EXECUTE` package. The latter is recommended if you have a lot of data. The subprograms enable you to incrementally update the data in a large table in parallel, in two high-level steps:

1. Group sets of rows in the table into smaller chunks.
2. Apply the desired `UPDATE` statement to the chunks in parallel, committing each time you have finished processing a chunk.

The advantages are:

- You lock only one set of rows at a time, for a relatively short time, instead of locking the entire table.

- You do not lose work that has been done if something fails before the entire operation finishes.

For both the DBMS_SQLPARSE procedure and the DBMS_PARALLEL_EXECUTE subprograms, the actual parameter values for apply_crossedition_trigger, fire_apply_trigger, and sql_stmt are the same:

- For apply_crossedition_trigger, specify the name of the forward crossedition trigger that defines the transform to be applied.
- To fire the trigger on every row of the table, one row at a time:
 - For the value of fire_apply_trigger, specify TRUE.
 - For sql_stmt, supply a SQL statement whose only significant effect is to select the forward crossedition trigger to be fired; for example, an UPDATE statement that sets some column to its own existing value in each row.
- To run a SQL statement that does what the trigger would do, and then fire any triggers that follow that trigger:
 - For the value of fire_apply_trigger, specify FALSE.
 - For sql_stmt, supply a SQL statement that does what the forward crossedition trigger would do, but faster—for example, a PL/SQL anonymous block that calls one or more PL/SQL subprograms.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_SQLPARSE procedure
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_PARALLEL_EXECUTE package

24.3.5.1 Preventing Lost Updates

To prevent lost updates when applying a transform, use this procedure:

1. Enable crossedition triggers.
2. Wait until pending changes to the affected tables are either committed or rolled back.

Use the procedure DBMS.Utility.WAIT_ON_PENDING_DML, described in *Oracle Database PL/SQL Packages and Types Reference*.

3. Apply the transform.

Note: This scenario, where the forward crossedition trigger changes only the table on which it is defined, is sufficient to illustrate the risk. Suppose that Session One issues an UPDATE statement against the table when the crossedition trigger is not yet enabled; and that Session Two then enables the crossedition trigger and immediately applies the transformation.

A race condition can now occur when both Session One and Session Two will change the same row (row n). Chance determines which session reaches row n first. Both updates succeed, even if the session that reaches row n second must wait until the session that reached it first commits its change and releases its lock.

The problem occurs when Session Two wins the race. Because its SQL statement was compiled after the trigger was enabled, the program that implements the statement also implements the trigger action; therefore, the intended post-upgrade column values are set for row n . Now Session One reaches row n , and because its SQL statement was compiled before the trigger was enabled, the program that implements the statement does not implement the trigger action. Therefore, the values that Session Two set in the post-upgrade columns do not change—they reflect the values that the source columns had before Session One updated row n . That is, the intended side-effect of Session One's update is lost.

24.3.6 Dropping the Crossedition Triggers

To drop a crossedition trigger, use the `DROP TRIGGER` statement, described in *Oracle Database PL/SQL Language Reference*. Alternatively, you can drop crossedition triggers by dropping the edition in which they are actual, by using the `DROP EDITION` statement with the `CASCADE` clause. For information about dropping editions, see [Section 24.1.8](#).

You drop crossedition triggers in these situations:

- You are rolling back the application upgrade (dropping the post-upgrade edition). Before dropping the post-upgrade edition, you must disable or drop any constraints on the new columns.
- You have finished the application upgrade and made the post-upgrade edition available to all users.

When all sessions are using the post-upgrade edition, you can drop the forward crossedition triggers. However, before dropping the reverse crossedition triggers, you must disable or drop any constraints on the old columns.

To disable or drop constraints, use the `ALTER TABLE` statement with the `DISABLE CONSTRAINT` or `DROP CONSTRAINT` clause. For information about the `ALTER TABLE` statement, see *Oracle Database SQL Language Reference*.

24.4 Displaying Information About EBR Features

Topics:

- [Displaying Information About Editions](#)
- [Displaying Information About Editioning Views](#)
- [Displaying Information About Crossedition Triggers](#)

24.4.1 Displaying Information About Editions

[Table 24-1](#) briefly describes the static data dictionary views that display information about editions. For more information about a specific view, see *Oracle Database Reference*.

Table 24-1 * Dictionary Views with Edition Information

View	Description
*_EDITIONS	Describes every edition in the database.
*_EDITION_COMMENTS	Shows the comments associated with every edition in the database.
*_EDITIONED_TYPES	Lists the schema object types that are editioned by default in each schema.
*_OBJECTS	Describes every object in the database that is visible in the current edition. For each object, this view shows whether it is editionable.
*_OBJECTS_AE	Describes every object in the database, in every edition. For each object, this view shows whether it is editionable.
*_ERRORS	Describes every error in the database in the current edition.
*_ERRORS_AE	Describes every error in the database, in every edition.
*_USERS	Describes every user in the database. Useful for showing which users have editions enabled.
*_SERVICES	Describes every service in the database. The EDITIONS column shows the default initial current edition.
*_MVIEWS	Describes every materialized view. If the materialized view refers to editioned objects, then this view shows the evaluation edition and the range of editions where the materialized view is eligible for query rewrite.
*_TAB_COLS	Describes every column of every table, view, and cluster. For each virtual column, this view shows the evaluation edition and the usable range.

Note: *_OBJECTS and *_OBJECTS_AE include dependent objects that are invalidated by operations in [Table 23-2](#) only after one of the following:

- A reference to the object (either during compilation or execution)
- An invocation of DBMS.Utility.COMPILE_SCHEMA (described in *Oracle Database PL/SQL Packages and Types Reference*)
- An invocation of any UTL_RECOMP subprogram (described in *Oracle Database PL/SQL Packages and Types Reference*)

24.4.2 Displaying Information About Editioning Views

[Table 24-2](#) briefly describes the static data dictionary views that display information about editioning views. For more information about a specific view, see *Oracle Database Reference*.

Table 24–2 * _ Dictionary Views with Editioning View Information

View	Description
*_VIEWS	Describes every view in the database that is visible in the current edition, including editioning views.
*_EDITIONING_VIEWS	Describes every editioning view in the database that is visible in the current edition. Useful for showing relationships between editioning views and their base tables. Join with *_OBJECTS_AE for additional information.
*_EDITIONING_VIEWS_AE	Describes every actual object in every editioning view in the database, in every edition.
*_EDITIONING_VIEW_COLS	Describes the columns of every editioning view in the database that is visible in the current edition. Useful for showing relationships between the columns of editioning views and the table columns to which they map. Join with *_OBJECTS_AE, *_TAB_COL, or both, for additional information.
*_EDITIONING_VIEW_COLS_AE	Describes the columns of every editioning view in the database, in every edition.

Each row of *_EDITIONING_VIEWS matches exactly one row of *_VIEWS, and each row of *_VIEWS that has EDITIONING_VIEW = 'Y' matches exactly one row of *_EDITIONING_VIEWS. Therefore, in this example, the WHERE clause is redundant:

```
SELECT ...
  FROM DBA_EDITIONING_VIEWS INNER JOIN DBA_VIEWS
    USING (OWNER, VIEW_NAME)
 WHERE EDITIONING_VIEW = 'Y'
   AND ...
```

The row of *_VIEWS that matches a row of *_EDITIONING_VIEWS has EDITIONING_VIEW = 'Y' by definition. Conversely, no row of *_VIEWS that has EDITIONING_VIEW = 'N' has a counterpart in *_EDITIONING_VIEWS.

24.4.3 Displaying Information About Crossedition Triggers

The static data dictionary views that display information about triggers are described in *Oracle Database Reference*. Crossedition triggers that appear in static data dictionary views are actual objects in the current edition.

Child cursors cannot be shared if the set of crossedition triggers that might run differs. The dynamic performance views V\$SQL_SHARED_CURSOR and GV\$SQL_SHARED_CURSOR have a CROSSEDITION_TRIGGER_MISMATCH column that tells whether this is true. For information about V\$SQL_SHARED_CURSOR, see *Oracle Database Reference*.

24.5 Using EBR to Upgrade an Application

To use EBR to upgrade your application online, you must first ready your application:

1. Editions-enable the appropriate users and the appropriate schema object types in their schemas.

In schemas where you will create editioning views (in the next step), the type VIEW must be editionable.

For instructions, see [Section 24.1.1.4](#).

2. Prepare your application to use editioning views.

For instructions, see [Section 24.5.1](#).

With the editioning views in place, you can use EBR to upgrade your application online as often as necessary. For each upgrade:

- If the type of every object that you will redefine is editionable (tables are not editionable), then use the procedure in [Section 24.5.2, "Procedure for EBR Using Only Editions."](#)
- If you will change the structure of one or more tables, and while you are doing so, other users *need not* be able to change data in those tables, then use the procedure in [Section 24.5.3, "Procedure for EBR Using Editioning Views."](#)
- If you will change the structure of one or more tables, and while you are doing so, other users must be able to change data in those tables, then use the procedure in [Section 24.5.4, "Procedure for EBR Using Crossedition Triggers."](#)

Topics:

- [Preparing Your Application to Use Editioning Views](#)
- [Procedure for EBR Using Only Editions](#)
- [Procedure for EBR Using Editioning Views](#)
- [Procedure for EBR Using Crossedition Triggers](#)
- [Rolling Back the Application Upgrade](#)
- [Reclaiming Space Occupied by Unused Table Columns](#)
- [Example: Using EBR to Upgrade an Application](#)

24.5.1 Preparing Your Application to Use Editioning Views

An application that uses one or more tables must cover each table with an editioning view. An editioning view **covers** a table when all of these statements are true:

- Every ordinary object in the application references the table only through the editioning view. (An **ordinary object** is any object except an editioning view or crossedition trigger. Editioning views and crossedition triggers must reference tables.)
- Application users are granted object privileges only on the editioning view, not on the table.
- Oracle Virtual Private Database (VPD) policies are attached only to the editioning view, not to the table. (Regular auditing and fine-grained auditing (FGA) policies are attached only to the table.)

When the editioning view is actualized, a copy of the VPD policy is attached to the actualized editioning view. (A policy is uniquely identified by its name and the object to which it is attached.) If the policy function is also actualized, the copy of the policy uses the actualized policy function; otherwise, it uses the original policy function.

The static data dictionary views `*_POLICIES`, which describe the VPD policies, can have different results in different editions.

See Also:

- *Oracle Database Security Guide* for information about VPD, including that static data dictionary views that show information about VPD policies
- *Oracle Database Reference* for information about *_POLICIES

If an existing application does not use editioning views, prepare it to use them by following this procedure for each table that it uses:

1. Give the table a new name (so that you can give its current name to its editioning view).

Oracle recommends choosing a new name that is related to the original name and reflects the change history. For example, if the original table name is `Data`, the new table name might be `Data_1`.

2. (Optional) Give each column of the table a new name.

Again, Oracle recommends choosing new names that are related to the original names and reflect the change history. For example, `Name` and `Number` might be changed to `Name_1` and `Number_1`.

Any triggers that depend on renamed columns are now invalid. For details, see the entry for `ALTER TABLE table RENAME column` in [Table 23-2](#).

3. Create the editioning view, giving it the original name of the table.

For instructions, see [Section 24.2.1](#).

Because the editioning view has the name that the table had, objects that reference that name now reference the editioning view.

4. If triggers are defined on the table, drop them, and rerun the code that created them.

Now the triggers that were defined on the table are defined on the editioning view.

5. If VPD policies are attached to the table, drop the policies and policy functions and rerun the code that created them.

Now the VPD policies that were attached to the table are attached to the editioning view.

6. Revoke all object privileges on the table from all application users.

To see which application users have which object privileges on the table, use this query:

```
SELECT GRANTEE, PRIVILEGE
  FROM DBA_TAB_PRIVS
 WHERE TABLE_NAME='table_name';
```

7. For every privilege revoked in step 6, grant the same privilege on the editioning view.

8. For each user who owns a private synonym that refers to the table, enable editions, specifying that the type `SYNONYM` is editionable in the schema (for instructions, see [Section 24.1.1.4](#)).

9. Notify the owners of private synonyms that refer to the table that they must re-create those synonyms.

24.5.2 Procedure for EBR Using Only Editions

Use this procedure only if every object that you will redefine is editioned (as defined in [Section 24.1.1](#)). Tables are never editioned objects.

1. Create a new edition.
For instructions, see [Section 24.1.2](#).
2. Make the new edition your session edition.
For instructions, see [Section 24.1.6.2](#).
3. Make the necessary changes to the editioned objects of the application.
4. Ensure that all objects are valid.

Query the static data dictionary `*_OBJECTS_AE`, which describes every actual object in the database, in every edition. If invalid objects remain, recompile them, using any `UTL_RECOMP` subprogram (described in *Oracle Database PL/SQL Packages and Types Reference*).

5. Check that the changes work as intended.
If so, go to step 6.
If not, either make further changes (return to step 3) or roll back the application upgrade (for instructions, see [Section 24.5.5](#)).
6. Make the new edition (the upgraded application) available to all users.
For instructions, see [Section 24.1.5](#).
7. Retire the old edition (the original application), so that all users except `SYS` use only the upgraded application.
For instructions, see [Section 24.1.7](#).

Example 24–6 shows how to use the preceding procedure to change a very simple PL/SQL procedure.

Example 24–6 EBR of Very Simple Procedure

1. Create PL/SQL procedure for this example:

```
CREATE OR REPLACE PROCEDURE hello IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, edition 1.');
END hello;
/
```

2. Invoke PL/SQL procedure:

```
BEGIN hello(); END;
/
```

Result:

```
Hello, edition 1.

PL/SQL procedure successfully completed.
```

3. Do EBR of procedure:

- a. Create new edition:

```
CREATE EDITION e2 AS CHILD OF ora$base;
```

Result:

Edition created.

- b.** Make new edition your session edition:

```
ALTER SESSION SET EDITION = e2;
```

Result:

Session altered.

- c.** Change procedure:

```
CREATE OR REPLACE PROCEDURE hello IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, edition 2.');
END hello;
/
```

Result:

Procedure created.

- d.** Check that change works as intended:

```
BEGIN hello(); END;
/
```

Result:

```
Hello, edition 2.
PL/SQL procedure successfully completed.
```

- e.** Make new edition available to all users (requires system privileges):

```
ALTER DATABASE DEFAULT EDITION = e2;
```

- f.** Retire old edition (requires system privileges):

List grantees:

```
SELECT GRANTEE, PRIVILEGE
FROM DBA_TAB_PRIVS
WHERE TABLE_NAME = UPPER('ora$base')
/
```

Result:

GRANTEE	PRIVILEGE
PUBLIC	USE

1 row selected.

Revoke use on old edition from all grantees:

```
REVOKE USE ON EDITION ora$base FROM PUBLIC;
```

24.5.3 Procedure for EBR Using Editioning Views

Use this procedure only if you will change the structure of one or more tables, and while you are doing so, other users *need not* be able to change data in those tables.

1. Create a new edition.
For instructions, see [Section 24.1.2](#).
2. Make the new edition your session edition.
For instructions, see [Section 24.1.6.2](#).
3. In the new edition, if the editioning views are read-only, make them read-write.
For instructions, see [Section 24.2.3](#).
4. In every edition except the new edition, make the editioning views read-only.
5. Make the necessary changes to the objects of the application.
6. Ensure that all objects are valid.
Query the static data dictionary *_OBJECTS_AE, which describes every actual object in the database, in every edition. If invalid objects remain, recompile them, using any UTL_RECOMP subprogram (described in *Oracle Database PL/SQL Packages and Types Reference*).
7. Check that the changes work as intended.
If so, go to step 8.
If not, either make further changes (return to step 5) or roll back the application upgrade (for instructions, see [Section 24.5.5](#)).
8. Make the upgraded application available to all users.
For instructions, see [Section 24.1.5](#).
9. Retire the old edition (the original application), so that all users except SYS use only the upgraded application.
For instructions, see [Section 24.1.7](#).

24.5.4 Procedure for EBR Using Crossedition Triggers

Use this procedure only if you will change the structure of one or more tables, and while you are doing so, other users must be able to change data in those tables.

1. Create a new edition.
For instructions, see [Section 24.1.2](#).
2. Make the new edition your session edition.
For instructions, see [Section 24.1.6.2](#).
3. Make the permanent changes to the objects of the application.
For example, add new columns to the tables and create any new permanent subprograms.
Objects that depend on objects that you changed might now be invalid. For more information, see [Table 23-2](#).
4. Ensure that all objects are valid.
Query the static data dictionary *_OBJECTS_AE, which describes every actual object in the database, in every edition. If invalid objects remain, recompile them, using any UTL_RECOMP subprogram (described in *Oracle Database PL/SQL Packages and Types Reference*).

5. Create the temporary objects—the crossedition triggers (in the disabled state) and any subprograms that they need.

For instructions, see [Section 24.3.4](#).

You need reverse crossedition triggers only if you do step 10, which is optional.

6. When the crossedition triggers compile successfully, enable them.

Use the ALTER TRIGGER statement with the ENABLE option. For information about this statement, see *Oracle Database PL/SQL Language Reference*.

7. Wait until pending changes are either committed or rolled back.

Use the procedure DBMS_UTILITY.WAIT_ON_PENDING_DML, described in *Oracle Database PL/SQL Packages and Types Reference*.

8. Apply the transforms.

For instructions, see [Section 24.3.5](#).

Note: It is impossible to predict whether this step visits an existing row before a user of an ancestor edition updates, inserts, or deletes data from that row.

9. Check that the changes work as intended.

If so, go to step 10.

If not, either make further changes (return to step 3) or roll back the application upgrade (for instructions, see [Section 24.5.5](#)).

10. (Optional) Grant the USE privilege on your session edition to the early users of the upgraded application.

For instructions, see [Section 24.1.4](#).

11. Make the upgraded application available to all users.

For instructions, see [Section 24.1.5](#).

12. Disable or drop the constraints and then drop the crossedition triggers.

For instructions, see [Section 24.3.6](#).

13. Retire the old edition (the original application), so that all users except SYS use only the upgraded application.

For instructions, see [Section 24.1.7](#).

24.5.5 Rolling Back the Application Upgrade

To roll back the application upgrade:

1. Change your session edition to something other than the new edition that you created for the upgrade.

For instructions, see [Section 24.1.6.2](#).

2. Drop the new edition that you created for the upgrade.

For instructions, see [Section 24.1.8](#).

3. If you created new table columns during the upgrade, reclaim the space that they occupy (for instructions, see [Section 24.5.6](#)).

24.5.6 Reclaiming Space Occupied by Unused Table Columns

If you roll back an upgrade for which you created new table columns,

To reclaim the space that unused columns occupy:

1. Set the values of the unused columns to `NULL`.

To avoid locking out other users while doing this operation, use the `DBMS_PARALLEL_EXECUTE` procedure (described in *Oracle Database PL/SQL Packages and Types Reference*).

2. Set the unused columns to `UNUSED`.

Use the `ALTER TABLE` statement (described in *Oracle Database SQL Language Reference*) with the `SET UNUSED` clause (described in *Oracle Database SQL Language Reference*).

3. Shrink the table.

Use the `ALTER TABLE` statement (described in *Oracle Database SQL Language Reference*) with the `SHRINK SPACE` clause (described in *Oracle Database SQL Language Reference*).

24.5.7 Example: Using EBR to Upgrade an Application

This example uses an edition, an editioning view, a forward crossedition trigger, and a reverse crossedition trigger.

Topics:

- [Existing Application](#)
- [Preparing the Application to Use Editioning Views](#)
- [Using EBR to Upgrade the Application](#)

Note: Before you can use EBR to upgrade an application, you must enable editions for every schema that the application uses. For instructions, see [Section 24.1.1.4](#).

24.5.7.1 Existing Application

The existing application—the application to be upgraded—consists of a single table on which a trigger is defined. The application was created as in [Example 24-7](#).

Example 24-7 Creating the Existing Application

1. Create table:

```
CREATE TABLE Contacts (
    ID          NUMBER(6,0) CONSTRAINT Contacts_PK PRIMARY KEY,
    Name        VARCHAR2(47),
    Phone_Number VARCHAR2(20)
);
```

2. Populate table (not shown).

3. Prepare to create trigger on table:

```
ALTER TABLE Contacts ENABLE VALIDATE CONSTRAINT Contacts_PK;
```

```
DECLARE Max_ID INTEGER;
```

```

BEGIN
  SELECT MAX(ID) INTO Max_ID FROM Contacts;
  EXECUTE IMMEDIATE '
    CREATE SEQUENCE Contacts_Seq
      START WITH '||To_Char(Max_ID + 1)';
END;
/

```

4. Create trigger:

```

CREATE TRIGGER Contacts_BI
  BEFORE INSERT ON Contacts FOR EACH ROW
BEGIN
  :NEW.ID := Contacts_Seq.NEXTVAL;
END;
/

```

[Example 24–8](#) shows how the table Contacts looks after being populated with data.

Example 24–8 Viewing Data in Existing Table

Query:

```

SELECT * FROM Contacts
ORDER BY Name;

```

Result:

ID	NAME	PHONE_NUMBER
174	Abel, Ellen	011.44.1644.429267
166	Ande, Sundar	011.44.1346.629268
130	Atkinson, Mozhe	650.124.6234
105	Austin, David	590.423.4569
204	Baer, Hermann	515.123.8888
116	Baida, Shelli	515.127.4563
167	Banda, Amit	011.44.1346.729268
172	Bates, Elizabeth	011.44.1343.529268
192	Bell, Sarah	650.501.1876
151	Bernstein, David	011.44.1344.345268
129	Bissot, Laura	650.124.5234
169	Bloom, Harrison	011.44.1343.829268
185	Bull, Alexis	650.509.2876
187	Cabrio, Anthony	650.509.4876
148	Cambrault, Gerald	011.44.1344.619268
154	Cambrault, Nanette	011.44.1344.987668
110	Chen, John	515.124.4269
...		
120	Weiss, Matthew	650.123.1234
200	Whalen, Jennifer	515.123.4444
149	Zlotkey, Eleni	011.44.1344.429018

107 rows selected.

Suppose that you must redefine Contacts, replacing the Name column with the columns First_Name and Last_Name, and adding the column Country_Code. Also suppose that while you are making this structural change, other users must be able to change the data in Contacts.

You need all features of EBR: the edition, which is always needed; the editioning view, because you are redefining a table; and crossedition triggers, because other users must be able to change data in the table while you are redefining it.

24.5.7.2 Preparing the Application to Use Editioning Views

[Example 24-9](#) shows how to create the editioning view from which other users will access the table `Contacts` while you are redefining it in the new edition.

Example 24-9 Creating an Editioning View for the Existing Table

1. Give table a new name (so that you can give its current name to editioning view):

```
ALTER TABLE Contacts RENAME TO Contacts_Table;
```

2. (Optional) Give columns of table new names:

```
ALTER TABLE Contacts_Table
  RENAME COLUMN Name TO Name_1;
```

```
ALTER TABLE Contacts_Table
  RENAME COLUMN Phone_Number TO Phone_Number_1;
```

3. Create editioning view:

```
CREATE OR REPLACE EDITIONING VIEW Contacts AS
  SELECT
    ID,
    Name_1          Name,
    Phone_Number_1  Phone_Number
  FROM Contacts_Table;
```

4. Move trigger `Contacts_BI` from table to editioning view:

```
DROP TRIGGER Contacts_BI;
```

```
CREATE TRIGGER Contacts_BI
  BEFORE INSERT ON Contacts FOR EACH ROW
BEGIN
  :NEW.ID := Contacts_Seq.NEXTVAL;
END;
/
```

24.5.7.3 Using EBR to Upgrade the Application

[Example 24-10](#) shows how to create an edition in which to upgrade the existing application (in [Section 24.5.7.1](#)), make the new edition the session edition, and check that the new edition really is the session edition.

Example 24-10 Creating Edition in Which to Upgrade the Application

1. Create new edition:

```
CREATE EDITION Post_Upgrade AS CHILD OF Ora$Base;
```

2. Make new edition your session edition:

```
ALTER SESSION SET EDITION = Post_Upgrade;
```

3. Check session edition:

```
SELECT
  SYS_CONTEXT('Userenv', 'Current_Edition_Name') "Current_Edition"
```

```
FROM DUAL;
```

Result:

```
Current_Edition
```

```
-----
```

```
POST_UPGRADE
```

```
1 row selected.
```

In the Post_Upgrade edition, [Example 24-11](#) shows how to add the new columns to the physical table and recompile the trigger that was invalidated by adding the columns. Then, it shows how to replace the editioning view Contacts so that it selects the columns of the table by their desired logical names.

Note: Because you will change the base table, see "[Nonblocking and Blocking DDL Statements](#)" on page 6-31.

Example 24-11 *Changing the Table and Replacing the Editioning View*

1. Add new columns to physical table:

```
ALTER TABLE Contacts_Table ADD (
    First_Name_2      varchar2(20),
    Last_Name_2       varchar2(25),
    Country_Code_2    varchar2(20),
    Phone_Number_2    varchar2(20)
);
```

(This is nonblocking DDL.)

2. Recompile invalidated trigger:

```
ALTER TRIGGER Contacts_BI COMPILE REUSE SETTINGS;
```

3. Replace editioning view so that it selects replacement columns with their desired logical names:

```
CREATE OR REPLACE EDITIONING VIEW Contacts AS
  SELECT
    ID          ID,
    First_Name_2 First_Name,
    Last_Name_2 Last_Name,
    Country_Code_2 Country_Code,
    Phone_Number_2 Phone_Number
  FROM Contacts_Table;
```

In the Post_Upgrade edition, [Example 24-12](#) shows how to create two procedures for the forward crossedition trigger to use, create both the forward and reverse crossedition triggers in the disabled state, and enable them.

Example 24-12 *Creating and Enabling the Crossedition Triggers*

1. Create first procedure that forward crossedition trigger uses:

```
CREATE OR REPLACE PROCEDURE Set_First_And_Last_Name (
  Name      IN  VARCHAR2,
  First_Name OUT VARCHAR2,
  Last_Name  OUT VARCHAR2)
IS
  Comma_Pos NUMBER := INSTR(Name, ',');
```

```

BEGIN
  IF Comma_Pos IS NULL OR Comma_Pos < 2 THEN
    RAISE Program_Error;
  END IF;

  Last_Name := SUBSTR(Name, 1, Comma_Pos-1);
  Last_Name := RTRIM(Ltrim(Last_Name));

  First_Name := SUBSTR(Name, Comma_Pos+1);
  First_Name := RTRIM(LTRIM(First_Name));
END Set_First_And_Last_Name;
/

```

2. Create second procedure that forward crossedition trigger uses:

```

CREATE OR REPLACE PROCEDURE Set_Country_Code_And_Phone_No (
  Phone_Number      IN VARCHAR2,
  Country_Code      OUT VARCHAR2,
  Phone_Number_V2   OUT VARCHAR2)
IS
  Char_To_Number_Error EXCEPTION;
  PRAGMA EXCEPTION_INIT(Char_To_Number_Error, -06502);
  Bad_Phone_Number EXCEPTION;
  Nmbr VARCHAR2(30) := REPLACE(Phone_Number, '.', '-');

  FUNCTION Is_US_Number(Nmbr IN VARCHAR2)
    RETURN BOOLEAN
  IS
    Len NUMBER := LENGTH(Nmbr);
    Dash_Pos NUMBER := INSTR(Nmbr, '-');
    n PLS_INTEGER;
  BEGIN
    IF Len IS NULL OR Len <> 12 THEN
      RETURN FALSE;
    END IF;
    IF Dash_Pos IS NULL OR Dash_Pos <> 4 THEN
      RETURN FALSE;
    END IF;
    BEGIN
      n := TO_NUMBER(SUBSTR(Nmbr, 1, 3));
      EXCEPTION WHEN Char_To_Number_Error THEN
        RETURN FALSE;
    END;

    Dash_Pos := INSTR(Nmbr, ' ', 5);

    IF Dash_Pos IS NULL OR Dash_Pos <> 8 THEN
      RETURN FALSE;
    END IF;

    BEGIN
      n := TO_NUMBER(SUBSTR(Nmbr, 5, 3));
      EXCEPTION WHEN Char_To_Number_Error THEN
        RETURN FALSE;
    END;

    BEGIN
      n := TO_NUMBER(SUBSTR(Nmbr, 9));
      EXCEPTION WHEN Char_To_Number_Error THEN
        RETURN FALSE;
    END;
  END;

```

```

        RETURN TRUE;
    END Is_US_Number;

    BEGIN
        IF Nmbr LIKE '011-%' THEN
            DECLARE
                Dash_Pos NUMBER := INSTR(Nmbr, '-', 5);
            BEGIN
                Country_Code := '+' || TO_NUMBER(SUBSTR(Nmbr, 5, Dash_Pos-5));
                Phone_Number_V2 := SUBSTR(Nmbr, Dash_Pos+1);
            EXCEPTION WHEN Char_To_Number_Error THEN
                raise Bad_Phone_Number;
            END;
        ELSIF Is_US_Number(Nmbr) THEN
            Country_Code := '+1';
            Phone_Number_V2 := Nmbr;
        ELSE
            RAISE Bad_Phone_Number;
        END IF;
    EXCEPTION WHEN Bad_Phone_Number THEN
        Country_Code := '+0';
        Phone_Number_V2 := '000-000-0000';
    END Set_Country_Code_And_Phone_No;
/

```

3. Create forward crossedition trigger in disabled state:

```

CREATE OR REPLACE TRIGGER Contacts_Fwd_Xed
    BEFORE INSERT OR UPDATE ON Contacts_Table
    FOR EACH ROW
    FORWARD CROSSEDITION
    DISABLE
    BEGIN
        Set_First_And_Last_Name(
            :NEW.Name_1,
            :NEW.First_Name_2,
            :NEW.Last_Name_2
        );
        Set_Country_Code_And_Phone_No(
            :NEW.Phone_Number_1,
            :NEW.Country_Code_2,
            :NEW.Phone_Number_2
        );
    END Contacts_Fwd_Xed;
/

```

4. Enable forward crossedition trigger:

```
ALTER TRIGGER Contacts_Fwd_Xed ENABLE;
```

5. Create reverse crossedition trigger in disabled state:

```

CREATE OR REPLACE TRIGGER Contacts_Rvrs_Xed
    BEFORE INSERT OR UPDATE ON Contacts_Table
    FOR EACH ROW
    REVERSE CROSSEDITION
    DISABLE
    BEGIN
        :NEW.Name_1 := :NEW.Last_Name_2 || ', ' || :NEW.First_Name_2;
        :NEW.Phone_Number_1 :=

```

```

CASE :New.Country_Code_2
WHEN '+1' THEN
    REPLACE(:NEW.Phone_Number_2, '-', '.')
ELSE
    '011.' || LTRIM(:NEW.Country_Code_2, '+') || '.' ||
    REPLACE(:NEW.Phone_Number_2, '-', '.')
END;
END Contacts_Rvrs_Xed;
/

```

6. Enable reverse crossedition trigger:

```
ALTER TRIGGER Contacts_Rvrs_Xed ENABLE;
```

7. Wait until pending changes are either committed or rolled back:

```

DECLARE
    scn          NUMBER  := NULL;
    timeout CONSTANT INTEGER := NULL;
BEGIN
    IF NOT DBMS_UTILITY.WAIT_ON_PENDING_DML(Tables  => 'Contacts_Table',
                                              timeout => timeout,
                                              scn      => scn)
THEN
    RAISE_APPLICATION_ERROR(-20000,
                           'Wait_On_Pending_DML() timed out. CETs were enabled before SCN: ' || SCN);
    END IF;
END;
/

```

For information about the DBMS_UTILITY.WAIT_ON_PENDING_DML procedure, see *Oracle Database PL/SQL Packages and Types Reference*.

In the Post_Upgrade edition, [Example 24-13](#) shows how to apply the transforms.

Example 24-13 Applying the Transforms

```

DECLARE
    c NUMBER := DBMS_SQL.OPEN_CURSOR();
    x NUMBER;
BEGIN
    DBMS_SQLPARSE(
        c           => c,
        Language_Flag => DBMS_SQL.NATIVE,
        Statement      => 'UPDATE Contacts_Table SET ID = ID',
        Apply_Crossedition_Trigger => 'Contacts_Fwd_Xed'
    );
    x := DBMS_SQL.EXECUTE(c);
    DBMS_SQL CLOSE_CURSOR(c);
    COMMIT;
END;
/

```

In the Post_Upgrade edition, [Example 24-14](#) shows how to check that the change worked as intended. Compare [Example 24-14](#) to [Example 24-8](#).

Example 24-14 Viewing Data in Changed Table

1. Format columns for readability:

```
COLUMN ID FORMAT 999
COLUMN Last_Name FORMAT A15
```

```
COLUMN First_Name FORMAT A15
COLUMN Country_Code FORMAT A12
COLUMN Phone_Number FORMAT A12
```

2. Query:

```
SELECT * FROM Contacts
ORDER BY Last_Name;
```

Result:

ID	FIRST_NAME	LAST_NAME	COUNTRY_CODE	PHONE_NUMBER
174	Ellen	Abel	+44	1644-429267
166	Sundar	Ande	+44	1346-629268
130	Mozhe	Atkinson	+1	650-124-6234
105	David	Austin	+1	590-423-4569
204	Hermann	Baer	+1	515-123-8888
116	Shelli	Baida	+1	515-127-4563
167	Amit	Banda	+44	1346-729268
172	Elizabeth	Bates	+44	1343-529268
192	Sarah	Bell	+1	650-501-1876
151	David	Bernstein	+44	1344-345268
129	Laura	Bissot	+1	650-124-5234
169	Harrison	Bloom	+44	1343-829268
185	Alexis	Bull	+1	650-509-2876
187	Anthony	Cabrio	+1	650-509-4876
154	Nanette	Cambrault	+44	1344-987668
148	Gerald	Cambrault	+44	1344-619268
110	John	Chen	+1	515-124-4269
...				
120	Matthew	Weiss	+1	650-123-1234
200	Jennifer	Whalen	+1	515-123-4444
149	Eleni	Zlotkey	+44	1344-429018

107 rows selected.

If the change worked as intended, you can now follow steps 10 through 13 of the procedure in [Section 24.5.4](#).

25

Using Transaction Guard

Transaction Guard provides a generic tool for applications to use for at-most-once execution in case of planned and unplanned outages. Applications use the logical transaction ID to determine the outcome of the last transaction open in a database session following an outage. Without Transaction Guard, applications that attempt to replay operations following outages can cause logical corruption by committing duplicate transactions.

Transaction Guard provides these benefits:

- Preserves the commit outcome
- Ensures a known outcome for every transaction
- Can be used to provide at-most-once transaction execution

This chapter assumes that you are familiar with the major relevant concepts and techniques of the technology or product environment in which you are using Transaction Guard.

Topics:

- [Problem that Transaction Guard Solves](#)
- [Solution that Transaction Guard Provides](#)
- [Transaction Guard Concepts and Scope](#)
- [Database Configuration for Transaction Guard](#)
- [Developing Applications that Use Transaction Guard](#)
- [Transaction Guard and Its Relationship to Application Continuity](#)

See Also:

- *Oracle Database JDBC Developer's Guide* for more information about using Transaction Guard with Oracle Java Database Connectivity (JDBC)
- *Oracle Call Interface Programmer's Guide* for more information about using Transaction Guard with OCI

25.1 Problem that Transaction Guard Solves

In applications without Transaction Guard, a fundamental problem for recovering applications after an outage is that the commit message that is sent back to the client is not durable. If there is a break between the client and the server, the client sees an error message indicating that the communication failed. This error does not inform the

application if the submission executed any commit operations, if a procedural call completed and executed all expected commits and session state changes, or if a call failed part way through or, yet worse, is still running disconnected from the client.

Without Transaction Guard, it is impossible or extremely difficult to determine the outcome of the last commit operation, in a guaranteed and scalable manner, after a communication failure to the server. If an application must determine whether the submission to the database was committed, the application must add custom exception code to query the outcome for every possible commit point in the application. Given that a system can fail anywhere, this is almost impractical because the query must be specific to each submission. After an application is built and is in production, this is completely impractical. Moreover, a query cannot give the correct answer because the transaction could commit immediately after that query executed. Indeed, after a communication failure the server may still be running the submission not yet aware that the client has disconnected. For PL/SQL or Java in the database, for a procedural submission, there is also no record as to whether that submission ran to completion or was aborted part way through. While such a procedure may have committed, subsequent work may not have been done for the procedure.

Failing to recognize that the last submission has committed, or will commit sometime soon or has not run to completion, can lead applications that attempt to replay, thus causing duplicate transaction submissions and other forms of "logical corruption" because the software might try to reissue already persisted changes.

Without Transaction Guard, if a transaction has been started and commit has been issued, the commit message that is sent back to the client is not durable. The client is left not knowing whether the transaction committed. The transaction cannot be validly resubmitted if the nontransactional state is incorrect or if it already committed. In the absence of guaranteed commit and completion information, resubmission can lead to transactions applied more than once and in a session with the incorrect state.

25.2 Solution that Transaction Guard Provides

Effective with Oracle Database 12c Release 1 (12.1.0.1), Transaction Guard provides new, integrated tools for applications to use to achieve idempotence automatically and transparently, and in a manner that scales. Its key features are the following:

- Durability of COMMIT outcome by saving a logical transaction identifier (LTXID) at commit for all supported transaction types against the database (Oracle Database 12c Release 1 (12.1.0.1) or later). This includes idempotence for transactions executed using autocommit, from inside PL/SQL, from remote transactions, and from callouts that cannot otherwise be identified using generic means.
- Use of the LTXID to support at-most-once execution semantics, such that database transactions protected by logical transaction identifiers cannot be duplicated when there are multiple copies of that transaction in flight identified by the LTXID.
- Blocking of a commit of in-flight work to ensure that regardless of the outage situation, another submission of the same transaction protected by that LTXID cannot commit.
- Identification of whether work committed at an LTXID was committed as part of a top-level call (client to server), or was embedded in a procedure (such as PL/SQL) at the server. An embedded commit state indicates that while a commit completed, the entire procedure in which the commit executed has not yet run to completion. Any work beyond the commit cannot be guaranteed to have completed until that procedure itself returns to the database engine.

- Identification of whether the database to which the commit resolution is directed is ahead of, in sync with, or behind the original submission, and rejection when there are gaps in the submission sequence of transactions from a client. It is considered an error to attempt to obtain an outcome if the server or client are not in sync on an LTXID sequence.
- A callback on the JDBC Thin client driver that fires when the LTXID changes. This can be used by higher layer applications such as WebLogic Server and third parties to maintain the current LTXID ready to use if needed.
- Namespace uniqueness across globally disparate databases and across databases that are consolidated into a container (pluggable) infrastructure. This includes Oracle Real Application Clusters (Oracle RAC), Data Guard, and pluggable databases.
- Service name uniqueness across global databases and across databases that are consolidated into a container (pluggable databases) infrastructure. This ensures that connections are properly directed to the transaction information.

25.3 Transaction Guard Concepts and Scope

This section explains some key concepts for Transaction Guard, and what Transaction Guard covers and does not cover.

Topics:

- [Logical Transaction Identifier \(LTXID\)](#)
- [At-Most-Once Execution](#)
- [Transaction Guard Coverage](#)
- [Transaction Guard Exclusions](#)

See Also:

- *Oracle Database Concepts* for more information about how Transaction Guard works
- *Oracle Database JDBC Developer's Guide* for more information about using Transaction Guard with Oracle Java Database Connectivity (JDBC)

25.3.1 Logical Transaction Identifier (LTXID)

Applications use a concept called the **logical transaction identifier (LTXID)** to determine the outcome of the last transaction open in a database session following an outage. The logical transaction ID is stored in the OCI session handle and in a connection object for the JDBC Thin and ODP.Net drivers. The logical transaction ID is the foundation of the at-most-once semantics.

The Transaction Guard protocol ensures that:

- Execution of each logical transaction is unique.
- Duplication is detected at supported commit time to ensure that for all commit points, the protocol must not be circumvented.
- When the transaction is committed, the logical transaction ID is persisted for the duration of the retention period for retries (default = 24 hours, maximum = 30 days).

- When obtaining the outcome, an LTXID is blocked to ensure that an earlier in-flight version of that LTXID cannot commit, by enforcing the uncommitted status. If the earlier version with the same LTXID was already committed or forced, then blocking the LTXID returns the same result.

The logical session number is automatically assigned at session establishment. It is an opaque structure that cannot be read by an application. For scalability, each LTXID carries a running number called the commit number, which is increased when a database transaction is committed for each round trip to the database. This running commit number is zero-based.

25.3.2 At-Most-Once Execution

Transaction Guard uses the logical transaction identifier (LTXID) to avoid duplicate transactions. This ability to ensure at most one execution of a transaction is referred to as *transaction idempotence*. The LTXID is persisted on commit and is reused following a rollback. During normal runtime, an LTXID is automatically held in the session at both the client and server for each database transaction. At commit, the LTXID is persisted as part of committing the transaction.

The at-most-once protocol requires that the database maintain the LTXID for the retention period agreed for replay. The default retention period is 24 hours, although you might need a shorter or longer period, conceivably even a week or longer. The longer the retention period, the longer the at-most-once check blocks an old transaction using an old LTXID from replay. The setting is available on each service. When multiple databases are involved, as is the case when using Data Guard and Active Data Guard, the LTXID is replicated to each database involved through the use of redo.

The `getLTXID` API, provided for Oracle JDBC Thin (with similar APIs for OCI, OCCI, and ODP.Net clients), lets an application retrieve the logical transaction identifier that was in use on the dead session. This is needed to determine the status of this last transaction.

The `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL subprogram (described in *Oracle Database PL/SQL Packages and Types Reference*) lets an application find the outcome of an action for a specified logical transaction identifier. Calling `DBMS_APP_CONT.GET_LTXID_OUTCOME` may involve the server blocking the LTXID from committing so that the outcome is known. This is a requirement if a transaction using that LTXID is in flight or is about to commit. An application using Transaction Guard obtains the LTXID following a recoverable error, and then calls `DBMS_APP_CONT.GET_LTXID_OUTCOME` before attempting a replay.

25.3.3 Transaction Guard Coverage

Transaction Guard supports commits from these sources:

- Local transactions
- Data definition language (DDL) transactions
- Data control language (DCL) transactions
- Distributed transactions
- Remote transactions
- Parallel transactions
- Commit on success (auto-commit)

- PL/SQL with embedded commit-supported client drivers
- Supported client drivers:
 - 12c JDBC type 4 driver
 - 12c OCI and OCCI client drivers
 - 12c Oracle Data Provider for .NET (ODP.NET) client driver

25.3.4 Transaction Guard Exclusions

Transaction Guard intentionally excludes recursive transactions and autonomous transactions so that they can be replayed.

As of Oracle Database 12c Release 1 (12.1.0.1), Transaction Guard also excludes:

- XA transactions
- Active Data Guard with read/write database links for forwarding transactions
- Golden Gate and Logical Standby

25.4 Database Configuration for Transaction Guard

This section contains information relevant to configuring the database for using Transaction Guard.

Topics:

- [Configuration Checklist](#)
- [Transaction History Table](#)
- [Service Parameters](#)

25.4.1 Configuration Checklist

To use Transaction Guard with an application, you must do the following:

- Use Oracle Database 12c Release 1 (12.1.0.1) or later.
 - Use an application service for all database work. Create the service using the `srvctl` command if you are using Oracle RAC, or using the `DBMS_SERVICE.CREATE_SERVICE` PL/SQL subprogram if you are not using Oracle RAC.
- Do **not** use the default database services, because these services are for administration purposes and cannot be manipulated. That is, do not use a service name that is set to `db_name` or `db_unique_name`.
- Grant permission on the `DBMS_APP_CONT` package to the database users who will call `GET_LTXID_OUTCOME`:
- ```
GRANT EXECUTE ON DBMS_APP_CONT TO <user-name>;
```

To use Transaction Guard with an application, Oracle recommends that you do the following:

- Locate and define the transaction history table for optimal performance. (The transaction history table is described in [Section 25.4.2](#).)
- If you are using Oracle RAC or Oracle Data Guard, ensure that FAN is configured to communicate to interrupt clients fast on error.
- Set the following parameter: `AQ_HA_NOTIFICATIONS = TRUE` (if using OCI FAN).

## 25.4.2 Transaction History Table

The transaction history table maintains the mapping of logical transaction identifiers (LTXIDs) to database transaction. This table can be accessed only by databases users with DBA privileges. It is maintained automatically by Oracle Database, and users must not issue DDL or DML statements directly against the transaction history table.

The transaction history table (LTXID\_TRANS) is created by default in the SYSAUX tablespace at database creation and upgrade. New partitions are added when instances are added, using the storage of the last partition. However, if the location of this tablespace is not optimal for performance, the DBA can move partitions to another tablespace. For example, the following statement alters the transaction history table to move it to a tablespace named FastPace:

```
ALTER TABLE LTXID_TRANS move partition LTXID_TRANS_4
 tablespace FastPace
 storage (initial 10G next 10G
 minextents 1 maxextents 121);
```

### See Also:

- *Oracle Database SQL Language Reference* for information about the ALTER TABLE statement

## 25.4.3 Service Parameters

Configure the services for commit outcome and retention. For example:

```
COMMIT_OUTCOME = TRUE
RETENTION_TIMEOUT = <retention-value>
```

COMMIT\_OUTCOME determines whether transaction commit outcome is accessible after the commit has executed. This feature makes the outcome of the commit durable, and it is used by applications to enforce the status of the last transaction executed before an outage. The feature is used internally by the Oracle replay driver and by WebLogic Server, and it is available for use by other applications to determine an outcome. The COMMIT\_OUTCOME possible values are FALSE (the default) and TRUE, and the value must be TRUE for Transaction Guard to be in effect.

The following considerations apply to COMMIT\_OUTCOME:

- Using the DBMS\_APP\_CONT.GET\_LTXID\_OUTCOME PL/SQL procedure requires that COMMIT\_OUTCOME be TRUE.
- COMMIT\_OUTCOME has no effect on Active Data Guard and read-only databases. Using Transaction Guard with read/write Active Data Guard combined with DML forwarding is not supported.
- COMMIT\_OUTCOME is allowed on user-defined database services. Use on the database service is excluded because this service does not switch across Data Guard and cannot be started, stopped, or disabled for planned outages at the primary database.

RETENTION\_TIMEOUT is used in conjunction with COMMIT\_OUTCOME to set the amount of time that the commit outcome is retained. The retention timeout value is specified in seconds; the default is 86400 (24 hours), and the maximum is 2592000 (30 days). You can use the `srvctl` command or the `DBMS_SERVICE` PL/SQL package to specify the retention timeout value.

If you are using Oracle RAC or Oracle RAC One, use the `srvctl` command to create and modify services. [Example 25-1](#) and [Example 25-2](#) show the use of `srvctl`. You can also use Global Data Services (GDSCTL).

**Example 25–1 Adding and Modifying a Service for a Server Pool**

```
srvctl add service -database orcl -service GOLD -poolname ora.Srvpool -commit_
outcome TRUE -retention 604800
srvctl modify service -database orcl -service GOLD -commit_outcome TRUE
-retention 604800
```

**Example 25–2 Adding an Administrator-Managed Service**

```
srvctl add service -database codedb -service GOLD -prefer serv1 -available serv2
-commit_outcome TRUE -retention 604800
```

If you are using a single-instance database, use the DBMS\_SERVICE.MODIFY\_SERVICE PL/SQL procedure to modify services and use FAN. [Example 25–3](#) modifies a service (but substitute the actual service name for <service-name>).

**Example 25–3 Modifying a Service (PL/SQL)**

```
DECLARE
 params dbms_service.svc_parameter_array;
BEGIN
 params('COMMIT_OUTCOME') := 'true';
 params('RETENTION_TIMEOUT') := 604800;
 dbms_service.modify_service('<service-name>', params);
END;
/
```

**See Also:**

- *Oracle Database Administrator's Guide* for information about the `srvctl add service` and `srvctl modify service` commands
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SERVICE` package.
- *Oracle Database 2 Day + Real Application Clusters Guide* for information about configuring OCI clients for high availability

## 25.5 Developing Applications that Use Transaction Guard

To use Transaction Guard, review the requirements and recommendations in [Section 25.4.1, "Configuration Checklist"](#), and follow these steps in the error handling when a recoverable error occurs:

1. Check that the error is a recoverable error that has made the database session unavailable.
2. Acquire the LTXID from the previous failed session using the client driver provided APIs (getLTXID for JDBC, OCI\_ATTR\_GET with LTXID for OCI, and LogicalTransactionId for ODP.net).
3. Acquire a new session with that sessions' own LTXID.
4. Invoke the `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL procedure with the LTXID obtained from the API. The return state tells the driver if the last transaction was COMMITTED (TRUE/FALSE) and USER\_CALL\_COMPLETED (TRUE/FALSE). This PL/SQL function returns an error if the client and database are out of sync (for example, not the same database or restored database).
5. The application can return the result to the user to decide. An application can replay itself. If the replay itself incurs an outage, then the LTXID for the replaying session is used for the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure.

**Topics:**

- [Typical Transaction Guard Usage](#)
- [Details for Using the LTXID](#)
- [Connection-Pool LTXID Usage](#)

### 25.5.1 Typical Transaction Guard Usage

The following pseudocode shows a typical usage of Transaction Guard:

```
Receive a FAN down event (or recoverable error)

FAN aborts the dead session

If recoverable error (new OCI_ATTRIBUTE for OCI, isRecoverable for JDBC)
 Get last LTXID from dead session using getLTXID or from your callback
 Obtain a new session
 Call GET_LTXID_OUTCOME with last LTXID to obtain COMMITTED and USER_CALL_
 COMPLETED status

If COMMITTED and USER_CALL_COMPLETED
 Then return result

ELSEIF COMMITTED and NOT USER_CALL_COMPLETED
 Then return result with a warning (that details such as out binds or row
 count were not returned)

ELSEIF NOT COMMITTED
 Cleanup and resubmit request, or return uncommitted result to the client
```

### 25.5.2 Details for Using the LTXID

For replay and returning results, the application or third party container needs access to the next LTXID to be committed at the server for each session. The LTXID can be obtained using APIs (getLTXID for JDBC and OCI\_ATTR\_GET with LTXID for OCI) from a failed session after a recoverable outage.

The JDBC Thin driver also provides a callback that executes on each commit number change received from the database. A third party container can use this callback to save the current LTXID in preparation to use if failover is needed. Within each session, the current LTXID is in use, so the callback can override earlier ones.

If failovers cascade without completing (that is, if during recovery from one failure, another failure occurs), the application *must* obtain and then pass the LTXID in effect on the current session into GET\_LTXID\_OUTCOME.

Table 25-1 shows several conditions or situations that require some LTXID-related action, and for each the application action and next LTXID to use.

**Table 25–1 LTXID Condition or Situation, Application Actions, and Next LTXID to Use**

| Condition or Situation                                                                                                          | Application Action                                                                                                                                                                                                                                 | Next LTXID to Use (Callback on LTXID Change for Containers - JDBC Thin Only)      |
|---------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| Application receives a recoverable error and calls <code>GET_LTXID_OUTCOME</code> to determine the transaction status.          | Application takes a new connection (with its own LTXID-B 0) and calls <code>GET_LTXID_OUTCOME</code> with the LTXID of the last failed session (LTXID-A).                                                                                          | New LTXID-B 0<br>Also set using the JDBC callback when registered                 |
| Application finds that the last session transaction status is <code>COMMITTED</code> and <code>USER_CALL_COMPLETED</code> .     | Returns committed status to client; the application may be able to continue.                                                                                                                                                                       | (Not applicable)                                                                  |
| Application finds that the last session transaction status is <code>COMMITTED</code> and <code>NOT USER_CALL_COMPLETED</code> . | Returns committed status to client and exits - some applications cannot progress as the work in the call is not complete. (for example, an out bind or row count was not returned). Whether the application can continue is application dependent. | (Not applicable)                                                                  |
| Application finds that the last session transaction status is <code>NOT COMMITTED</code> .                                      | Application returns the result to the user, or cleans up if needed, and resubmits with the LTXID on the new session in effect, LTXID-B 0.                                                                                                          | New LTXID-B 2 .. N<br>Also set using the JDBC callback when registered            |
| Application receives a recoverable error if it has decided to replay.                                                           | If the new request executes any commits, server returns commit messages with LTXID-B 2 and increasing.                                                                                                                                             | LTXID-C 0 on the new session.<br>Also set using the JDBC callback when registered |
| Application receives another recoverable error if it has decided to replay.                                                     | Application takes a new connection (with LTXID-D 0) and calls <code>GET_LTXID_OUTCOME</code> again with the LTXID of LAST session (LTXID-C N).                                                                                                     | LTXID-D 0 on the new session.<br>Also set using the JDBC callback when registered |

### 25.5.3 Connection-Pool LTXID Usage

Connection pools create a different use case for managing LTXIDs because connections and sessions are preestablished and shared. In the simplest model for connection pools and middle tiers, an LTXID exists on each session handle. It is associated with an application request at check-out from the connection pool, and is disassociated from the application request at check-in back to the pool. Between check-out and check-in, the LTXID on the session is exclusively held by that application request. After check-in, the LTXID belongs to an idle, pooled session. It is associated with the next application request that checks-out that connection.

Using Transaction Guard in this way:

- Can support duplicate detection and failover for the present HTTP request
- Might support basic replay by a third party container – replaying the last request following a nonrecoverable outage

### 25.5.4 Additional Requirements for Transaction Guard Development

Transaction Guard is a tool for developers to use after recoverable errors to provide a known outcome. It must be used when an error is returned indicating that the last session is dead.

The Transaction Guard APIs must *not* be used in the following cases:

- Do not use `GET_LTXID_OUTCOME` on the current session. It will return an error.
- Do not use `GET_LTXID_OUTCOME` against a session that did not receive a recoverable error—that is, a live session. It will block that session from committing.
- Do not use `GET_LTXID_OUTCOME` from a different user or to a different database. It will return an error.
- Do not obtain the LTXID and save it for use later, as opposed to using it immediately. The result of `GET_LTXID_OUTCOME` is valid only for the last open or completed transaction. If it is used with an earlier transaction on the same session, it will return an error.

## 25.6 Transaction Guard and Its Relationship to Application Continuity

Transaction Guard provides a unique identifier (LTXID) for each database transaction. This identifier can be used to query the commit outcome of the transaction, and can also be used to ensure that the transaction is applied only once. Transaction Guard is used by Application Continuity and automatically enabled by it, but it can also be enabled independently. Transaction Guard prevents the transaction being replayed by Application Continuity from being applied more than once. If the application has implemented an application-level replay, then it requires the application to be integrated with transaction guard to provide idempotence.

For pre-packaged replay, your application is able to use Application Continuity.

**See Also:**

- [Chapter 26, "Ensuring Application Continuity"](#)
- *Oracle Database 2 Day + Real Application Clusters Guide* for information about Transaction Guard and Application Continuity with Oracle RAC

---

## Ensuring Application Continuity

**Application Continuity** is a feature that enables the replay, in a nondisruptive and rapid manner, of a request against the database after a recoverable error that makes the database session unavailable. The request can contain transactional and nontransactional work. After a successful replay, the application can continue where that database session left off, instead of having users left in doubt not knowing what happened to their funds transfers, flight bookings, and so on, and avoiding the need to reboot mid-tier machines to recover from logon storms. With Application Continuity, the end user experience is improved by masking many outages, planned and unplanned, without the application developer needing to attempt to recover the request.

Without Application Continuity, it can be almost impossible for an application to mask outages in a safe way, for reasons that include the following:

- The state at the client remains at present time, with entered data, returned data, and variables cached.
- If a `COMMIT` has been issued, the `COMMIT` message is not durable.
- Checking a lost request is no guarantee that it will not `COMMIT` after being checked.
- Nontransactional database session state that the application needs to operate is lost.
- If the request can continue, the database and the database session must be in the right state.

Application Continuity, however, does this work for the application developer, thus masking many outages in a safe way.

Application Continuity improves developer productivity by attempting to mask outages that can be masked. However, applications still must include error handling for these cases:

- *Nonrecoverable* errors, such as invalid input data. (Application Continuity applies only to recoverable errors.)
- Recoverable errors when replay has encountered a restriction (see [Section 26.4, "Restrictions and Other Considerations for Application Continuity"](#)), such as use of concrete classes in the application, or when replay has not been able to restore the client-visible state to that on which the client may have made decisions so far.

Introduced in Oracle Database 12c Release 1 (12.1.0.1), Application Continuity strengthens the fault tolerance of systems and applications that use an Oracle database.

Currently, the terms "Application Continuity" and the named feature "Application Continuity for Java" are synonymous, for purposes of this chapter.

This chapter assumes that you are familiar with the major relevant concepts and techniques of the technology or product environment in which you are using Application Continuity, such as Oracle WebLogic Server, Oracle RAC, or Oracle Active Data Guard (Oracle ADG).

**Topics:**

- [About Application Continuity](#)
- [Application Continuity Operation and Usage](#)
- [Potential Side Effects of Application Continuity](#)
- [Restrictions and Other Considerations for Application Continuity](#)

**See Also:** [Chapter 25, "Using Transaction Guard"](#)

## 26.1 About Application Continuity

Application Continuity attempts to mask many recoverable database outages (when replay is successful) from applications and users by restoring the database session: the full session, including all states, cursors, variables, and the last transaction if there is one.

Application Continuity addresses the problem that arises when an application is trying to access the database and the database session becomes unavailable due to an unplanned or planned outage (timeout, network outage, instance failure, repair, configuration change, patch apply, and so on). Without Application Continuity in place, database recovery does not mask outages to applications and end users. In such scenarios, developers and end users must cope with exception conditions, and end users can be left not knowing what happened to their funds transfers, time sheets, orders, bill payments, and so on. Users might lose screens of uncommitted data, and must log in again and reenter that data. In the worst cases, the administrator might be forced to restart the middle tier to recover from the logon storm.

With Application Continuity, if the database session became unavailable, Application Continuity attempts to rebuild the session and any open transactions using the correct states; and if the transaction succeeded and need not be reexecuted, the successful return status is returned to the application. If replay is successful, the request can continue safely without risk of duplication. If replay cannot restore data that the application has already processed and potentially made decisions on, the database rejects the replay and the application receives the original error.

Application Continuity performs the recovery of in-flight transactions and database session state, while ensuring the transaction idempotence provided by Transaction Guard. Each database session is tagged with a logical transaction ID (LTXID), so the database recognizes whether each replay committed any transactions, and if it did commit any transactions, whether the work also ran to completion. While Application Continuity attempts to replay, the replay appears to the application as a delayed execution, or the application receives the commit response for the original transaction (if the last transaction had completed before the outage).

Application Continuity is supported for Oracle Real Application Clusters (Oracle RAC), Data Guard, Active Data Guard, and WebLogic Server in conjunction with the JDBC Thin Driver or the Universal Connection Pool. It is supported for both nonconsolidated database and for consolidated databases (with consolidated database failover at PDB level). (It is not currently supported for Golden Gate, Logical Standby, or DML redirection if using Active Data Guard.) You can think of Application

Continuity as a feature in Oracle Database that enables users of many Oracle technologies to benefit.

**Topics:**

- [Key Concepts for Application Continuity](#)

**See Also:** *Oracle Database 2 Day + Real Application Clusters Guide* for information about Transaction Guard and Application Continuity with Oracle RAC

### 26.1.1 Key Concepts for Application Continuity

This section describes several terms and concepts that you must understand to use Application Continuity. These terms are used throughout this chapter.

**request**

A request is a unit of work submitted from the application. It typically corresponds to the SQL and PL/SQL, and other database calls, of a single web request on a single database connection, and it is generally demarcated by the calls made to check-out and check-in the database connection from a connection pool. For recoverable errors, Application Continuity reestablishes the conversation state for a database session and repeats the request.

**recoverable error**

A recoverable error is an error that arises due to an external system failure, independent of the application session logic that is executing. Recoverable errors occur following planned and unplanned outages of foregrounds, networks, nodes, storage, and databases. The application receives an error code that can leave the application not knowing the status of the last operation submitted. Application Continuity reestablishes database sessions and resubmits the pending work for the class of recoverable errors.

Application Continuity does not resubmit work following call failures due to nonrecoverable errors. An example of a nonrecoverable error that would *not* be replayed is submission of invalid data values.

**commit outcome**

A transaction is committed by updating its entry in the transaction table. Oracle Database generates a redo-log record corresponding to this update and writes out this redo-log record. Once this redo-log record is written out to the redo log on disk, the transaction is considered committed at the database. From the client perspective, the transaction is considered committed when an Oracle message (termed *Commit Outcome*), generated after that redo is written, is received by the client. However, the commit message is not durable. (Transaction Guard, described in [Chapter 25](#), obtains the commit outcome available when it has been lost.)

**mutable objects**

Mutable objects are nondeterministic function that can obtain a new value every time it is called, and thus their results can change frequently. Mutable objects cause a problem for replay because the results can change at replay. Consider *sequence.NEXTVAL* and *SYSDATE*, often used in key values. If a primary key is built with values from these function calls, and is used in later foreign keys or other binds, at replay the same function result must be returned.

Application Continuity provides mutable object value replacement at replay for granted Oracle function calls to provide opaque bind-variable consistency. If the call uses database functions that are mutable, including `sequence.NEXTVAL`, `SYSDATE`, `SYSTIMESTAMP`, and `SYSGUID`, the original values returned from the function execution are saved and are reapplied at replay.

### **session state consistency**

After a `COMMIT` statement has executed, if state was changed in that transaction, it is not possible to replay the transaction to reestablish that state if the session is lost. When configuring Application Continuity, the applications are categorized depending on whether the session state after the initial setup is static or dynamic, and thus whether it is correct to continue past a `COMMIT` operation within a request.

- A session has **dynamic** state if the session state changes are not fully encapsulated by the initialization, and cannot be fully captured in a callback at failover. After the first transaction completes, failover is internally disabled until the next request begins.
- A session has a **static** state if all session state changes (for example, NLS settings and PL/SQL package state) occur as part of initialization, and can be encapsulated in a callback at failover.

## **26.2 Application Continuity Operation and Usage**

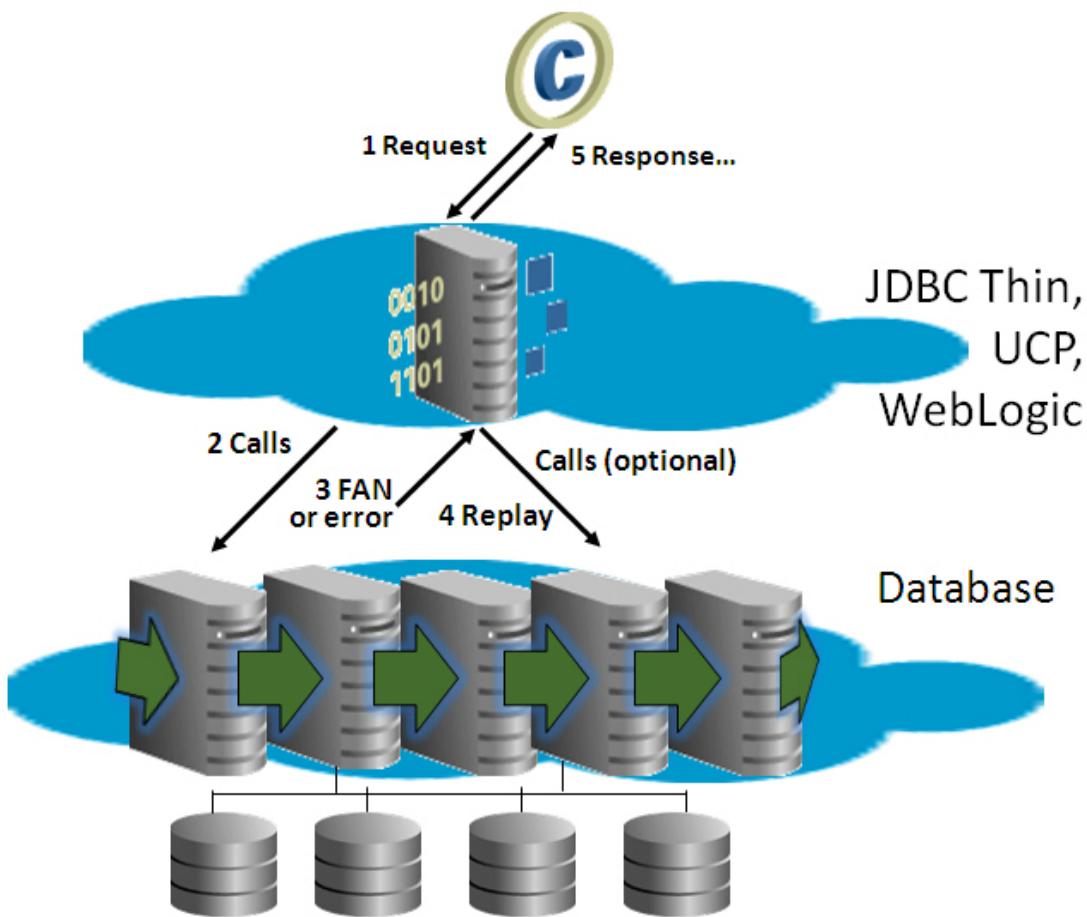
This section explains how Application Continuity works, and how you can use it in applications.

### **Topics:**

- [How Application Continuity Works for Database Sessions](#)
- [Actions for Using Application Continuity for Java](#)
- [Mutable Objects and Application Continuity](#)
- [Session State Consistency](#)
- [Static Session State Consistency](#)

### **26.2.1 How Application Continuity Works for Database Sessions**

If a recoverable error occurs and if replay is enabled, recovery of the database session is attempted. Application Continuity performs the major steps shown in [Figure 26-1](#). The major steps apply for both unplanned and planned outages, although specific steps vary depending on the type of outage. (For example, for planned outages further optimizations are possible, as explained in [Section 26.2.6, "Using Application Continuity for Planned Outages"](#).)

**Figure 26–1 How Application Continuity Works**

As shown in Figure 26–1:

1. The client application makes a request, which is passed to a middle tier (such as the JDBC Thin driver, Universal Connection Pool, or WebLogic Server or third party pool) or directly to the database using the JDBC replay driver.
2. The JDBC replay driver issues each call in the request.
3. A FAN unplanned or planned down interrupt or recoverable error occurs. FAN/FCF then aborts the dead physical session.
4. Application Continuity begins the replay and does the following:
  - a. Replaces the dead physical session with a new clean session and rewrites FAN in case a later error occurs during or after replay.
  - b. Prepares for replay by using Transaction Guard to determine the outcome of the in-flight transaction if one was open.
  - c. Optionally, calls back using a labeling callback or reconnect callback for the initial state.
  - d. Rebuilds the database session, recovering the transactional and nontransactional states, and validating at each step that the data and messages seen by the client driver are the same as those that the client may have seen and used to make a decision.
  - e. Ends the replay and returns to runtime mode.

- f. Submits the last queued call.

This is the last call made when the outage was discovered. During replay, only this call can execute a COMMIT. A COMMIT midway through rebuilding the session aborts replay (excluding autonomous transactions).

5. The response is returned to the application.

If replay succeeded, the application can continue with the problem masked. If not, the application must handle the original error.

The behavior of Application Continuity after a communication failure depends on the Oracle products and technologies involved. For example:

- If you use Oracle Real Application Clusters or an Active Data Guard farm, after the connection instance is reestablished on another running instance, Application Continuity attempts to rebuild the session and replay the last transaction if there is one in flight.
- If you use Oracle Data Guard and fail over to a standby site, Application Continuity connects to the failover instance and attempts to rebuild the session and replay the last transaction there if a transaction was in flight. (Application Continuity does not replay if the Data Guard switchover or failover has lost data, and if this is not Active Data Guard reader farm with approved lags.)
- If you are using Oracle RAC or Oracle RAC One and not using Data Guard, and if an outage causes a break in all public networks or causes the database or database session to shut down briefly, Application Continuity attempts to rebuild the session and replay the last transaction (if a transaction was in flight) against the database after connectivity is restored.

## 26.2.2 Actions for Using Application Continuity for Java

Application Continuity for Java is available for general use with the following Oracle technologies:

- JDBC Thin Oracle replay driver
- Universal Connection Pool
- WebLogic Server

A main appeal of Application Continuity for Java is its ability to mask many outages when using the Oracle stack with few or no application changes. Applications must be verified and tested to ensure that they are suitable for replay before releasing with Application Continuity for Java. If you must perform any actions, in most cases these do not involve core application source code changes, but rather configuration changes or providing a callback to disable replay for any code module.

The Application Continuity for Java solution is embedded in Oracle Universal Connection Pool (UCP), and in the Oracle WebLogic Server and Generic data sources. When you use the Oracle connection pools, request boundaries are implicitly marked at check-out and check-in delimiting the size of each replay. However, when you are using third-party connection pools with Oracle JDBC Thin, or when you are using UCP or WebLogic Server but not returning connections to the pools, you may need to perform actions to obtain the benefits of Application Continuity for Java.

Support for Application Continuity is integrated into many Oracle applications, so the features in such applications are used automatically if you set the Application Continuity-related service attributes. For your own applications, however, you also must follow the steps described in this section.

The main actions for ensuring automatic application continuity for an application are the following:

1. Determine whether the application uses Oracle JDBC concrete classes. For Application Continuity to be used, the deprecated concrete classes must be replaced. For information about the deprecation of concrete classes, including actions to take if an application uses them, see My Oracle Support Note 1364193.1 (<https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=1364193.1>).
2. Ensure that you have the necessary CPU and memory resources.
  - **CPU:** Application Continuity is managed on the client and server sides and needs additional CPU to operate.
 

At the client, CPU is used to build proxy objects and for garbage collection (GC).

At the server, CPU is used for validation. CPU overhead is reduced for platforms with current Intel and Sparc chips where validation is assisted in the hardware.
  - **Memory:** The replay driver requires more memory than the base driver because the calls are retained until the end of a request. At the end of the request, the calls are released to the garbage collector. This action differs from the base driver that releases closed calls.
 

The memory consumption of the replay driver depends on the number of calls per request. If this number is small, then the memory consumption of the replay driver is less, and comparable to the base driver.

To obtain the best performance, you must set the same value for both the `-Xmx` and `-Xms` parameters on the server. For example, if there is sufficient memory, then allocate 4 to 8 GB (or more) of memory for the Virtual Machine (VM), for example, by setting `-Xms4g` for 4 GB. If the `-Xms` parameter has a lower value, then the VM also uses a lower value from the operating system, and performance might suffer and garbage collection operations increase.
3. Determine whether the application borrows and returns connections from the WebLogic Server Pool or Universal Connection Pool for each request, or whether to add `beginRequest` and `endRequest` APIs to the application's own connection pool to identify request boundaries.
 

(Do not use `beginRequest` and `endRequest` anywhere other than at request boundaries (check-in). `endRequest` indicates that the request is complete, and that it is now stateless. Replay starts from the next `beginRequest`. If there is prior state, it must be reestablished in the callback.)
4. Decide whether to use an optional callback for initializing connections. When using Oracle WebLogic Server or the Universal Connection Pool, connection labeling is recommended. The labeling is used for both runtime and replay.
5. Determine whether the application requires, and therefore needs to configure keeping original values for, `SYSDATE`, `SYSTIMESTAMP`, and `SYS_GUID` and sequences during failover (see [Section 26.2.3, "Mutable Objects and Application Continuity"](#)).
6. Assess the application style for the `session_state_consistency` value, and set the appropriate value on the service:
  - If `session_state_consistency` is `Dynamic`, the application changes the environment or settings during the request. Replay is disabled after the first

COMMIT until the end of replay API is called. Dynamic is the default mode, appropriate for most applications.

- If session\_state\_consistency is Static, the application *never* changes the session state after initial setup. This mode is typical for database agnostic applications that do not use PL/SQL state and do not use ALTER part-way through transactions. Use this mode with caution, and only for "static" applications.

For more information, see [Section 26.2.4, "Session State Consistency,"](#) and [Section 26.2.4.2, "Static Session State Consistency."](#)

7. Determine whether replay must be explicitly disabled for any code path.

For example, replay may need to be disabled for requests using external PL/SQL actions (see [Section 26.2.2.8, "Disabling Replay in Application Continuity for Java"](#)).

8. Follow these configuration guidelines:

- Use Oracle Database 12c Release 1 (12.1.0.1) or later.
- Use Universal Connection Pool 12.1 (or later) or WebLogic Server 12.1.2 (or later) configured with the JDBC Replay data source; or for third party applications, including third party JDBC pools, use JDBC replay driver. Custom Java pools and standalone Java applications can use the JDBC Replay data source directly. When using custom Java pools and standalone applications, add the beginRequest and endRequest calls.
- If the application does not borrow and return from the Oracle connection pools, explicitly mark request boundaries. For example, if using custom JDBC pools, WebSphere, TomCat, JBOSS, or other pools, then call beginRequest at check-out and call endRequest at check-in. These APIs can also be used for standalone JDBC applications without a connection pool.
- Use a single pool with FAN/FCF from the WebLogic Data Source or UCP or third-party pool.
- Use a database service to connect; never use a SID or an instance name.
- Use a connection string that sets retries for new incoming connections and a delay between these retries.
- For the service, set FAILOVER\_TYPE to TRANSACTION, COMMIT\_OUTCOME to TRUE, and Notification to TRUE. Optionally to find the best connections to use, set GOAL to SERVICE\_TIME and CLB\_Goal to Short.

#### Topics:

- [When Is Application Continuity Transparent?](#)
- [Configuring Oracle JDBC for Application Continuity for Java](#)
- [Configuring Oracle Database for Application Continuity for Java](#)
- [Registering a Connection Initialization Callback in Application Continuity for Java \(optional\)](#)
- [Delaying the Reconnection in Application Continuity for Java](#)
- [Using Application Continuity for Planned Outages](#)
- [Running Without Application Continuity](#)
- [Disabling Replay in Application Continuity for Java](#)

- [Killing or Disconnecting a Session Without Replay](#)

**See Also:**

- *Oracle Database JDBC Developer's Guide* for more information about developing applications that include Application Continuity for Java
- *Oracle Universal Connection Pool for JDBC Developer's Guide* for information about using UCP
- *Oracle Real Application Clusters Administration and Deployment Guide* and *Oracle Database 2 Day + Real Application Clusters Guide* for DBA-related information

#### 26.2.2.1 When Is Application Continuity Transparent?

Application Continuity is transparent (performed automatically) for J2EE applications that use standard JDBC and that use Oracle connection pools (UCP or WLS). For applications with external actions (for example, autonomous transactions or using UTL\_HTTP to issue an SOA call), Application Continuity is still transparent *if* the application's correctness is preserved when these external actions are replayed after a failure.

For other scenarios in which Application Continuity is not transparent, the following infrastructure changes may be needed:

- If the connection pool or container does not use an Oracle connection pool, the application must use Application Continuity APIs to mark request boundaries. Request boundaries are needed to reclaim the memory used for holding calls, and to establish a point at which to resume recording following nonreplayable operations.
- If the application has requests that the application does not want repeated, the application can explicitly call an API to disable replay for those requests. Such calls are likely to be isolated to one or a few specialized pieces of application code.

#### 26.2.2.2 Configuring Oracle JDBC for Application Continuity for Java

You must use the `oracle.jdbc.replay.OracleDataSourceImpl` data source to obtain JDBC connections. This data source supports all the properties and configuration parameters of all the Oracle JDBC data sources, for example, the `oracle.jdbc.pool.OracleDataSource`.

You must remember the following points while using the connection URL:

- If the `REMOTE_LISTENER` setting for the database does not match the addresses in the `ADDRESS_LIST` at the client, then it does not connect, showing services cannot be found. So, the `REMOTE_LISTENER` setting for the database *must* match the addresses in the `ADDRESS_LIST` at the client:
  - If `REMOTE_LISTENER` is set to the SCAN Names, then the `ADDRESS_LIST` must use SCAN VIPs.
  - If the connect string uses the SCAN Name, then `REMOTE_LISTENERS` must be set to the SCAN Name.
  - If the connect string uses an `ADDRESS_LIST` of host VIPs, then `REMOTE_LISTENERS` must be set to an `ADDRESS` list including all SCAN VIPs and all host VIPs

---

**Note:** A motivation for using SCAN is location independence: the client need not be reconfigured when nodes are added or removed, or when databases change to running on different nodes.

---

- Set RETRY\_COUNT, CONNECT\_TIMEOUT, and TRANSPORT\_CONNECT\_TIMEOUT parameters in the connection string. This is a general recommendation for configuring the JDBC Thin driver connections, starting from Oracle Database 11g Release 2 (11.2.0.2). These settings improve acquiring new connections at runtime, at replay, and during work drains for planned outages.

The CONNECT\_TIMEOUT parameter is equivalent to the SQLNET.OUTBOUND\_CONNECT\_TIMEOUT parameter in the `sqlnet.ora` file and applies to the full connection. The TRANSPORT\_CONNECT\_TIMEOUT parameter is determined by the ADDRESS parameter. If the service is not registered for a failover or restart, then retrying is important when you use SCAN. (Using Easy Connect is *not* recommended because Easy Connect does not support the RETRY\_COUNT, CONNECT\_TIMEOUT, and TRANSPORT\_CONNECT\_TIMEOUT parameters.)

When the connect string uses host VIPs, then `REMOTE_LISTENERS` must include host VIPs. When the connect string uses the SCAN Name, then `REMOTE_LISTENERS` must be set to the SCAN Name, or include the SCAN VIPs and the host VIPs. Therefore, `REMOTE_LISTENERS` must be set to the SCAN Name unless any clients use host VIPs in their connect strings, in which case `REMOTE_LISTENERS` must be set to an `ADDRESS_LIST` of all SCAN VIPs and all host VIPs.

**See Also:**

- *Oracle Database Net Services Reference* for more information about local naming parameters
- *Oracle Real Application Clusters Administration and Deployment Guide*
- *Oracle Database 2 Day + Real Application Clusters Guide*
- *Single Client Access Name (SCAN) white paper at:* <http://www.oracle.com/technetwork/database/clustering/overview/scan-129069.pdf>
- My Oracle Support Note *11gR2 Grid Infrastructure Single Client Access Name (SCAN) Explained (Doc ID 887522.1)*

### 26.2.2.3 Configuring Oracle Database for Application Continuity for Java

Your Oracle Database configuration must include the following to use Application Continuity for Java:

- If you are using Oracle Real Application Clusters (Oracle RAC) or Oracle RAC One, Oracle Data Guard, or Oracle Active Data Guard, then ensure that FAN is configured with Oracle Notification System (ONS) to communicate with Oracle WebLogic Server or the Universal Connection Pool (UCP)
- Set the required properties on the service for replay and load balancing. For example, set:
  - `FAILOVER_TYPE = TRANSACTION` for using Application Continuity

- REPLAY\_INITIATION\_TIMEOUT =  $n$  for setting the duration in seconds to allow replay to start (where  $n$  might be, for example, 60, 300, 900, or 1800, depending on your needs)
- FAILOVER\_RETRIES = 30 for specifying the number of connection retries for each replay
- FAILOVER\_DELAY = 10 for specifying the delay in seconds between connection retries
- GOAL = SERVICE\_TIME -- if you are using Oracle RAC or Oracle GDS (Global Data Services), then this is a recommended setting
- CLB\_GOAL = SHORT -- if you are using Oracle RAC or Oracle GDS, then this is a recommended setting
- Do not use the database service, that is, the default service corresponding to the DB\_NAME or DB\_UNIQUE\_NAME. The use of the database service is not recommended for high availability, because this service cannot be enabled and disabled, and cannot be relocated on Oracle RAC or switched over to Oracle Data Guard. This service is reserved for Oracle Enterprise Manager Cloud Control (Cloud Control) and for DBAs.

#### 26.2.2.4 Registering a Connection Initialization Callback in Application Continuity for Java (optional)

Nontransactional session state (NTSS) is state of a database session that exists outside database transactions and is not protected by recovery. For applications that use stateful requests, the nontransactional state is reestablished as the session is rebuilt by Application Continuity.

For applications that set state only at the beginning of a request, or for stateful applications that gain performance benefits from using connections with a preset state, choose one of these callback options:

- [No Callback](#)
- [Connection Labeling](#)
- [Connection Initialization Callback](#)

**26.2.2.4.1 No Callback** In this scenario, the application builds up its own state during each request.

**26.2.2.4.2 Connection Labeling** Connection Labeling is a generic pool feature that is recommended for its excellent performance. When Connection Labeling is present, Application Continuity uses it.

This scenario is applicable to Universal Connection Pool (UCP) and Oracle WebLogic server. The application can be modified to take advantage of the preset state on connections. Connection Labeling APIs determine how well a connection matches, and use a callback to populate the gap when a connection is borrowed.

**See Also:** *Oracle Universal Connection Pool for JDBC Developer's Guide*

**26.2.2.4.3 Connection Initialization Callback** In this scenario, the replay driver uses an application callback to set the initial state of the session during runtime and replay. The JDBC replay driver provides an optional connection initialization callback interface and methods to register and unregister connection initialization callbacks in the `oracle.jdbc.replay.OracleDataSource` interface.

When registered, the initialization callback is executed at each successful reconnection following a recoverable error. An application is responsible for ensuring that the initialization actions are the same as that on the original connection before failover. If the callback invocation fails, then replay is disabled on that connection. Use the connection initialization callback only when the application has not implemented Connection Labeling.

#### 26.2.2.5 Delaying the Reconnection in Application Continuity for Java

By default, when JDBC Replay Driver initiates a failover, the driver attempts to recover the in-flight work at an instance where the service is available. For recovering the work, the driver must establish a good connection with the instance. This reconnection can take some time if the database or the instance must be restarted before the service is relocated and published. So, the failover must be delayed until the service is available from another instance or database.

You must use the FAILOVER\_ATTEMPTS and FAILOVER\_DELAY parameters to manage reconnecting. These parameters can work well in conjunction with a planned outage, for example, an outage that may make a service unavailable for several minutes. While setting the FAILOVER\_DELAY and FAILOVER\_ATTEMPTS parameters, check the value of the REPLAY\_INITIATION\_TIMEOUT parameter first. The default value for this parameter is 900 seconds. A high value for the FAILOVER\_DELAY parameter can cause replay to be canceled.

| Parameter Name    | Possible Value                 | Default Value |
|-------------------|--------------------------------|---------------|
| FAILOVER_ATTEMPTS | Positive integer zero or above | 30            |
| FAILOVER_DELAY    | Time in seconds                | 10            |

The following examples show various failover scenarios:

- [Creating Services on Oracle RAC](#)
- [Modifying Services on Single-Instance Databases](#)

**26.2.2.5.1 Creating Services on Oracle RAC** If you are using Oracle RAC or Oracle RAC One, then use the SRVCTL command to create and modify services in the following way:

- [For Policy Managed Databases](#)
- [For Administrator Managed Databases](#)

##### For Policy Managed Databases

```
srvctl add service -db codedb -service GOLD -serverpool ora.Srvpool -clbgoal SHORT
-rlbgoal SERVICE_TIME -failoverretry 30 -failoverdelay 10 -commit_outcome TRUE
-failovertype TRANSACTION -replay_init_time 1800 -retention 86400 -notification
TRUE
```

##### For Administrator Managed Databases

```
srvctl add service -db codedb -service GOLD -preferred serv1 -available serv2
-clbgoal SHORT -rlbgoal SERVICE_TIME -failoverretry 30 -failoverdelay 10 -commit_
outcome TRUE -failovertype TRANSACTION -replay_init_time 1800 -retention 86400
-notification TRUE
```

**26.2.2.5.2 Modifying Services on Single-Instance Databases** If you are using a single-instance database, then use the DBMS\_SERVICE package to modify services in the following way:

```
declare
 params dbms_service.svc_parameter_array;
begin
 params('FAILOVER_TYPE') := 'TRANSACTION';
 params('REPLAY_INITIATION_TIMEOUT') := 1800;
 params('RETENTION_TIMEOUT') := 86400;
 params('FAILOVER_DELAY') := 10;
 params('FAILOVER_RETRIES') := 30;
 params('commit_outcome') := 'true';
 params('aq_ha_notifications') := 'true';
 dbms_service.modify_service('[your service]', params);
end;
/
```

### 26.2.2.6 Using Application Continuity for Planned Outages

For planned outages the recommended approach is to drain requests from Oracle connection pools in combination with Application Continuity for those requests that do not complete. Instances do need to be stopped to switch over to the patched software. This has the least impact when there is minimal recovery to complete.

The steps are the following:

1. Use any FAN-aware pool: OCI, UCP, WebLogic Server, or ODP.Net.  
The FAN planned event drains at request boundaries.
2. Use `srvctl relocate` to relocate the service from the instance without disrupting the sessions (no `-force` flag), or for a uniform service use `srvctl stop service` at the instance (no `-force` flag).
  - For Oracle RAC One, use `relocate database` (no `-force` flag).

The FAN planned event clears the idle sessions immediately and marks the active sessions to be released at check in (end of request). This drains the sessions from the instance without disrupting work.

3. If not all sessions have checked in and the time to stop the instance has been reached, stop the instance (abort).

For Application Continuity enabled pools (UCP and WLS), and any pool that adds `beginRequest/endRequest` on the JDBC Thin replay driver, Application Continuity attempts to recover those remaining sessions.

4. Restart the instance and service.

Runtime load balancing, when enabled, balances the sessions back to the restored instance at the next request boundaries.

### 26.2.2.7 Running Without Application Continuity

Sometimes Application Continuity is not in effect, through deliberate choice, error, or oversight. Application Continuity is not in effect when it has not been started or when it has been disabled. If it has been disabled, it remains so through the `endRequest` call.

Application Continuity is not started when the service property `FAILOVER_TYPE` does *not* have the value set to `TRANSACTION`. For a planned outage, set the `FAILOVER_TYPE` value to `TRANSACTION` beforehand; the setting applies to new connections, and existing connections retain their original service value.

Application Continuity is disabled when any of the following occurs:

- The application executes a statement that is restricted for Application Continuity (for example, if it uses Oracle JDBC concrete classes).
- Application Continuity is explicitly disabled using `disableReplay` (see [Section 26.2.2.8](#)).
- A `COMMIT` statement is issued when the service parameter `session_state_consistency` is set to `Dynamic` (the default).
- An `endRequest` statement is issued until the next `beginRequest` is issued.
- The session is killed or disconnected and the `NOREPLAY` keyword is specified (see [Section 26.2.2.9](#)).
- A request issues an `ALTER SYSTEM` or `ALTER DATABASE` statement.

### **26.2.2.8 Disabling Replay in Application Continuity for Java**

By default, the JDBC replay driver replays following a recoverable error. If the application has requests that the application does not want repeated, the application can explicitly call an API to disable replay for those requests. For example, if the application uses `UTL_SMTP` and does not want messages repeated, the `disableReplay` API is in effect for the request that must be disabled. All other requests continue to be replayed.

For applications with external actions (for example, autonomous transactions or using `UTL_HTTP` to issue an SOA call), Application Continuity remains transparent if the application's correctness is preserved when these external actions are replayed after a failure.

The following are scenarios to consider before configuring an application for replay:

- [Application Calls Autonomous Transactions, External PL/SQL, or Java Actions that Should Not Be Repeated](#)
- [Application Synchronizes Independent Sessions](#)
- [Application Uses Time at the Middle Tier in the Execution Logic](#)
- [Application Assumes that ROWIDs Do Not Change](#)
- [Application Assumes that Location Values Do Not Change](#)

#### **See Also:**

- [Section 26.3, "Potential Side Effects of Application Continuity"](#)
- [Section 26.4, "Restrictions and Other Considerations for Application Continuity"](#)

#### **26.2.2.8.1 Application Calls Autonomous Transactions, External PL/SQL, or Java Actions that Should Not Be Repeated**

Autonomous transactions, external PL/SQL calls, and Java callouts can have side effects that are separate from the main transaction, and these side effects are replayed unless you specify otherwise.

Examples of side effects separate from the main transaction include writing to an external table, sending email, forking sessions out of PL/SQL (including calls to `UTL_HTTP`, `UTL_URL`, `UTL_FILE`, `UTL_FILE_TRANSFER`, `UTL_SMPT`, `UTL_TCP`, `UTL_MAIL`, `DBMS_PIPE`, or `DBMS_ALERT`) or Java (including executing a shell script in the form `Process proc = rt.exec(command);`), transferring files, and accessing external URLs. Actions such as these leave persistent side effects. PL/SQL messaging and Java callouts can leave persistent results behind. For example, if a user walks away part

way through some work without committing and the session times out or the user issues Ctrl+C, the foreground or a component fails; the main transaction rolls back while the side effects may have been applied. (For more information about side effects, see [Section 26.3, "Potential Side Effects of Application Continuity"](#).)

Application developers must decide whether to allow replay for external actions. Examples include using UTL\_HTTP to issue an SOA call, or UTL\_SMTP to send a message, or UTL\_URL to access a website. If such external actions must not be replayed, use the disableReplay API.

**26.2.2.8.2 Application Synchronizes Independent Sessions** You must not configure an application for replay if the application synchronizes independent sessions using volatile entities that are held until COMMIT, ROLLBACK, or session loss. For example, the application might synchronize multiple sessions connected to several data sources that are otherwise inter-dependent using resources such as a database lock. This synchronization may be acceptable if the application is only serializing these sessions and understands that any session may fail. However, if the application assumes that a lock or any other volatile resource held by one data source implies exclusive access to data on the same or a separate data source from other connections, then this assumption may be invalidated when replaying.

During replay, the driver is not aware that the sessions are dependent on one session holding a lock or other volatile resource. You can also use pipes, buffered queues, stored procedures taking a resource (such as a semaphore, device, or socket) to implement the synchronization that are lost by failures.

**26.2.2.8.3 Application Uses Time at the Middle Tier in the Execution Logic** You must not configure an application for replay if the application uses the wall clock at the middle tier as part of the execution logic. The JDBC replay driver does not repeat the middle-tier time logic, but uses the database calls that execute as part of this logic. For example, an application using middle-tier time might assume that a statement executed at Time T1 is not reexecuted at Time T2, unless the application explicitly does so.

**26.2.2.8.4 Application Assumes that ROWIDs Do Not Change** If an application caches ROWIDs, then access to these ROWIDs might be invalidated due to database changes. Although a ROWID uniquely identifies a row in a table, a ROWID might change its value in the following situations:

- The underlying table is reorganized.
- An index is created on the table.
- The underlying table is partitioned.
- The underlying table is migrated.
- The underlying table is exported and imported using EXP/IMP/DUL.
- The underlying table is rebuilt using Golden Gate or Logical Standby or other replication technology.
- The database of the underlying table is flashed back or restored.

It is bad practice, in general, for an application to store ROWIDs for later use because the corresponding row might either not exist or contain completely different data.

**26.2.2.8.5 Application Assumes that Location Values Do Not Change** SYSCONTEXT options comprise a location-independent set such as National Language Support (NLS) settings, ISDBA, CLIENT\_IDENTIFIER, MODULE, and ACTION, and a location-dependent set

that uses physical locators. Typically, an application does not use the physical identifier, except in testing environments. If physical locators are used in mainline code, then the replay finds the mismatch and rejects it. However, it is acceptable to use physical locators in callbacks.

### Example

```
select
 sys_context('USERENV', 'DB_NAME')
 ,sys_context('USERENV', 'HOST')
 ,sys_context('USERENV', 'INSTANCE')
 ,sys_context('USERENV', 'IP_ADDRESS')
 ,sys_context('USERENV', 'ISDBA')
 ,sys_context('USERENV', 'SESSIONID')
 ,sys_context('USERENV', 'TERMINAL')
 ,sys_context('USERENV', 'SID')
from dual;
```

#### 26.2.2.9 Killing or Disconnecting a Session Without Replay

If Application Continuity is configured and if a DBA kills or disconnects a session by using the `ALTER SYSTEM KILL SESSION` or `ALTER SYSTEM DISCONNECT SESSION` statement, Application Continuity by default attempts to recover the session. However, if you do *not* want the session to be replayed, use the `NOREPLAY` keyword:

```
alter system kill session 'sid, serial#, @inst' noreplay;
alter system disconnect session 'sid, serial#, @inst' noreplay
```

To kill all sessions executing on the local instance (rather than only one session) and not have the sessions replayed, you can also use the `DBMS_SERVICE.DISCONNECT_SESSION` PL/SQL procedure and specify `NOREPLAY` for the `disconnect_option` parameter.

#### See Also:

- *Oracle Database SQL Language Reference* for information about the `ALTER SYSTEM` statements
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SERVICE.DISCONNECT_SESSION` procedure

#### 26.2.3 Mutable Objects and Application Continuity

When a request is replayed, the default and desired treatment of mutable objects can vary. A **mutable object** is a nondeterministic function that can obtain a new value every time it is called. An example of a mutable object use is a call to the `SYSTIMESTAMP` function. Client applications using Application Continuity can determine whether to keep the original value for mutable functions if the request is replayed.

Support for keeping mutable object values is currently provided for `SYSDATE`, `SYSTIMESTAMP`, `SYS_GUID`, and `sequence.NEXTVAL`. If the original values are not kept and if different values for these mutable objects are returned to the client, replay is rejected because the client sees different results. If the application can use original values, configure mutable objects using the `KEEP` clause for owned sequences and `GRANT KEEP` for other users. (Most applications need sequence values to be kept at replay, for bind variable consistency.)

---

**Note:** Keeping `SYS_GUID` values is supported only for serial execution plans. When parallel query is used, Application Continuity is not able to restore original values for `SYS_GUID`.

---

**Table 26–1** shows examples of the treatment of mutable objects by products during replay. (Actual implementation depends on specific products and releases.)

**Table 26–1 Example Treatment of Mutable Objects by Products During Replay**

| Mutable Object                                         | Product 1        | Product 2        | Product 3        |
|--------------------------------------------------------|------------------|------------------|------------------|
| <code>SYSDATE</code> , <code>SYSTIMESTAMP</code>       | Original         | Original         | Current          |
| Sequence <code>NEXTVAL</code> and <code>CURRVAL</code> | Original         | Original         | (Not applicable) |
| <code>SYS_GUID</code>                                  | Original         | (Not applicable) | (Not applicable) |
| LOB access                                             | Fail on mismatch | (Not applicable) | (Not applicable) |

To allow Application Continuity to keep and use original function results at replay:

- The database user running the application might have the `KEEP DATE TIME` and `KEEP SYSGUID` privileges granted, and the `KEEP SEQUENCE` object privilege on each sequence whose value is to be kept. For example:

```
grant KEEP DATE TIME to user2;
grant KEEP SYSGUID to user2;
grant KEEP SEQUENCE on sales.seq1 to user2;
```

---

**Note:** `GRANT ALL ON <object>` does *not* include (that is, does not grant the access provided by) the `KEEP DATE TIME` and `KEEP SYSGUID` privileges and the `KEEP SEQUENCE` object privilege.

---

Grant privileges related to mutable object support only to application users, and to each application user, grant only the necessary privileges.

Do *not* grant DBA privileges to database users running applications for which you want replay to be enabled.

- Sequences in the application can use the `KEEP` attribute, which keeps the original values of `sequence.NEXTVAL` for the sequence owner, so that the keys match during replay. Most applications need sequence values to be kept at replay. The following example sets the `KEEP` attribute for a sequence (in this case, one owned by the user executing the statement; for others, use `GRANT KEEP SEQUENCE`):

```
SQL> CREATE SEQUENCE my_seq KEEP;
SQL> -- Or, if the sequence already exists but without KEEP:
SQL> ALTER SEQUENCE my_seq KEEP;
```

---

**Note:** Specifying `ALTER SEQUENCE ... KEEP/NOKEEP` applies to the owner of the sequence. It does not affect other users (not the owner) that have the `KEEP SEQUENCE` object privileges. If you want `NOKEEP` for all users, be sure not to grant the `KEEP SEQUENCE` object privilege to these users (or to revoke it from each if they have been granted it).

---

- To keep function results (for named functions) at replay, the DBA must grant `KEEP` privileges to the user invoking the function. This security restriction ensures that it

is valid for replay to save and restore function results for code that is not owned by that user.

The following additional considerations apply to granting privileges on mutable objects:

- If a user has the KEEP privilege granted on mutable object values, then the objects inherit mutable access when the SYS\_GUID, SYSDATE, and SYSTIMESTAMP functions are called.
- If the KEEP privilege is revoked on mutable values on a sequence object, then SQL or PL/SQL blocks using that object do not allow mutable collection or application for that sequence.
- If granted privileges are revoked between runtime and failover, then the mutable values that are collected are not applied for replay.
- If new privileges are granted between runtime and failover, then mutable values are not collected and these values are not applied for replay.

**See Also:** *Oracle Database SQL Language Reference* for information about the ALTER SEQUENCE and GRANT statements

## 26.2.4 Session State Consistency

Session state consistency describes how nontransactional state is changed during a request. Examples of session state are NLS settings, optimizer preferences, event settings, PL/SQL global variables, temporary tables, advanced queues, LOBs, and result cache. If nontransactional values change after the request starts, use the default value, Dynamic.

After a COMMIT has executed, if the state was changed in that transaction, it is not possible to replay the transaction to reestablish that state if the session is lost.

Applications can be categorized depending on whether the session state after the initial setup or is static or dynamic, and hence whether it is correct to continue past a COMMIT operation.

Dynamic mode is appropriate for almost all applications. If you are unsure, use Dynamic mode. If your customers or users can modify your application, you must use Dynamic mode.

### Topics:

- [Dynamic Session State Consistency](#)
- [Static Session State Consistency](#)

### 26.2.4.1 Dynamic Session State Consistency

A session has **dynamic** state if the session state changes are not fully encapsulated by the initialization, and cannot be fully captured in a callback at failover. Once the first transaction completes, failover is internally disabled until the next request starts. In Dynamic session state consistency mode, state changes occur during the request and replay is enabled at the call to beginRequest.

Set the session state consistency mode to **Dynamic** if the nontransactional session state changes while transactions are executing. Examples of nontransactional session state that can change at runtime are ALTER SESSION, PL/SQL global variables, SYS\_CONTEXT, and temporary table contents. If the application changes nontransactional state inside transactions and commits, this state cannot be replayed and the state setting must be

Dynamic. When using Dynamic mode for Application Continuity, replay is disabled at COMMIT until the next request begins. Dynamic is the default value.

Figure 26-2 shows the nontransactional session state (NTSS) changes during a request when the session state consistency mode is Dynamic.

**Figure 26-2 Dynamic Mode: Nontransaction State Can Change During Request**

## Dynamic Mode: NTSS changes during request

- enable replay at beginRequest
- disable replay at COMMIT, endRequest, restricted call



As shown in Figure 26-2, replay (that is, Application Continuity) is enabled at the beginRequest call, and is disabled on a COMMIT or an endRequest call. (The use of a restricted call also disables Application Continuity, but this is not shown in the figure.) Also shown in Figure 26-2 is the step logic for three application scenarios: no transaction, a transaction with COMMIT as the last statement, and a transaction with an embedded COMMIT statement.

For the request with **no transaction**, the logic is as follows:

1. Check out.
2. Replay is disabled at endRequest, at a restricted call, and for an explicit disableReplay call.
3. Begin request and enable replay.
4. Issue one or more SELECT statements and perhaps other PL/SQL statements.
5. (Other actions, indicated by a gap.)
6. Check in.
7. End request and disable replay.

For the request with **a transaction with COMMIT as the last statement**, the logic is as follows:

1. Check out.
2. Replay is disabled at COMMIT, at endRequest, at a restricted call, and for an explicit disableReplay call.
3. Begin request and enable replay.

4. Issue one or more SELECT statements and perhaps other PL/SQL statements.
5. The transaction begins.
6. (Other actions, indicated by a gap.)
7. Commit (which disables replay).
8. Check in.
9. End request.

For the request with a **transaction with an embedded COMMIT statement**, the logic is as follows:

1. Check out.
2. Replay is disabled at COMMIT, at endRequest, at a restricted call, and for an explicit disableReplay call.
3. Begin request and enable replay.
4. Issue one or more SELECT statements and perhaps other PL/SQL statements.
5. The transaction begins.
6. (Other actions, indicated by a gap.)
7. Commit (which disables replay).
8. (Other actions, indicated by a gap during which Application Continuity is not covering the application.)
9. Check in.
10. End request.

#### 26.2.4.2 Static Session State Consistency

Set the session state consistency mode to `Static` if all nontransactional state changes, such as NLS settings, `SYS_CONTEXT`, PL/SQL variables, and optimizer preferences, are set as part of the initialization once per request, and if this session state does not change during transactions. The settings can be established once per connection at connection establishment with UCP labeling, for example, or at each checkout from a pool. These settings must be repeated in the replay callback. When using `Static` mode for Application Continuity, transactional failover continues beyond the first transaction of a request.

Static mode is not supported for applications that use calls that change nontransactional state in requests. Specific examples of such calls include:

- PL/SQL subprograms
- `SYS_CONTEXT`
- Hints
- DDL operations
- Autocommit
- `ALTER SESSION`
- `ALTER SYSTEM`

Specify static mode with caution. Use static mode only when the application does not change the NTSS (nontransactional session state) inside transactions. Declaring the session state consistency mode as `Static` indicates that it is safe to continue beyond

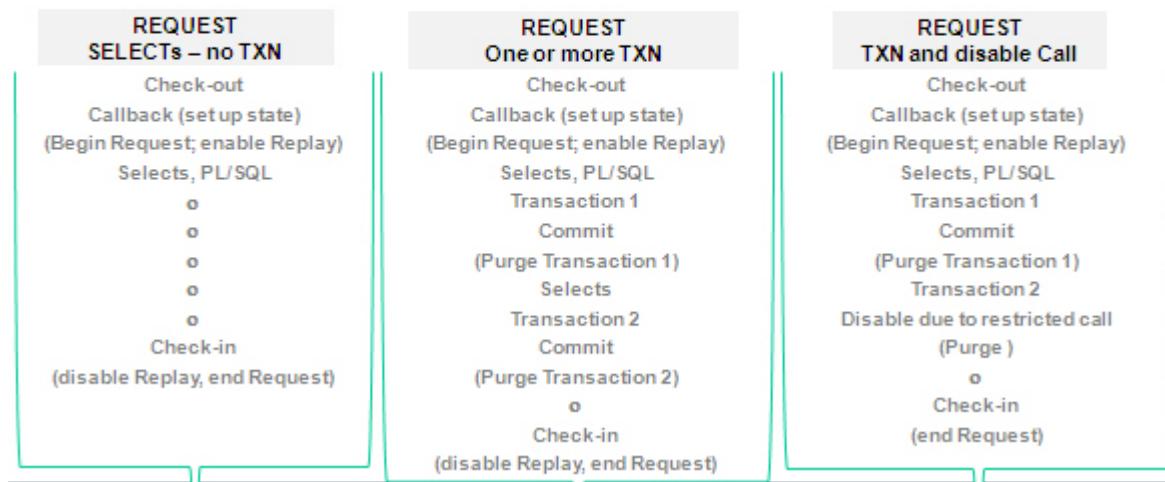
the first COMMIT in a request. Dynamic mode is appropriate for most applications. Do *not* use static mode if users or customers can modify or customize the application.

Figure 26-3 shows the NTSS (nontransactional session state) remaining constant (that is, not changing) during a request when the session state consistency mode is Static.

**Figure 26-3 Static Mode: Nontransactional State Never Changes During Request**

## Static Mode: NTSS constant during request

- enable replay at beginRequest
- disable replay at endRequest, restricted call



As shown in Figure 26-3, replay (that is, Application Continuity) is enabled at the beginRequest call, and is disabled on a restricted call or on a disableReplay or endRequest call. Also shown in Figure 26-3 is the step logic for three application scenarios: no transaction, one or more transactions each ending with COMMIT as the last statement, and a transaction with a COMMIT statement followed by a transaction with a restricted call that disables Application Continuity.

For the request with **no transaction**, the logic is as follows:

1. Check out.
2. Replay is disabled at endRequest, at a restricted call, and for an explicit disableReplay call.
3. Begin request and enable replay.
4. Issue one or more SELECT statements and perhaps other PL/SQL statements.
5. (Other actions, indicated by a gap.)
6. Check in.
7. End request and disable replay.

For the request with **one or more transactions (each with COMMIT as the last statement)**, the logic is as follows:

1. Check out.
2. Replay is disabled at COMMIT, at endRequest, at a restricted call, and for an explicit disableReplay call.

3. Begin request and enable replay.
4. Issue one or more SELECT statements and perhaps other PL/SQL statements.
5. The transaction begins.
6. The transaction commits.
7. The transaction is purged.
8. (For each additional transaction, steps 4 through 7 occur.)
9. (Other actions, indicated by a gap.)
10. Check in.
11. End request.

For the request with a **transaction with a COMMIT followed by a transaction with a restricted call**, the logic is as follows:

1. Check out.
2. Replay is disabled at COMMIT, at endRequest, at a restricted call, and for an explicit disableReplay call.
3. Begin request and enable replay.
4. Issue one or more SELECT statements and perhaps other PL/SQL statements.
5. The transaction begins.
6. The transaction commits.
7. The transaction is purged.
8. The second transaction begins.
9. The transaction makes a restricted call, which causes Application Continuity to be disabled.
10. The transaction is purged.
11. (Other actions, indicated by a gap.)
12. Check in.
13. End request.

## 26.3 Potential Side Effects of Application Continuity

When a session is rebuilt, all states are rebuilt. This includes reexecuting statements that leave side effects. These side effects might be exactly what is required, such as writing a report or completing some auditing. However, the actions that are replayed to build the state might include some for which you want to take action to accommodate or mitigate the effects of the replay.

Application Continuity replays PL/SQL chronologically to restore database state. This serves to rebuild the session as if the user submission was delayed. Most applications want the full state rebuilt as if the submission was repeated, such as writing a report or completing some auditing. However, the actions that are replayed to build the state might include some for which you want to take action to accommodate or mitigate the effects of the replay. Some applications elect to use the disableReplay API for requests that contain calls that they do not want to repeat.

If requests have external actions that use a messaging mechanism (such as UTL\_SMTP, UTL\_HTTP, or UTL\_FILE), then review the requests to decide if they must be replayed.

Examples of actions that create side effects include the following:

- Autonomous transactions (independent transactions that can be called from other transactions, as explained in [Section 6.8, "Autonomous Transactions"](#))
- DBMS\_ALERT calls (email or other notifications)
- DBMS\_FILE\_TRANSFER calls (copying files)
- DBMS\_PIPE and RPC calls (to external sources)
- UTL\_FILE calls (writing text files)
- UTL\_HTTP calls (making HTTP callouts)
- UTL\_MAIL calls (sending email)
- UTL\_SMTP calls (sending SMTP messages)
- UTL\_TCP calls (sending TCP messages)
- UTL\_URL calls (accessing URLs)

**See Also:** [Section 26.2.2.8, "Disabling Replay in Application Continuity for Java"](#) for information about disabling replay and for related considerations

## 26.4 Restrictions and Other Considerations for Application Continuity

The following restrictions and other considerations apply to Application Continuity for Java:

- It applies only to JDBC Thin connections (JDBC OCI is not supported).
- For applications using JDBC, there is no support for `oracle.sql` deprecated concrete classes: BLOB, CLOB, BFILE, OPAQUE, ARRAY, STRUCT, or ORADATA. (See My Oracle Support Note 1364193.1, *New Jdbc Interfaces for Oracle types*: <https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=1364193.1>)
- For JDBC streams arguments, replay is on a "best effort" basis. For example, if the application is using physical addresses, the address has gone with the outage and cannot be repositioned. JDBC stream setters (such as `setBinaryStream`), for example, cause replay to be disabled.
- The replay target database must have the same database ID, pluggable database ID, ancestors, and descendants as the source database.

Application Continuity does not replay if the target is a different database or if it is the same database or same pluggable database but with data loss, such as one flashed back, recovered incompletely by media recovery, or opened by Oracle Data Guard earlier in time.

- If a statement cache at the application server level is enabled (for example, the WebLogic or third-party application server statement cache), this must be disabled when the replay is used. Instead, configure the JDBC statement cache, which performs better because it is optimized for JDBC and Oracle and because it supports Application Continuity. Use `oracle.jdbc.implicitstatementcachesize=nnn`.
- Replay is not supported for applications developed using Oracle XA.
- Replay is disabled if a request issues an `ALTER SYSTEM` or `ALTER DATABASE` statement.

- Replay is not supported if you are using Active Data Guard with read/write database links to another database.
- Replay does not apply for failure of a parallel query call when this is a statement-level failure. For example, replay would not occur after an ORA-12805 error ("parallel query server died unexpectedly") for a call failure encountered during an instance or node failure or memory issue.

---

**Note:** If you are creating clones of databases by splitting disk images (for example, BCVs) or by cloning so it is a "different" database to make a logical standby or logical copy that is not a physical or Active Data Guard database, **nid** **must** be used to change the DBID to differentiate the databases. For information about **nid** program usage, see these My Oracle Support notes: *How to Change the DBID and the DBNAME by using NID* (Doc Id 224266.1) and *Changing DBNAME and DBID of Oracle RAC Database Using NID* (Doc Id 464922.1).

---

---

---

# Index

## Numerics

---

32-bit IEEE 754 format, 7-8  
64-bit IEEE 754 format, 7-8

## A

---

Abstract Data Type (ADT), 11-10  
    native floating-point data types in, 7-12  
    resetting evolved, 24-8  
ACCESSIBLE BY clause  
    in package specification, 11-3  
    stored subprogram and, 11-2  
actual object, 24-10  
actualization, 24-10  
ADDM (Automatic Database Diagnostic Monitor), 2-7  
address of row (rowid), 7-22  
administrators, restricting with Oracle Database Vault, 3-3  
ADT  
    *See* Abstract Data Type (ADT)  
AFTER SUSPEND trigger, 6-40  
agent, 20-3  
aggregate function, 11-1  
altering application online  
    *See* edition-based redefinition (EBR)  
analytic function, 1-7  
ancestor edition, 24-10  
ANSI data type, 7-21  
ANYDATA data type, 7-20  
ANYDATASET data type, 7-20  
AP (application program), 19-4  
application architecture, 17-2  
application continuity, 26-1  
application domain index, 9-2  
application program (AP), 19-4  
application SQL, 24-25  
APPLYING\_CROSSEDITION\_TRIGGER function, 24-28  
AQ (Oracle Advanced Queuing), 20-2  
archive  
    *See* Flashback Data Archive  
ARGn data type, 7-24  
arithmetic operation  
    with datetime data type, 7-16

    with native floating-point data type, 7-10  
assignment, reported by PL/Scope, 12-9  
auditing  
    available options, 3-6  
    unified auditing, 3-6  
auditing policy, editioning view and, 24-35  
AUTHID clause  
    in package specification, 11-3  
    stored subprogram and, 11-2  
AUTHID property  
    of invoked subprogram, 11-20  
    of PL/SQL unit, 11-4, 11-26  
Automatic Database Diagnostic Monitor (ADDM), 2-7  
Automatic Undo Management system, 16-1  
Automatic Workload Repository (AWR), 2-19  
autonomous transaction, 6-32  
    nonblocking DDL statement in, 6-32  
    trigger as, 6-40  
auto-tuning OCI client statement cache, 2-23

## B

---

backward compatibility  
    LONG and LONG RAW data types for, 7-19  
    RESTRICT\_REFERENCES pragma for, 11-30  
BATCH commit redo option, 6-6  
benchmark, 2-4  
BFILE data type, 7-18  
binary floating-point number, 7-8  
binary format, 7-8  
binary large object (BLOB) data type, 7-18  
BINARY\_DOUBLE data type, 7-7  
BINARY\_FLOAT data type, 7-7  
BINARY\_INTEGER data type  
    *See* PLS\_INTEGER data type  
bind variable, 2-10  
BLOB data type, 7-18  
block, PL/SQL, 11-1  
blocking DDL statement, 6-31  
BOOLEAN data type, 11-8  
branch, 19-3  
built-in data type  
    *See* SQL data type  
built-in function  
    *See* SQL function

bulk binding, 11-15  
business rule, 10-1

## C

---

C external subprogram  
callback with, 18-36  
global variable in, 18-40  
interface between PL/SQL and, 18-10  
invoking, 18-29  
loading, 18-4  
passing parameter to, 18-15  
publishing, 18-12  
running, 18-26  
service routine and, 18-30  
*See also* external subprogram

call specification  
for external subprogram, 18-3  
in package, 11-3  
location of, 18-12

CALL statement, 18-26

calling subprogram  
*See* invoking subprogram

cascading invalidation, 23-5

CHANGE DUPKEY\_ERROR\_INDEX hint, 24-28

CHAR data type, 7-5

character data type class, 23-16

character data types, 7-5

character large object (CLOB) data type, 7-18

CHECK constraint  
compared to NOT NULL constraint, 10-15  
designing, 10-15  
multiple, 10-15  
naming, 10-17  
restrictions on, 10-14  
when to use, 10-14

client configuration parameter, 2-19

client notification, 20-2

client result cache, 2-11

client statement cache auto-tuning (OCI client session feature), 2-23

CLIENT\_RESULT\_CACHE\_LAG server initialization parameter, 2-19

CLIENT\_RESULT\_CACHE\_SIZE server initialization parameter, 2-18

client/server architecture, 17-2

CLOB data type, 7-5, 7-18

coarse-grained invalidation, 23-5

collection, 11-9  
referenced by DML statement, 11-16  
referenced by FOR loop, 11-17  
referenced by SELECT statement, 11-16

column  
generated, 24-4  
multiple foreign key constraints on, 10-11  
virtual, 24-4  
when to use default value for, 10-6

commit redo management, 6-5

COMPATIBLE server initialization parameter, 2-18

compilation parameter, 11-3

composite PL/SQL data type, 11-9

concurrency  
serializable transaction for, 6-25  
under explicit locking, 6-17

conditional compilation, 5-2

connection class, 2-28

connection pool, 17-16

constraint, 10-1  
altering, 10-21  
CHECK  
*See* CHECK constraint

compared to trigger, 10-3

crossedition trigger and  
collisions, 24-28  
dropping, 24-32

deferring checking of, 10-11

disabling  
effect of, 10-18  
existing, 10-20  
new, 10-19  
reasons for, 10-18

dropping, 10-23

editioning view and, 24-22

enabling  
effect of, 10-18  
existing, 10-19  
new, 10-18

exception to, 10-21

FOREIGN KEY  
*See* FOREIGN KEY constraint

minimizing overhead of, 10-13

naming, 10-17

NOT NULL  
*See* NOT NULL constraint

on view, 10-1

PRIMARY KEY  
*See* PRIMARY KEY constraint

privileges needed for defining, 10-17

referential integrity  
*See* FOREIGN KEY constraint

renaming, 10-22

UNIQUE  
*See* UNIQUE constraint

viewing definition of, 10-25

violation of, 10-21

Continuous Query Notification (CQN), 2-24, 15-1  
example, 15-34

converting data types  
*See* data type conversion

copy-on-change strategy, 24-10

coupling, 19-4

CQ\_NOTIFICATION\$\_DESCRIPTOR object, 15-45

CQ\_NOTIFICATION\$ QUERY object, 15-46

CQ\_NOTIFICATION\$ REG\_INFO object, 15-19

CQ\_NOTIFICATION\$ ROW object, 15-47

CQ\_NOTIFICATION\$ TABLE object, 15-46

CQN (Continuous Query Notification), 15-1

CREATE OR REPLACE optimization, 23-5  
actualization and, 24-10

crossedition trigger, 24-23

creating, 24-27  
displaying information about, 24-34  
dropping, 24-32  
execution of, 24-27  
forward, 24-23  
interaction with editions, 24-24  
read-only editioning view and, 24-21  
read-write editioning view and, 24-21  
reverse, 24-24  
scope of, 24-2  
sharing child cursor and, 24-34  
crossedition trigger SQL  
    forward, 24-25  
    reverse, 24-25  
cross-session PL/SQL function result cache, 11-15  
current date and time, displaying, 7-13  
current edition, 24-15  
cursor, 11-10  
    crossedition trigger and, 24-34  
    explicit, 11-10  
    implicit, 11-10  
    Oracle XA application and, 19-11  
    schema object dependency and, 23-19  
    session, 11-10  
cursor variable, 11-9  
    advantages of, 11-11  
    disadvantages of, 11-11

## D

---

data cartridge, 1-2  
data definition language statement  
    *See* DDL statement  
data integrity, 10-1  
    data type and, 7-2  
    *See also* constraint  
data modeling, 2-1  
data type  
    ANSI, 7-21  
    BOOLEAN, 11-8  
    DB2, 7-21  
    dynamic, 7-20  
    external, 7-1  
    for character data, 7-5  
    for datetime data, 7-12  
    for geographic data, 7-18  
    for large amount of data, 7-18  
    for multimedia data, 7-18  
    for numeric data, 7-6  
    for spatial data, 7-18  
    for XML data, 7-19  
    importance of correct, 7-1  
    PLS\_INTEGER, 11-8  
    PL/SQL, 11-7  
    REF CURSOR, 11-9  
    SQL, 7-1  
    SQL/DS, 7-21  
data type class, 23-16  
data type conversion  
    of ANSI and IBM data types, 7-21

    of datetime data types, 7-16  
    of native floating-point data types, 7-11  
data type family  
    PL/SQL, 11-8  
    SQL, 7-24  
database hardening, 16-17  
database logins, automated, 3-2  
Database Resident Connection Pool (DRCP), 2-24  
date  
    default format for, 7-13  
    displaying, 7-14  
        current, 7-13  
    inserting, 7-14  
    *See also* datetime data types  
datetime data type class, 23-17  
datetime data types, 7-12  
    arithmetic operations with, 7-16  
    conversion functions for, 7-16  
    importing, exporting, and comparing, 7-17  
day, default value for, 7-15  
DB2 data type, 7-21  
DBMS\_APPLICATION\_INFO package, 2-5  
DBMS\_DEBUG package, 11-35  
DBMS\_FLASHBACK package, 16-13  
DBMS\_FLASHBACK\_ARCHIVE procedures, 16-17  
DBMS\_FLASHBACK\_TRANSACTION\_BACKOUT  
    procedure, 16-14  
DBMS\_HPROF package, 13-2  
DBMS\_LOCK package, 6-24  
DBMS\_OUTPUT package, 11-33  
DBMS\_PARALLEL\_EXECUTE package, 24-30  
DBMS\_SQL.RETURN\_RESULT procedure, 11-14  
DBMS\_STATS package, 16-26  
DBMS\_TYPES package, 7-20  
DBMS\_XA package, 19-16  
DDL statement  
    blocking, 6-31  
    Flashback Data Archive and, 16-21  
    for creating package, 11-6  
    for creating subprogram, 11-6  
    ineffective, 23-5  
    nonblocking, 6-31  
        in autonomous transaction, 6-32  
    Oracle XA and, 19-26  
    processing, 6-2  
        that generates notification, 15-5  
DDL\_LOCK\_TIMEOUT parameter, 6-32  
deadlock, undetected, 6-24  
debugging  
    compiling PL/SQL unit for, 11-34  
    external subprogram, 18-39  
    low-level code, 11-35  
    subprogram, 11-33  
        wrap utility and, 11-34  
decimal floating-point number, 7-8  
default column value, 10-6  
deferring constraint checks, 10-11  
definer's rights, 3-4  
denormal floating-point number, 7-9  
dependency mode, 23-12

dependent object  
    *See* schema object dependency

dependent transaction, 16-14  
descendent edition, 24-10  
design

    physical, 2-2, 2-3

DETERMINISTIC function  
    function-based index and, 9-4  
    RPC signature and, 23-14

dirty read, 6-26

disabling constraint

    effect of, 10-18

    existing, 10-20

    new, 10-19

    reasons for, 10-18

distributed database

    FOREIGN KEY constraint and, 10-13  
    remote dependency management and, 23-11

distributed transaction, 19-3

    remote subprogram and, 11-25

DLL (dynamic link library), 18-3

DML statement

    bulk binding for, 11-16

    that references collection, 11-16

DML\_LOCKS initialization parameter, 6-13

domain index, 9-2

double-precision IEEE 754 format, 7-8

drivers, Oracle JDBC, 17-6

DTP (X/Open Distributed Transaction

    architecture), 19-2

dynamic link library (DLL), 18-3

dynamic registration, 19-4

dynamic session state consistency, 26-18

dynamic SQL, 11-18

    implicit query results and, 11-14

    RESTRICT\_REFERENCES pragma and, 11-32

dynamically typed data, 7-20

## E

EBR (edition-based redefinition), 24-1

edition, 24-2

    ancestor, 24-10

    creating, 24-10

    crossedition triggers and, 24-24

    current, 24-15

    descendent, 24-10

    displaying information about, 24-33

    dropping, 24-19

    enabling for user and types, 24-6

    evaluation

*See* evaluation edition

    leaf, 24-10

    making available

        to all users, 24-14

        to some users, 24-14

    ora\$base, 24-2, 24-10

    retiring, 24-18

    root, 24-10

    scope of, 24-2

session, 24-15

unusable

*See* unusable edition

visibility of trigger in, 24-24

EDITIONABLE property, 24-8

editionable schema object type, 24-5

edition-based redefinition (EBR), 24-1

editioned object, 24-2

    creating or replacing, 24-9

    name resolution and, 24-3

editioning view, 24-20

    auditing policy and, 24-35

    changing base table of, 24-22

    changing writability of, 24-22

    covering table with, 24-35

    creating, 24-21

    displaying information about, 24-33

    partition-extended name for, 24-22

    preparing application for, 24-35

    read-only, 24-21

    read-write, 24-21

    replacing, 24-22

    scope of, 24-2

    SQL optimizer hint and, 24-23

Electronic Product Code (EPC), 22-22

embedded PL/SQL gateway, 14-3

    how to use, 14-4

enabling constraint

    effect of, 10-18

    existing, 10-19

    new, 10-18

enabling editions, 24-6

encoding scheme, adding, 22-13

environment, programming, 17-1

EPC (Electronic Product Code), 22-22

evaluation edition, 24-3

    dropping edition and, 24-20

    for materialized view, 24-4

    for virtual column, 24-4

    retiring edition and, 24-19

exception

    IEEE 754 standard

        not raised, 7-10

        raised during conversion, 7-11

    in multilanguage program, 18-30

    to constraint, 10-21

exception handling, 5-2

    for storage allocation error, 6-39

EXCLUSIVE MODE option of LOCK TABLE

    statement, 6-16

EXECUTE privilege, 11-6

execution plan, 2-6

    data type and, 7-2

EXPLAIN PLAN statement, 2-6

explicit cursor, 11-10

EXPR data type, 7-24

expression

    index built on

*See* function-based index

    regular

*See* regular expression  
extensibility, 1-2  
external data type, 7-1  
external large object (BFILE) data type, 7-18  
external subprogram, 18-3  
    call specification for, 18-3  
    debugging, 18-39  
    loading, 18-4  
    managing for applications, 3-6  
    publishing, 18-9  
external transaction manager, 19-3

## F

---

family of data types  
    PL/SQL, 11-8  
    SQL, 7-24  
FAN event, load balancing advisory, 2-37  
Fast Application Notification (FAN), 2-35  
fine-grained access control, 3-3  
fine-grained auditing (FGA) policy, editioning view  
    and, 24-35  
fine-grained invalidation, 23-5  
firing order of triggers, 24-26  
FIXED\_DATE initialization parameter, 7-14  
fixed-point data type, 7-6  
Flashback Data Archive, 16-16  
Flashback Transaction, 16-14  
FLASHBACK\_TRANSACTION\_QUERY  
    view, 16-10  
FLOAT data type, 7-7  
floating-point data type, 7-6  
    range and precision of, 7-8  
    *See also* native floating-point data type  
floating-point number  
    binary, 7-8  
    components of, 7-7  
    decimal, 7-8  
    denormal, 7-9  
    format of, 7-8  
    rounding, 7-8  
    subnormal, 7-9  
FOR loop  
    bulk binding for, 11-17  
    that references collection, 11-17  
FORCE option of ALTER USER statement, 24-7  
FOREIGN KEY constraint, 10-8  
    distributed databases and, 10-13  
    dropping, 10-23  
    editioned view and, 24-9  
    enabling, 10-24  
    Flashback Transaction and, 16-15  
    indexing, 10-13  
    multiple, 10-11  
    naming, 10-17  
    NOT NULL constraint on, 10-10, 10-11  
    NULL value and, 10-10  
    privileges needed to create, 10-24  
    referential integrity enforced by, 10-24  
    UNIQUE constraint on, 10-10, 10-11

    without other constraints, 10-10  
foreign key dependency, 16-15  
forward compatibility, 1-2  
forward crossedition trigger, 24-23  
forward crossedition trigger SQL, 24-25  
function  
    aggregate, 11-1  
    analytic, 1-7  
    built-in  
        *See* SQL function  
DETERMINISTIC  
    function-based index and, 9-4  
    RPC signature and, 23-14  
invoking from SQL statement, 11-26  
MGD\_ID ADT, 22-10  
OCI or OCCI, 17-20  
PARALLEL\_ENABLE, RPC signature and, 23-14  
PL/SQL, invoked by SQL statement, 11-25  
purity of, 11-28  
    RPC signature and, 23-14  
result-cached, 11-15  
returning large amount of data from, 11-14  
SQL  
    *See* SQL function  
SQL analytic, 1-7  
    *See also* subprogram  
function result cache, 11-15  
function-based index, 9-2  
    examples of, 9-5  
    optimizer and, 9-2, 9-3

## G

---

generated column, 24-4  
Geographic Information System (GIS) data, 7-18  
GET\_LTXID\_OUTCOME procedure, 6-9  
global transaction, 19-3  
global variable, in C external subprogram, 18-40  
greedy operator in regular expression, 8-5  
group commit, 6-6

## H

---

hierarchical profiler, 13-1  
historical data, importing and exporting, 16-17  
host language, 17-15  
host program, 17-15  
hot rollover, 24-1

## I

---

IA-32 and IA-64 instruction set architecture, 7-11  
IBM CICS, 19-3  
IBM Transarc Encina, 19-3  
Identity Code Package, 22-1  
IEEE 754 standard, 7-7  
exception  
    not raised, 7-10  
    raised during conversion, 7-11  
special values supported by, 7-9  
    *See also* native floating-point data type

IGNORE\_ROW\_ON\_DUPKEY\_INDEX hint, 24-28

IMMEDIATE commit redo option, 6-6  
implementing database application, 2-3  
implicit cursor, 11-10  
independent transaction  
    *See* autonomous transaction

index, 9-1  
    domain, 9-2  
    edition-based redefinition and, 24-22  
    function-based  
        *See* function-based index  
    on MGD\_ID column, 22-10

infinity, 7-9  
in-flight transaction, 6-7  
INHERIT ANY PRIVILEGES system privilege, 3-5  
INHERIT PRIVILEGES privilege, 3-5  
inherited object, 24-10  
initialization parameter, 11-3  
    DML\_LOCKS, 6-13  
    FIXED\_DATE, 7-14  
    NLS\_DATE\_FORMAT, 7-13

instrumentation, 2-9  
integer data type class, 23-16  
integrity constraint  
    *See* constraint

integrity of data  
    *See* data integrity

interface  
    between PL/SQL and C, 18-10  
    between PL/SQL and Java, 18-10  
    program, 17-3  
    TX, 19-4  
    user, 17-3  
        stateful or stateless, 17-3

*See also* Oracle C++ Call Interface  
*See also* Oracle Call Interface

invalidation  
    cascading, 23-5  
    coarse-grained, 23-5  
    fine-grained, 23-5  
    of dependent object, 23-4  
    of package, 11-35

invoker's rights, 3-4  
    affect on invoker's privileges, 3-5  
    Java stored procedures, 3-5

invoking subprogram, 11-19  
    from subprogram, 11-22  
    from trigger, 11-22  
    interactively from Oracle Database tools, 11-20  
    through embedded PL/SQL gateway, 14-17

isolation level

*See* transaction isolation level

## J

---

Java class method  
    calling, 18-29  
    interface between PL/SQL and, 18-10  
    publishing, 18-11  
*See also* external subprogram

Java Database Connectivity

*See* Oracle JDBC

Java language  
    compared to PL/SQL, 17-13  
    Oracle Database support for, 17-4

Java stored procedures, 3-5

Java Virtual Machine

*See* Oracle JVM

JDBC

*See* Oracle JDBC

JVM

*See* Oracle JVM

## K

---

key  
    foreign  
        *See* FOREIGN KEY constraint  
    primary  
        *See* PRIMARY KEY constraint  
    unique  
        *See* UNIQUE constraint

## L

---

Large Object (LOB), 7-18  
leaf edition, 24-10  
LGWR (log writer process), 6-5  
lightweight queue, 20-3  
live operation, 24-1  
load balancing advisory FAN event, 2-37  
LOB  
    *See* Large Object (LOB)  
LOCK TABLE statement, 6-13  
    SELECT FOR UPDATE statement with, 6-17  
locking row explicitly, 6-16  
locking table  
    explicitly, 6-12  
    implicitly, 6-16  
log writer process (LGWR), 6-5  
logical design, 2-2  
logical transaction identifier (LTXID), 6-7  
logical value, 11-8  
LONG and LONG RAW data types, 7-19  
LONG data type, 7-5  
loose coupling, 19-4  
low-level code, debugging, 11-35  
LTXID (logical transaction identifier), 6-7

## M

---

main transaction, 6-32  
maintaining database and application, 2-4  
managing default rights, 3-5  
materialized view, 1-8  
    that depends on editioned object, 24-4  
maximum availability of table, 24-21  
memory advisor, 2-8  
metacharacter in regular expression, 8-1  
metadata for SQL operator or function, 7-23  
metrics, 2-4

MGD\_ID ADT, 22-1  
 MGD\_ID database ADT function, 22-10  
 mod\_plsql module, 14-2  
 mode  
     dependency, 23-12  
     lock, 6-13  
     serialized  
         *See* serializable transaction  
 MODIFY CONSTRAINT clause of ALTER TABLE statement, 10-21  
 modifying  
     *See* altering  
 monitoring database performance, 2-7  
 multilanguage program, 18-1  
     error or exception in, 18-30  
 multiline mode, 8-3  
 multilingual data, 8-7  
 multimedia data, 7-18

## N

---

name resolution, 23-10  
     editions and, 24-3  
 NaN (not a number), 7-9  
 national character large object (NCLOB) data type, 7-18  
 native execution, 11-19  
 native floating-point data type, 7-7  
     arithmetic operation with, 7-10  
     binary format for, 7-8  
     conversion functions for, 7-11  
     in client interfaces, 7-11  
     special values for, 7-9  
 NCHAR data type, 7-5  
 NCLOB data type, 7-5, 7-18  
 negative infinity, 7-9  
 negative zero, 7-9  
 nested subprogram, 11-2  
 NLS\_DATE\_FORMAT initialization parameter, 7-13  
 NO\_RESULT\_CACHE hint, 2-14  
 nonblocking DDL statement, 6-31  
     in autonomous transaction, 6-32  
 NONEDITABLE property, 24-8  
 noneditionable schema object type, 24-5  
 noneditioned object, 24-2  
     creating or replacing, 24-9  
     name resolution and, 24-3  
     that can depend on editioned object, 24-3  
         dropping edition and, 24-19  
         FORCE and, 24-7  
 nongreedy operator in regular expression, 8-8  
 nonpersistent queue, 20-3  
 noreplay keyword (killing or disconnecting session), 26-16  
 normalized significand, 7-9  
 NOT NULL constraint  
     compared to CHECK constraint, 10-15  
     naming, 10-17  
     on FOREIGN KEY constraint, 10-10, 10-11  
     when to use, 10-4

NOWAIT commit redo option, 6-6  
 NOWAIT option of LOCK TABLE statement, 6-14  
 NULL value  
     FOREIGN KEY constraint and, 10-10  
     function-based index and, 9-3  
     indexing and, 10-5  
 NUMBER data type, 7-6  
 number data type class, 23-16  
 numeric data types, 7-6  
 NVARCHAR2 data type, 7-5

## O

---

object  
     actual, 24-10  
     dependent  
         *See* schema object dependency  
     editioned  
         *See* editioned object  
     inherited, 24-10  
     large  
         *See* Large Object (LOB)  
     noneditioned  
         *See* noneditioned object  
     potentially editioned, 24-2  
         with noneditioned dependents, 24-7  
     referenced  
         *See* schema object dependency  
     size limit for PL/SQL stored, 11-7  
 object change notification, 15-2  
 OCCI  
     *See* Oracle C++ Call Interface  
 OCI  
     *See* Oracle Call Interface  
 OCI\_ATTR\_CHDES\_DBNAME, 15-33  
 OCI\_ATTR\_CHDES\_NFTYPE, 15-33  
 OCI\_ATTR\_CHDES\_QUERIES, 15-33  
 OCI\_ATTR\_CHDES\_ROW\_OPFLAGS, 15-34  
 OCI\_ATTR\_CHDES\_ROW\_ROWID, 15-34  
 OCI\_ATTR\_CHDES\_TABLE\_CHANGES, 15-33  
 OCI\_ATTR\_CHDES\_TABLE\_NAME, 15-34  
 OCI\_ATTR\_CHDES\_TABLE\_OPFLAGS, 15-34  
 OCI\_ATTR\_CHDES\_TABLE\_ROW\_CHANGES, 15-34  
 OCI\_ATTR\_CHDES\_XID, 15-33  
 OCI\_ATTR\_CHNF\_CHANGELAG, 15-32  
 OCI\_ATTR\_CHNF\_OPERATIONS, 15-32  
 OCI\_ATTR\_CHNF\_ROWIDS, 15-32  
 OCI\_ATTR\_CHNF\_TABLENAMES, 15-32  
 OCI\_ATTR\_CQ\_QUERYID, 15-32  
 OCI\_ATTR\_CQDES\_OPERATION, 15-33  
 OCI\_ATTR\_CQDES\_QUERYID, 15-34  
 OCI\_ATTR\_CQDES\_TABLE\_CHANGES, 15-34  
 OCI\_ATTR\_SESSION\_STATE attribute, 2-33  
 OCI\_ATTR\_SUBSCR\_CALLBACK, 15-31  
 OCI\_ATTR\_SUBSCR\_CQ\_CHNF\_QOSFLAGS, 15-31  
 OCI\_ATTR\_SUBSCR\_NAMESPACE, 15-31  
 OCI\_ATTR\_SUBSCR\_QOSFLAGS, 15-31  
 OCI\_ATTR\_SUBSCR\_TIMEOUT, 15-31

OCI\_DTYPE\_CQDES, 15-33  
 OCI\_SESSGET\_PURITY\_NEW attribute, 2-28  
 OCI\_SESSGET\_PURITY\_SELF attribute, 2-28  
 OCI\_SESSION\_STATELESS attribute, 2-33  
 OCI\_SUBSCR\_QOS\_PURGE\_ON\_NTFN, 15-31  
 OCI\_SUBSCR\_QOS\_RELIABLE, 15-31  
 OCIAnyData and OCIAnyDataSet interfaces, 7-20  
 ODP.NET, 17-22  
 online application upgrade  
*See* edition-based redefinition (EBR)  
 operator  
   in regular expression, 8-4  
   greedy, 8-5  
   nongreedy, 8-8  
   metadata for, 7-23  
 optimizer  
   editioning view and, 24-23  
   function-based index and, 9-2, 9-3  
   RPC signature and, 23-14  
 ora\$base edition, 24-2, 24-10  
 Oracle Advanced Queuing (AQ), 20-2  
 Oracle C++ Call Interface, 17-18  
   building application with, 17-21  
   kinds of functions in, 17-20  
   native floating-point data types in, 7-12  
   procedural and nonprocedural elements of, 17-20  
 Oracle Call Interface, 17-18  
   building application with, 17-21  
   commit redo action in, 6-6  
   compared to precompiler, 17-21  
   kinds of functions in, 17-20  
   native floating-point data types in, 7-12  
   procedural and nonprocedural elements of, 17-20  
   with Oracle XA, 19-12  
 Oracle Data Provider for .NET, 17-22  
 Oracle Data Redaction, 3-3  
 Oracle data type  
*See* SQL data type  
 Oracle Database Tuning Pack, 2-8  
 Oracle Database Vault, 3-3  
 Oracle Extensibility Architecture framework,  
   user-defined aggregate functions and, 1-3  
 Oracle Extensibility Architecture, data cartridges  
   and, 1-2  
 Oracle Flashback Query, 16-6  
 Oracle Flashback Technology, 16-1  
   application development features, 16-2  
   configuring database for, 16-3  
   database administration features, 16-3  
   performance guidelines for, 16-26  
 Oracle Flashback Transaction Query, 16-10  
 Oracle Flashback Version Query, 16-8  
 Oracle JDBC, 17-5  
   compared to Oracle SQLJ, 17-10  
   native floating-point data types in, 7-12  
   sample program  
     2.0, 17-7  
     pre-2.0, 17-8  
 Oracle JDeveloper, Oracle SQLJ and, 17-10  
 Oracle JPublisher, 17-11  
 Oracle JVM, 17-4  
 Oracle Label Security, 3-3  
 Oracle Lock Management services, 6-24  
 Oracle Multimedia, 7-18  
 Oracle Real Application Clusters (Oracle RAC)  
   client result cache and, 2-18  
   DRCP and, 2-34  
   load balancing advisory FAN events and, 2-37  
   Oracle XA and, 19-22  
   runtime connection load balancing and, 2-35  
 Oracle SQLJ, 17-9  
   compared to Oracle JDBC, 17-10  
   Oracle JDeveloper and, 17-10  
 Oracle Text, 7-19  
 Oracle Total Recall, 16-16  
 Oracle Tuxedo, 19-3  
 Oracle Virtual Private Database (VPD), 3-3  
   editioning view and, 24-35  
 Oracle XA  
   Oracle RAC and, 19-22  
   subprograms, 19-5  
   when to use, 19-1  
 out-of-space error, 6-39  
 overloaded subprogram, 11-2

## P

---

package  
   creating, 11-7  
   dropping, 11-19  
   invalidation of, 11-35  
   session state and, 23-8  
   size limit for, 11-7  
   synonym for, 11-24  
 package body, 11-2  
 package invalidation and, 11-35  
 package specification, 11-2  
 package subprogram, 11-2  
 PARALLEL\_ENABLE function  
   RPC signature and, 23-14  
 parallelized SQL statement, 11-29  
 parameter  
   compilation  
*See* compilation parameter  
   initialization, 11-3  
 partition-extended editioning view name, 24-22  
 partitioning, 1-9  
 performance, 2-1  
 performance goals, 2-4  
 performance testing, 2-9  
 performance, data type and, 7-2  
 persistent LOB instance, 7-19  
 persistent queue, 20-3  
 phantom read, 6-26  
 physical design, 2-3  
 PLS\_INTEGER data type, 11-8  
 PL/Scope tool, 12-1  
 plshprof utility, 13-13  
 PL/SQL block, 11-1  
 PL/SQL data type, 11-7

PL/SQL function result cache, 11-15  
PL/SQL gateway, 14-2  
PL/SQL hierarchical profiler, 13-1  
PL/SQL language, 17-4  
    compared to Java, 17-13  
PL/SQL object  
    *See PL/SQL unit*  
PL/SQL optimize level, 11-4  
PL/SQL optimizer level, 5-1  
PL/SQL unit, 11-3  
    compiling for debugging, 11-34  
    CREATE OR REPLACE and, 23-5  
PL/SQL Web Toolkit, 14-3  
pool, connection, 17-16  
positive infinity, 7-9  
positive zero, 7-9  
POSIX standard for regular expressions  
    operators defined in, 8-4  
    Oracle SQL and, 8-4  
    Oracle SQL multilingual extensions to, 8-7  
    Oracle SQL PERL-influenced extensions to, 8-7  
potentially editioned object, 24-2  
    with noneditioned dependents, 24-7  
precompiler, 17-15  
    compared to Oracle Call Interface, 17-21  
    Oracle XA and, 19-11  
PRIMARY KEY constraint, 10-7  
    dropping, 10-23  
    Flashback Transaction and, 16-15  
    naming, 10-17  
primary key dependency, 16-15  
privileges, 3-1  
    for debugging subprogram, 11-34  
    for defining constraint, 10-17  
    for Oracle Flashback Technology, 16-5  
    for running subprogram, 11-20  
    granting secure application roles, 3-1  
    grouped into roles, 3-1  
    INHERIT ANY PRIVILEGES system  
        privilege, 3-5  
    INHERIT PRIVILEGES privilege, 3-5  
    revoked, object dependency and, 23-8  
Pro\*C/C++ precompiler, 17-15  
    native floating-point data types in, 7-12  
Pro\*COBOL precompiler, 17-17  
procedure  
    *See also* subprogram  
product code, 22-22  
profiler, 13-1  
program interface, 17-3  
programming environment, 17-1  
public information, required, 19-5  
publish-subscribe model, 20-1  
purity of function, 11-28  
    RPC signature and, 23-14

## Q

quality-of-service flag, 15-21  
query

registering for Continuous Query  
    Notification, 15-10  
returning results to client, 11-10  
    implicitly, 11-14  
Query Result Change Notification (QRCN), 15-2  
query rewrite, 1-9  
queue, 20-3

## R

Radio Frequency Identification (RFID)  
    technology, 22-21  
RAW data type, 7-19  
raw data type class, 23-16  
READ COMMITTED transaction isolation level  
    compared to SERIALIZABLE, 6-31  
    in Oracle Database, 6-26  
    transaction interactions with, 6-26  
read consistency, 6-12  
    statement-level, 6-11  
    transaction-level, 6-12  
        locking tables explicitly for, 6-12  
        read-only transaction for, 6-12  
read lock, 6-28  
READ UNCOMMITTED transaction isolation level  
    in Oracle Database, 6-26  
    transaction interactions with, 6-26  
read-only editioning view, 24-21  
read-only transaction, 6-12  
read-write editioning view, 24-21  
record, 11-10  
redefinition, edition-based (EBR), 24-1  
redo information for transaction, 6-5  
redo management, 6-5  
REF CURSOR data type, 11-9  
referenced object  
    *See* schema object dependency  
referential integrity  
    serializable transactions and, 6-28  
    trigger for enforcing, 6-29  
referential integrity constraint  
    *See* FOREIGN KEY constraint  
REGEXP\_COUNT function, 8-2  
REGEXP\_INSTR function, 8-3  
REGEXP\_LIKE condition, 8-2  
REGEXP\_REPLACE function, 8-3  
    back reference operator in, 8-6  
REGEXP\_SUBSTR function, 8-3  
registration  
    dynamic, 19-4  
    for Continuous Query Notification, 15-10  
    in publish-subscribe model, 20-3  
    static, 19-4  
regular expression, 8-1  
    in Oracle SQL, 8-2  
    in SQL statement, 8-9  
metacharacter in, 8-1  
POSIX standard and  
    *See* POSIX standard for regular expressions  
Unicode and, 8-4

remote dependency management, 23-11  
remote procedure call dependency management, 23-12  
remote subprogram, 11-23  
repeatable read, 6-12  
    locking tables explicitly for, 6-12  
    read-only transaction for, 6-12  
REPEATABLE READ transaction isolation level  
    in Oracle Database, 6-27  
    transaction interactions with, 6-26  
required public information, 19-5  
resource manager (RM), 19-3  
RESTRICT\_REFERENCES pragma  
    for backward compatibility, 11-30  
    static and dynamic SQL and, 11-32  
result cache, 11-15  
RESULT\_CACHE hint, 2-14  
RESULT\_CACHE\_MODE session parameter, 2-15  
resumable storage allocation, 6-39  
RETENTION GUARANTEE clause for undo  
    tablespace, 16-4  
RETENTION option of ALTER TABLE statement, 16-5  
RETURN\_RESULT procedure, 11-14  
reverse crossedition trigger, 24-24  
reverse crossedition trigger SQL, 24-25  
RFID (Radio Frequency Identification)  
    technology, 22-21  
RM (resource manager), 19-3  
roles, 3-1  
root edition, 24-10  
rounding floating-point numbers, 7-8  
row  
    address of (rowid), 7-22  
    locking explicitly, 6-16  
ROW EXCLUSIVE MODE option of LOCK TABLE statement, 6-14  
ROW SHARE MODE option of LOCK TABLE statement, 6-14  
rowid, 7-22  
ROWID data type  
    ROWID pseudocolumn and, 7-22  
ROWID pseudocolumn, 7-22  
    CQN and, 15-12  
    *See also* rowid  
RPC dependency management, 23-12  
    RPC signature and, 23-13  
RR datetime format element, 7-13  
rule on queue, 20-3  
rules engine, 20-4  
runtime connection load balancing, 2-35  
runtime error  
    *See* exception  
invalidation and, 23-4  
on nonexistence of other objects, 23-11  
revoked privileges and, 23-8  
shared pool and, 23-19  
schema object type  
    editionable, 24-5  
    enabling for editions, 24-6  
    noneditionable, 24-5  
searchable text, 7-19  
secure application roles, 3-1  
security  
    auditing, 3-6  
    external procedures for applications, 3-6  
    invoker's rights and definer's rights, 3-4  
    Java stored procedures default rights, 3-5  
    logon triggers and, 3-2  
    Oracle Data Redaction, 3-3  
    Oracle Database Vault, 3-3  
    Oracle Label Security, 3-3  
    Oracle Virtual Private Database, 3-3  
    privilege use, 3-1  
    privileges of invoking user, 3-5  
    role use, 3-1  
    secure application roles, 3-1  
SELECT FOR UPDATE statement, 6-16  
LOCK TABLE statement with, 6-17  
referential integrity and  
    inside trigger, 6-29  
    outside trigger, 6-29  
SELECT statement  
    bulk binding for, 11-16  
    referencing collection with, 11-16  
    with AS OF clause, 16-6  
    with FOR UPDATE clause  
        *See* SELECT FOR UPDATE statement  
    with VERSIONS BETWEEN clause, 16-8  
semi-available table, 24-21  
serendipitous change, 24-30  
    data transformation collisions and, 24-28  
    identifying, 24-28  
serializable transaction  
    for concurrency control, 6-25  
    interaction with, 6-27  
    referential integrity and, 6-28  
SERIALIZABLE transaction isolation level  
    compared to READ COMMITTED, 6-31  
    in Oracle Database, 6-27  
    transaction interactions with, 6-26  
        *See also* serializable transaction  
server-side programming, 17-2  
service routine, C external subprogram and, 18-30  
session cursor, 11-10  
session edition, 24-15  
session purity, 2-28  
session state, 23-8  
session state consistency, 26-18  
session variable, 11-21  
SET CONSTRAINTS statement, 10-11  
SET TRANSACTION statement with READ ONLY option, 6-12

## S

---

scalability, 2-1  
scalar PL/SQL data type, 11-8  
schema object dependency, 23-1  
    in distributed database, 23-11

SHARE MODE option of LOCK TABLE  
 statement, 6-14

SHARE ROW EXCLUSIVE MODE option of LOCK TABLE statement, 6-15

shared SQL area, 6-3

side effects of subprogram, 11-28

signature checking, 23-11

single-precision IEEE 754 format, 7-8

spatial data, 7-18

specification, package  
*See* package specification

SQL Access Advisor, 2-8

SQL advisor, 2-8

SQL analytic function, 1-7

SQL area, shared, 6-3

SQL data type, 7-1

SQL function  
 analytic, 1-7  
 display type of, 7-24  
 for data type conversion, 7-16  
 metadata for, 7-23

SQL optimizer hint and editioning view, 24-23

SQL statement  
 application, 24-25  
 crossedition trigger  
 forward, 24-25  
 reverse, 24-25  
 invoking PL/SQL function from, 11-26  
 parallelized, 11-29  
 processing  
*See* DDL statement, 6-2  
 stages of, 6-1  
 system management statement, 6-2

SQL Trace facility (SQL\_TRACE), 2-6

SQL Tuning Advisor, 2-8

SQL, dynamic  
*See* dynamic SQL

SQL/DS data type, 7-21

SQLJ  
*See* Oracle SQLJ

standalone subprogram, 11-2

state  
 session, 23-8  
 user interface and, 17-3  
 web application and, 14-25

stateful session, 2-33

stateless session, 2-34

statement  
*See* SQL statement

statement caching, 2-22

statement-level read consistency, 6-11, 6-12

static registration, 19-4

static session state consistency, 26-18

static SQL, RESTRICT\_REFERENCES pragma  
 and, 11-32

static variable, in C external subprogram, 18-40

statistics  
 for application, 13-1  
 for identifier, 12-1

storage allocation error, 6-39

storage requirements, decreasing, 7-2

stored standalone subprogram, dropping, 11-19

stored subprogram, 11-2

subnormal floating-point number, 7-9

subprogram, 11-2  
 creating, 11-7  
 external  
*See* external subprogram

invoking  
*See* invoking subprogram

Oracle XA, 19-5

overloaded, 11-2

privileges needed to debug, 11-34

privileges needed to run, 11-20

remote, 11-23

size limit for, 11-7

synonym for, 11-24

subscriber, 20-3

subscription services, 20-4

subtype, 11-8  
 user-defined, 11-9

synonym  
 CREATE OR REPLACE and, 23-5  
 for package, 11-24  
 for subprogram, 11-24

SYSDATE function, 7-13

system management statement, 6-2

## T

---

table  
 controlling user access to, 3-3

locking  
 choosing strategy for, 6-13  
 explicitly, 6-12  
 implicitly, 6-16  
 with maximum availability, 24-21  
 with semi-availability, 24-21

table annotation, 2-14

table result cache mode  
 annotation, 2-15  
 effective  
 determining, 2-15  
 displaying, 2-16

Tag Data Translation Markup Language  
 Schema, 22-4

Temporal Validity Support, 1-10

temporary LOB instance, 7-19

thin client configuration, 17-2

thread-safe application, 19-15

three-tier architecture, 17-2

tight coupling, 19-4

time  
 default value for, 7-15  
 displaying, 7-15  
 current, 7-13  
 inserting, 7-15  
*See also* datetime data types

time-stamp checking, 23-11

time-stamp dependency mode, 23-12

TM (transaction manager), 19-3  
TPM (transaction processing monitor), 19-3  
tracing tools, 11-1  
transaction  
    autonomous, 6-32  
    trigger as, 6-40  
    choosing isolation level for, 6-30  
    dependent, 16-14  
    determining outcome after outage, 6-7  
    distributed, 19-3  
        remote subprogram and, 11-25  
    ensuring idempotence of, 6-7  
    global, 19-3  
    grouping operations into, 6-4  
    improving performance of, 6-4  
    in-flight, 6-7  
    main, 6-32  
    read-only, 6-12  
    redo entry for, 6-5  
    serializable  
        *See* serializable transaction  
    that invokes remote subprogram, 11-25  
Transaction Guard, 6-7, 25-1  
transaction history table, 25-6  
transaction interaction  
    kinds of, 6-26  
    serializable transaction and, 6-27  
    transaction isolation level and, 6-26  
transaction isolation level  
    choosing, 6-30  
    setting, 6-27  
    transaction interaction and, 6-26  
transaction manager (TM), 19-3  
transaction processing monitor (TPM), 19-3  
transaction set consistency, 6-30  
transaction-level read consistency, 6-12  
    locking tables explicitly for, 6-12  
    read-only transaction for, 6-12  
transform, 24-24  
    applying, 24-30  
trigger, 11-2  
    AFTER SUSPEND, 6-40  
    as autonomous transaction, 6-40  
    automating database login with, 3-2  
    compared to constraint, 10-3  
    crossedition  
        *See* crossedition trigger  
    enforcing referential integrity with, 6-29  
in edition  
    firing order of, 24-26  
    visibility of, 24-24  
    what kind can fire, 24-24  
invoking subprogram from, 11-22  
size limit for, 11-7  
TRUST assertion (deprecated), 11-32  
TRUST keyword in RESTRICT\_REFERENCES  
    pragma, 11-32  
two-phase commit protocol, 19-4  
two-tier architecture, 17-2  
TX interface, 19-4

## U

---

undetected deadlock, 6-24  
undo data, 16-1  
UNDO\_RETENTION parameter, 6-12  
Unicode  
    data types for, 7-5  
    regular expressions and, 8-4  
unified auditing, 3-6  
UNIQUE constraint  
    crossedition trigger and, 24-28  
    dropping, 10-23  
    naming, 10-17  
    on FOREIGN KEY constraint, 10-10, 10-11  
    when to use, 10-7  
unrepeatable read, 6-26  
unable edition  
    dropping edition and, 24-20  
    for materialized view, 24-4  
    retiring edition and, 24-19  
upgrading applications online  
    *See* edition-based redefinition (EBR)  
UROWID data type, 7-22  
user access  
    *See* security  
user interface, 17-3  
    stateful and stateless, 17-3  
user lock, 6-24  
user-defined subtype, 11-9  
UTLLOCKT.SQL script, 6-25

## V

---

VARCHAR data type, 7-6  
VARCHAR data type class, 23-16  
VARCHAR2 data type, 7-5  
variable  
    cursor  
        *See* cursor variable  
    in C external subprogram  
        global, 18-40  
        static, 18-40  
VERSIONS\_ENDSCN pseudocolumn, 16-9  
VERSIONS\_ENDTIME pseudocolumn, 16-9  
VERSIONS\_OPERATION pseudocolumn, 16-9  
VERSIONS\_STARTSCN pseudocolumn, 16-9  
VERSIONS\_STARTTIME pseudocolumn, 16-9  
VERSIONS\_XID pseudocolumn, 16-9  
view  
    constraint on, 10-1  
    editioned, FOREIGN KEY constraint and, 24-9  
    editioning  
        *See* editioning view  
    materialized, 1-8  
        that depends on editioned object, 24-4  
virtual column, 24-4  
VPD policy, editioning view and, 24-35

## W

---

WAIT commit redo option, 6-6

WAIT option of LOCK TABLE statement, 6-14  
web application, 14-1  
    implementing, 14-2  
    state and, 14-25  
web services, 17-12  
web toolkit  
    *See* PL/SQL Web Toolkit  
white list  
    *See* ACCESSIBLE BY clause  
wrap utility, debugging and, 11-34  
writability of editioning view, 24-22  
write-after-write dependency, 16-14

## X

---

xa\_open string, 19-8  
XMLType data type, 7-19  
X/Open Distributed Transaction Processing (DTP)  
    architecture, 19-2

## Y

---

YY datetime format element, 7-15

